Bachelor Thesis

DynaBench – Testing and Evading Dynamic Analysis in Android Applications

University of Twente
BSc. Technology, Liberal Arts and Sciences

author:
Timme Bethe

December, 2021

supervisors:
dr.ir. Andrea Continella (SCS)
ir. Thijs van Ede (SCS)
co-supervisor:
dr. Pascal Wilhelm (ATLAS)

Abstract

Today's smart phones are a treasure trove of private information. Access to this data is often abused by apps. In response, the research community has proposed tools to analyse apps for privacy leaks. To evaluate these tools micro-benchmarks exist for static analysis tools but not for dynamic analysis tools. These existing micro-benchmarks are insufficient to test dynamic analysis since dynamic analysis has unique problems. In particular, apps can recognise analysis is happening and stop malicious behaviour. Techniques to apps use to do this recognition we call 'evasion techniques'.

We present DynaBench, an open source micro-benchmark for dynamic analysis tools. DynaBench functions by testing if the tool is robust against these evasion techniques. This is done by actively trying to evade analysis using different techniques. We tested DynaBench against the VirusTotal platform and found that seven out of thirteen techniques were succesful in evading dynamic analysis. We conclude that is remains relatively easy to evade dynamic analysis.

1 Introduction

Many mobile applications have been found to leak private data [9, 18]. We say data is leaked when it is moved to an external destination without authorisation [23]. This problem of data leakage is particularly troubling in mobile applications, because mobile phones have access to much private information that can potentially be leaked. Mobile phones are equiped with GPS, microphones, cameras and many more sensors and also play an integral part in most of our lives, resulting in them storing a lot of private data. For example, mobile phones can have access to our private information such as our location, private texts, photos, contact list, e-mails and credit card information.

One of the main mobile operating systems, Android, makes an effort to prevent apps from accessing private data by introducing a permission system. Accessing data that is privacy sensitive require that the app requesting the data holds the appropriate permissions. Otherwise , the operating system denies the app access. However, many apps request permissions they do not need and users install apps without fully knowing what the permissions mean [2], so permissions

do not solve all problems.

These privacy issues are not unique to Android. However, in this paper we focus on Android only. In Februari 2021 Android had a 72% world-wide market share [13], so we target the majority of mobile devices.

If we cannot prevent apps from leaking our private data, the next best thing is to know when data leaks happen. This information can then be used to decide whether the app is permitted to leak this data.

Considerable research has been done to create tools that detect whether an app is leaking private data. These tools do this by analysing the behaviour of the app. This analysis can either be done statically or dynamically. Static analysis works by analysing the byte-code of an app and reasoning what the app might do. Dynamically analysis works by running that app in a sandboxed environment and observing what it does. Behaviour indicating data leakage could be any private data leaving the app, e.g., the apps sends its location over the internet, writes it to a file, or sends it using SMS.

Most research has been focused on static analysis, with a plethora of tools being proposed. Some well-known examples of static analysis tools for detecting privacy leaks are AndroidLeaks, Amandroid, IccTA and FlowDroid [9, 22, 11, 3].

However, there are ways for app developers to shut down most possbilities for static analysis. One way is *packing*, which encrypts crucial parts of the app's byte-code. Another technique is called *dynamic code loading*, where new code is fetched from somewhere, like a remote server, and then loaded and executed by the app while it is running. This means that some of the possibly malicous code is not yet present at the time the app is being statically analysed.

Both packing and dynamic code loading have legitimate uses. For instance, a bank could use packing of their mobile banking apps to make reverse-engineering more difficult. Also, large apps might use dynamic code loading to keep the initial size of the app small and download addons when needed. Thus, using these techniques does not indicate malicious intent.

To overcome some of these issues with static analysis, we can use dynamic analysis, for which tools have also been proposed. Some examples of dynamic analysis tools are TaintDroid, Agrigento [8, 6], and DyDroid which uses a both static and dynamic analysis [17]. During dynamic analysis, packing will not work, since the app will decrypt itself in order to run. Similarly, dynamic analysis can trigger dynamic code loading such that the dynamic code will be fetched and executed and thus the behaviour can be analysed.

Unfortunately, dynamic analysis cannot solve the issue that the code that is fetched from a remote server can be modified at any time. It could contain benign code at first, to prevent the app from being flagged during analysis, then this code could be modified to be malicious at a later time.

So, dynamic analysis can solve some limitations that static analysis has. However, dynamic analysis has its own limitations. One limitation is that dynamic analysis requires heavy instrumentation. We need to set up a sandboxed environment, run the app in this environment using a real device or an emulator while being able to observe and capture its behaviour. This requires much heavier instrumentation compared to static analysis.

Another important limitation of dynamic analysis is the issue of code coverage. When navigating through the app, we cannot try every possible input combination. As a result, some behaviour might not be triggered during dynamic analysis because it requires an input combination we have not tried.

The aim of this research is to create a way to test and evaluate dynamic analysis tools, such as TaintDroid and Agrigento, and ascertain their strengths and weaknesses. What kind of evasion techniques are effective to evade dynamic analysis?

To do this, we propose DynaBench, a microbenchmark consisting of 13 apps, that each try to evade being detected by dynamic analysis using a unique technique while leaking private data to a personal server controlled by us. By letting a dynamic analysis tool analyse an app from DynaBench, we can see if the tool is equipped to deal with the evasion technique implemented in the app. Either the analysis tool will raise a warning, stating that the app is suspicious or dangerous, or the analyiser will not raise a warning, meaning our app successfully evaded analysis. Furthermore, we can also infer what techniques succesfully evaded dynamic analysis

by identifying which app sent private data to our server.

For the remainder of the paper, when we refer to an analysis tool, we assume this is a dynamic analyis tool aimed at analysing Android apps.

To test DynaBench, all apps were submitted to the VirusTotal platform using an automated pipeline. Seven out of 13 apps did not leak data to our server, meaning they successfully evaded dynamic analysis of the VirusTotal platform. For example, waiting for the device to have slept for 2 hours since its last reboot managed to evade analysis as well as inspecting if the accelerometer contained noise in its readings. It is clear that evading dynamic analysis remains rather easy and can be done in a myriad of ways.

The major contributions of this work are:

- A review of known evasion techniques for dynamic analysis.
- DynaBench, a benchmark for testing and evaluating dynamic analysis tools for Android apps. DynaBench is made open source.[1]
- An insight into the capabilities of AntiVirus software in terms of dynamic analysis of Android applications.

2 Motivation

Many tools, based on both static and dynamic analysis, have been proposed to detect privacy leaks. Evaluating and comparing these tools reliably has proven difficult. Many papers do not include an exact list of app names used to perform the evaluation, making the results irreproducible. [16] However, a more pressing problem is that evaluations used in many of the proposed tools consist of testing the tool on a large number of the most popular apps at that time from the Google Play Store or other public sources. [9, 6] This approach makes it impossible to accurately to determine the performance of the tool, since for the apps used in these evaluations it is not known whether they leak data, nor in what way, i.e., these evaluations lack well-defined ground truth. Moreover, many papers simply report the number of leaks found. This is problematic since it encourages false positives as higher numbers of found leaks are often seen as better, as explained by Pauck, Bodden and Wehrheim. [14] In conclusion, there is a clear need for ground truth.

Luckily, a solution to address the missing ground truth has been proposed in the form of a micro-benchmark, i.e., a set of artificial apps that serve the sole purpose of leaking data in a known way. This way, we can create ground truth. The two most popular benchmarks are DroidBench and ICC-Bench [3, 11]. These benchmarks are large collections of apps, made to evaluate the effectiveness of static taint analysis tools. Both have proven quite successful and have been used by a number papers to evaluate their approaches or compare different approaches. [16, 14, 10, 22]

While benchmarks aimed at static analysis can be useful when evaluating a dynamic analysis tool, dynamic analysis has unique problems for which static benchmarks do not test. Many approaches used to evade dynamic analysis are based on the app recognising that it is being analysed and acting upon that knowledge by behaving differently not to arouse suspicion.

Since an app is never run during static analysis, such techniques do not work to evade static analysis. As such, these approaches are missing from benchmarks aimed at static analysis such as DroidBench and ICC-Bench. Currently, it seems that there are no benchmarks tailored to evaluating dynamic analysis.

Thus, while there is sufficient ground truth for static analysis, ground truth for dynamic analysis is still missing. This is why we created DynaBench, to create ground truth for dynamic analysis. The idea is to create a new set of apps that can function as a benchmark. These apps will employ many different ways to evade dynamic analysis tools. This way, by using the benchmark to test dynamic analysis tools, it will become clear what kind of evasion techniques work to evade the dynamic analysis tool.

3 Threat Model

To argue what techniques might be effective to evade dynamic analysis, we need to define what characteristics a dynamic analysis tool has. We can then try to exploit these characteristics in our evasion techniques.

- Firstly, we assume the analysis tool analyses

[1] `https://github.com/tbethe/dynabench`

our app in a sandboxed environment, i.e., a closed off environment where the tool has complete control. Such an environment is often just called a sandbox. The analysis tool needs full control over the app that it is testing to properly capture the app's behaviour. Another justification for this assumption is that performing dynamic analysis in a sandbox enables reproducibility, since the sandbox can be set up in the same way each time.

This assumption is useful, because sandboxes produce artifacts that enable us to distinguish between a normal environment and a sandbox.

- We also assume dynamic analysis is performed on an emulated device, instead of on real hardware. Emulation makes it easier to scale up analysis; the process can be parallellised without additional hardware. Hardware is expensive, which is why emulation is advantageous. Moreover, we assume that emulation is not used by real users. Emulation is not widespread when it comes to Android. This assumption would not hold if our analysis tool would target operating systems such as Windows or Ubuntu, since for these operating systems emulation is common-place.

  This assumption is useful because we can often distinguish between emulators and real hardware devices.
- Another assumption we make is that the analysis tool is time constraint, i.e., the tool has limited time to analyse our app. Dynamic analysis is often used in an industrial setting, such as by Google to filter apps submitted to their Play Store, or by anti-virus companies. In these industrial application of the analysis tools, the number of apps that need to be analysed is large and having analysis be done quicker is a major advantage.

  This assumption is useful because we can try to stall leaking data until after the analysis is done.
- Lastly, we will assume the dynamic analysis is done without human intervention. This assumption is justified by the same reason we assume the analysis tool is time constraint; in industrial application the number of apps is large and requiring human intervention would severely limit the number of apps that can be analysed.

  This assumption is useful because it means any navigation that will occur during the dynamic analysis to trigger different behaviour of our app is done programmatically. Programmatic navigation also produces artifacts which we can use to distiguish between normal users navigating through the app and an analysis tool navigating through the app.

4 Approach

Our aim is to create a way to test and evaluate dynamic analysis tools to the ascertain robustness of these tools. To do this, we propose DynaBench, a micro-benchmark specifically designed to evade dynamic alreadyanalysis. DynaBench is a set of apps, just like existing benchmarks like DroidBench and ICC-Bench. However, in DynaBench each app implements a unique evasion technique instead of a unique way to leak data like benchmarks for static analsis do. Such a technique attempts to recognise if the app is currently dynamically being analysed and outputs yes or no. Yes, if the technique concludes the app is being analysed, no otherwise. Then, the apps leaks data if the evasion technique output no, and does not leak data is the output was yes. An overview of this approach can be seen in Figure 1.

To test an analysis tool with DynaBench, we follow the procedure outlined in Figure 2. We let the analysis tool analyse each app in DynaBench individually. For each app this gives us two pieces of information. First, the analysis tool returns an analysis report, which will either flag our app as suspicious, dangerous or not. If the analysis tool flagged our app, our app did not evade the analysis and the analysis tool is resistant to the technique implemented in that app. Second, depending on the conclusion of evasion technique we will or will not have received a data leak in our server. From the existance of the data leak, or the absence of it we can infer information about the analysis tool. It tells us the conclusions the evasion technique drew when executed in the analysis tool.

To test what strengths and limitations analysis tools have, we want to test them with a diverse set of techniques. First, we require a list of techniques that evade dynamic analysis. This list should reflect techniques that are also found

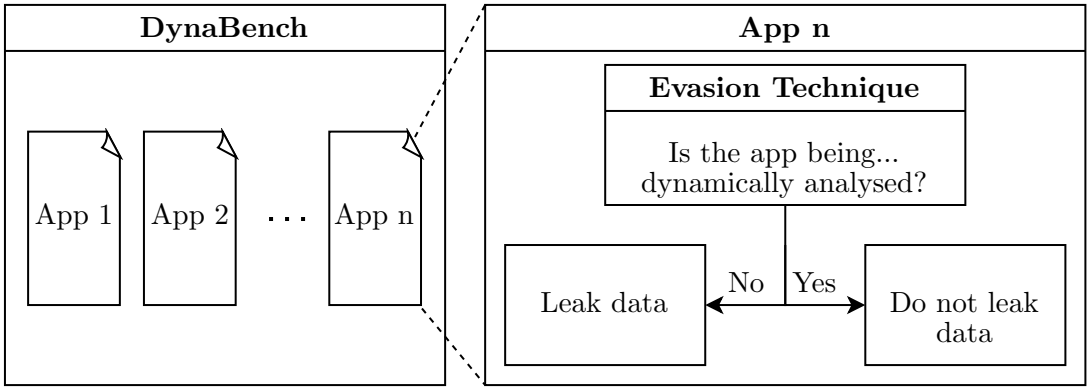

Figure 1: Overview of DynaBench and a magnified view of an individual app. Each app decides whether to leak data based on a unique evasion technique that decides if the app is being dynamically analysed.

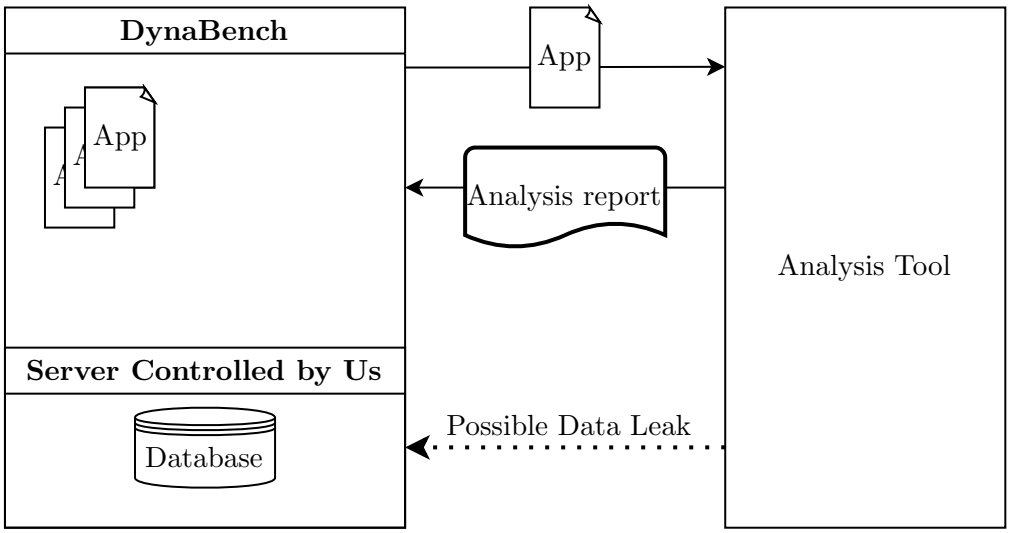

Figure 2: Overview of testing an analysis tool with DynaBench. Apps are sent to the analysis tool one-by-one. The analysis tool returns a report, and depending on the conclusion of the evasion technique, we receive a data leak.

'in the wild', i.e., found in apps used by normal users. Ideally, the list would be exhaustive, because then DynaBench would be able to spot all the weaknesses of an analysis tool. However, this is impossible, since some may not have been invented yet. Instead, our list attempts to provide a broad, albeit not full, spectrum of dynamic evasion techniques.

Lastly, we evaluate DynaBench by submitting the benchmark to an analysis tool, in our case the VirusTotal platform. VirusTotal is a platform that uses anti-virus engines of other companies to analyse files. This way we test our benchmark on multiple, different analysis tools.

4.1 Evasion Techniques

We performed a literature review to create an overview of techniques described in literature. We identify the following categorisations.

4.1.1 Baseline

To establish a baseline, we want an 'empty' technique, i.e., a technique that will always conclude that the app is not being analysed. This is useful to identify if the analysis tool can detect the data leak in the first place.

A. The empty technique.

4.1.2 Sandbox Detection

As explained in our threat model, we assume the analysis tool performs analysis inside a sandbox. We might be able to spot differences between a sandbox and a real environment. These differences are exploited in the following techniques.

B. We can look at the uptime of the device itself and only leak if it exceeds some amount of time. [19] The assumption here is that the device used for testing is reset after each analysis. [12]
C. Instead of looking at uptime, we can look at (deep) sleep time, i.e., the time the device spends with its CPU off, screen off or waiting for external input. This approach is better than looking at uptime, because even if the device is not reset after each analysis, it is unlikely the device has spent a long time idling, since its sole job is to perform analysis. Moreover, letting their device spend long periods of time idling is normal for normal users.
D. We can wait for a specific number of UI steps to have been taken before leaking. Since dynamic analysis is time constraint, we assume that the number of UI steps taken by the dynamic analysis tools is relatively small and the number of steps taken by a normal user will – quickly – exceed this. [20]
E. We could only leak if another, immensely popular, app is installed, such as Angry Birds, as was the case with the malware CrazyBirds. [20]. This could work because we assume sandboxes are clean.

4.1.3 Imperfect Emulation

As explained in the threat model, dynamic analysis is often done using an emulator as opposed to using hardware. It is difficult to emulate a mobile phone perfectly. The following techniques exploit this imperfect emulation.

F. The Android API exposes a `Build` class, describing hardware properties of the device. On the Google emulator, many properties are set to values that are indicative of the device being an emulator. For example, Google calls many of its emulated hardware components 'goldfish' or 'ranchu' (goldfish in Japanese). [21, 19]
G. Similarly to the `Build` class, the CPU properties in the file `/proc/cpuinfo` also has values indicative of emulation, such as 'virtual processor' or, again, 'goldfish'. [21]
H. On older versions of the Google emulator, the files indicating the minimum and maximum CPU frequencies were missing. [7] However, we found that these files are present in newer versions of the emulator.
I. One proposed technique is to fingerprint the CPU performance. Vidas and Christin differences in the duration of expensive calculations between hardware and emulated devices. [21]
J. An emulator also needs to emulate the sensors of the device. The current emulation of sensors is basic and can easily be distinguished from real devices. [15, 5, 4, 19] For example, the accelerometer returns the same values if the position of the device is not changed, i.e., there is no noise in the measurements.
K. Another simpler technique, could be to just check the number of sensors the device has. On the two real devices we tested, there were more then 25 sensors, while different versions of the Google emulators all reported less than 25 sensors. This was also found in different research by Vidas and Christin. [21]
L. The battery also needs to be emulated. Currently, the Google emulator does not change it's battery values. It is fixed at either 50% or 100% [7]. Therefore, we can retrieve information about the battery and see if the values are changing, to distinguish between emulator and real device.
M. By default, the Google emulator runs in the same subnets. Petsas et al. report that the emulator only runs in `10.0.2.0/24`. [15] However, our own experiments also show another subnet: `192.168.232.0/24`. While it is possible for real devices to connect to these two subnets, it is not a common occurrance.
N. As an extension, you could check if the device is running on a subnet different from the two above or even turn this technique into a logic bomb by only leaking once the device has connected to x number of different subnets.

4.1.4 Time Constraint Exploitation

As explained in our threat model, we assume dynamic analysis usually applied in an industrial setting to analysis a large number of apps. Therefore, we assume that dynamic analysis is time constraint. This time constraint is exploited in the following evasion techniques.

O. Wait for the app to be started up x times after being installed and only leaking afterwards. The assumption here is that dynamic analysis will only need to start the app a small number of times. Moreover, normal usage of apps will open the app far more often.
P. Wait for x amount of time after the app has been installed or after the first start up before leaking. We assume analysis should be done quickly after the app is installed on the test device. We want to choose x such that analysis is done is a shorter time. [19]

4.1.5 Reverse Turing Tests

A Turing test tests the ability of a machine to exhibit behaviour indistinguishable from human, judged by a human. Here, by reverse Turing test we mean the same test, to tell apart a machine and a human, but then with a machine being the judge.

During dynamic analysis, often, input such as touching the screen is generated programmatically. So, if we can tell apart this programmatically generated input and the input of a human, we can decide to only leak data when the input is not generated programmatically, i.e., is produced by a human [5]. The following techniques were all described by Wenrui et al. [7]

Q. UI trap 1: Create UI elements that are only interactable programmatically, such as an invisible 1x1 pixel button. If this button is triggered, it must have been triggered programmatically.
R. UI trap 2: Isolated activity: Create an an Android activity that cannot be triggered through normal navigation. If this activity is triggered, it must have been triggered programmatically.
S. Event sequences: Dynamic analyse often inputs event quickly (to analyse faster) and often do this in a uniform way. The time between input events might be uniform or repeated events are very similar, e.g., when swiping across the screen.

4.1.6 Logic Bombs

When some code is only triggered after some very specific conditions, we call this a logic bomb. In relation to evading dynamic analysis, the general idea is to use logic bombs to create logic that is unlikely to be triggered during dynamic analysis, but is quickly triggered by normal users.

T. Only leak if a specific sequence of UI steps have been taken. The problem with this approach is that it is difficult to craft a sequence of UI steps that is still likely to be triggered by normal users.
U. Wait for a certain system time to perform the leak. [1] The idea is that if the app only leaks data at one time during the day, it is unlikely that the app will be analysed exactly during this time.

4.1.7 Dynamic Code Loading

Dynamically loaded code is code that is loaded while the program is already running. This code could be loaded from anywhere, like a file or even a remote server. This means we can alter the code of our app from a distance.

V. Dynamically load the code that is responsible for leaking the data from a remote server.
W. The technique above can be extended by replacing the dynamically loaded code with code that does not leak data, only during the analysis of the app.

4.2 Benchmark Setup

In essence the benchmark is a set of Android applications. Each app consists of two components, an implemented technique which tries to determine whether the app is being dynamically analysed and a data leak.

First. What is important is that the data leak is the same for each app in the benchmark, since we want to discover what evasion techniques work against a given analysis tool. Otherwise, different apps might produce different results

which have nothing to do with the implemented evasion technique.

Second. Each app implements one technique to evade dynamic analysis. This should be different for each app.

Lastly, the configuration of the apps should also be the same, or at least as similar the implemented technique allows. By configuration we mean everything in the app that is not part of the technique it implements. For example, we should make sure the app support the same Android API versions and have the same permissions, unless the implemented technique requires additional permissions. This minimises differing triggers in the analysis tool that have nothing to do with the evasion technique that is implemented.

4.3 Benchmark Validation

We want to test if DynaBench is actually useful when it comes to testing dynamic analysis. To what extend can DynaBench uncover limitations of existing dynamic analysis tools?

To answer this question we will perform an experimental validation where we submit DynaBench to an analysis tool. If the benchmark reveils that the analysis tool can be bypassed by evasion techniques implemented in the benchmark, then we have sucessfully determined the (un)robustness of the tool by uncovering limitations.

5 Implementation

From the evasion techniques identified from literature, we choose a selection to implement. In general, the more the better, but as time is limited, DynaBench came to consist of thirteen techniques in total, including a baseline. This selection has techniques from almost every category outlined in our approach. We want to test a wide range of different techniques to give to benchmark breadth.

All apps use different techniques to determine if it is being analysed, but all apps leak the same data. They do so via a HTTPS POST request to a hardcoded URL of a webserver set up for this research. The data that is leaked is location data, i.e., the latitude and longitude of the current location reported by the device. Location data can only be retrieved if the app has one of two permissions:

- `ACCESS_COARSE_LOCATION`
- `ACCES_FINE_LOCATION`

Both of these permissions have a protection level called 'dangerous'. They have this protection level because the permissions are judged to pose a risk to user's privacy. All apps request the fine location permission to leak data. Now we will go over each app in more detail.

DynaBench can be easily extended by adding apps that add new evasion techniques, since each app mostly exists on its own. What is important to consider when extending DynaBench is to keep the configuration the same, i.e., make sure the new app only differs in the code that implements the evasion technique.

5.1 Always Leak

This app is based on technique A. This app does not attempt any evasion and will always try to leak the location. It serves as a sort control group. Any analysis tool, based on both static and dynamic analysis, should be able to detect the leak and flag it.

5.2 Accelerometer

This app is based on technique J. The app attempts to detect emulation by requesting data from the accelerometer sensor. Real accelerometers have noise in the data they return. For this reason it is almost impossible for consecutive readings to return the same data. However, the current implementation of the Google emulator returns the exact same readings if the position of the emulator is not changed.

Because the sensor emulation of the accelerometer is currently very simple, it is trivial to distinguish between real and emulated data. We retrieve 10 consecutive readings from the accelerometer and only leak if all readings are different.

5.3 Battery Level

This app is based on technique L. Emulated devices do not have a real battery, so the battery level must be emulated. Current implementations often use constant battery levels of either 50%, or 100%. To exploit this imperfect emulation, the apps checks the battery level and

only leaks if it has seen a battery level above and below some threshold. The app needs to observe the battery level below 40% and above 60%. This condition can only be satisfied if the battery levels are not constant.

5.4 CPU Properties 1

This app is based on technique G. The app checks CPU information, by reading from `/proc/cpuinfo`. Whether a device has a real of emulated CPU can often be identified by this information. Most notably, Google's emulator has values like 'virtual processor', 'Android' and 'goldfish'. These values will never be present on a real device. Goldfish is a name Google often uses for component names in their Android emulator. Based on this info the app decides whether it is running on a real or emulated device.

5.5 CPU Properties 2

This app is based on technique H. Real devices always have files indicating the minimum and maximum processor frequency. On emulated devices these files are often missing, probably because an emulated CPU does not have a real processor frequency. This app decides whether it is running on a real or emulated device based on if these files exist.

5.6 Hardware Properties

This app is based on technique F. The Android framework exposes the `Build` class, which specifies build details of the device. A lot of the values specified give away whether the device is emulated or not. For example, for emulated devices often specifiy names like: goldfish, ranchu (goldfish in Japanese), generic, sdk, emulator.

5.7 Subnet

This app is based on technique M. Google's emulator, by default, runs in one of two subnets:

- `192.168.232.0/24`
- `10.0.2.0/24`

While it is not impossible that a real device would connect to a network with one of these subnets, it is unlikely since these subnets are not generally used elsewhere. Therefore, checking whether the subnet is one of these subnets is a good heuristic to see if the device is an emulator.

This apps checks the subnet and assumes the device is emulated if it is running on either of these subnets.

5.8 Sleeptime

This app is based on technique C. It is based on the assumption that devices used for dynamic analysis do not spend a lot of time in 'deep sleep', i.e., when the CPU is off, the screen is dark or the device waiting for external input. The assumption is that devices used for dynamic analysis are immediately used for analysis after being turned on. The app calculates the deep sleep time since last reboot of the device and only leaks if this duration is longer than 2 hours.

5.9 Uptime

This app is based on technique B. We assume devices for dynamic analysis only stay turned on for a limited time; only during analysis. Thus, this app only leaks if it is started 4 hours after the device has booted.

5.10 Wait after Start Up

This app is based on technique P. This technique is based on the assumption that there are time contraints to dynamic analysis. The result of the analysis should be available quickly after installing the app. Therefore, this app only leaks after being installed on the device for at least 7 days.

5.11 Startup Counting

This app is based on technique O. This strategy is based on the assumption that dynamic analysis only starts an app a limited number of times after installing. Furthermore, we assume that this number is lower than when an app is used by an actual person. Thus, the app keeps track of the number of times the app is started. It only leaks after starting up the app at least 10 times.

5.12 Dynamic Code Loading

This app is based on technique V.

This app does not contain any code to leak private data. However, after a button is pressed,

it fetches code from our server. If the button is pressed again, the code is promptly executed. The fetched code is the code performing the leak.

5.13 Popular App Installed

This app is based on technique E We assume that the sandbox is 'clean', i.e., it only has the minimum amount of software installed to perform the analysis. So, the strategy is to check if a massively popular app is installed that is unlikely to be installed in a sandbox. We have decided to use the WhatsApp in our benchmark, because it is used all over the world.

This apps decides the device is real when it has WhatsApp installed.

6 Experimental Validation

All apps were tested by running them on the phone that is daily used by the author. All apps do leak data when run on this device.

The remaining question is of this experimental validation is to show that DynaBench is also useful. In other words, it can point out the strengths and weaknesses of dynamic analysis tools.

The main difficulty in showing this, is finding suitable analysis tools to test with. Such tools have to be publicly available and work with up-to-date versions of Android. In our case an up-to-date version means Android 6 or higher. While many dynamic analysis tools have been described in research, only a few are both freely available and support Android apps built for Android 6+. Often such research is done in collaboration with industry resulting in the tool being kept private. Other tools that have been developed have not been updated to work with recent versions of Android. [8]

One of the analysis tools that is suitable is the VirusTotal platform, which is used for this experiment. VirusTotal has an analysis service to which you can submit files for analysis. These files are then analysed by analysis tools of various anti-virus companies. For Android apps, it dynamic analysis.[2] It does so by outsourcing the analysis to around 70 anti-virus companies such as Kaspersky, MalwareBytes, McAfee, Symantec and F-Secure. This means we submit our benchmark to multiple dynamic analysis tools at the same time.

The idea is straight forward. We submit the apps in the benchmark to VirusTotal to be analysed. As explained in our approach and Figure 2, this will get us two kinds of insights. Firstly, the VirusTotal will return an analysis report, stating whether they have found that the apps in the benchmark leak private information. Secondly, if an app decides to leak data and has a working internet connection, we should receive a log in our database from which we can infer information about the analysis tool.

6.1 Setup

To receive the leaked data from an app, we had a webserver running a minimal REST API, receiving HTTPS POST requests. The endpoint of the webserver was hardcoded in the benchmark apps. Whenever an app leaks data, it sends two things to the webserver: The name of the app itself and the leaked data, the longitude and latitude of the location of the device running the app. The webserver only stores a log of the name of the app that leaked the data and a timestamp of when the leak happenend.

The apps were submitted to VirusTotal in an automated way, using a pipeline. First, we need to build the APKs from source. Before the apps are built, the version number of the app is incremented. This is done to change the hash of the APK. We do this to trick VirusTotal into thinking that it is dealing with a new file. This is necessary to avoid VirusTotal returning an analysis from a previous time the pipeline was run. Second, the APKs are submitted to VirusTotal via their open API and the result is report is retrieved.

6.2 Results

VirusTotal did not flag any of the submitted APKs as dangerous or suspicious in the analysis reports that were returned. From the logs collected in our webserver we see that six out of the thirteen apps have leaked data while being under analysis. Which apps did or did not leak data can be seen in Table 1.

It should be noted that most of the data leaks were received by our server long after the analysis

[2] https://support.virustotal.com/hc/en-us/articles/115002719069-Reports

reports were returned. For each app, VirusTotal returned the analysis report in no more than a few minutes, but our server kept receiving data leaks for about 24 hours after the last analysis report was returned by VirusTotal. These reports stated that the analysis had been completed.

Moreover, each app that did leak data, did so multiple times. However, the number of times varied when the experiment was repeated, making it difficult to use this information to draw any conclusions. The set of apps that leaked data did not change when the experiment did not differ.

Application	**Leaked Data** (yes/no)
Always Leak	yes
Accelerometer	no
Battery Level	no
CPU Properties 1	yes
CPU Properties 2	no
Hardware Properties	yes
Subnet	yes
Sleeptime	no
Uptime	yes
Wait after Start Up	no
Startup Counting	no
Dynamic Code Loading	no
Popular App Installed	yes
Total with data leaks:	6/13

Table 1: We show for each benchmark application whether it leaked data during analysis by VirusTotal.

6.3 Discussion

Firstly, the reports returned by VirusTotal did not flag any apps as dangerous or suspicious. This is not surprising since VirusTotal is an anti-virus platform. Leaking a location is not seen as malicious behaviour. After all, there are many situations where sending location data over the network is completely acceptable. Consequently, flagging all these apps as suspicious will lead to false positives.

While false positives might be the reason not to flag the app as suspicious, anti-virus engines could list what kind of private data the app is leaking. Then, the person who inspects the report can judge for themselves whether this data leak is (il)legitimate.

Secondly, some apps did and others did not leak data to our webserver. These results are shown in Table 1. More than half of the techniques did not leak data, meaning they evaded analysis.

There is no reason why techniques that drew the correct conclusion to evade detection by these anti-virus engines by not sending a data leak could not be used to disguise more nefarious behaviour than data leakage, making these results more significant. Moreover, we can infer a great deal about the dynamic analysis that VirusTotal subjected our benchmark apps to from these results.

Thirdly, it was curious that data leaks kept coming in after VirusTotal reported that the analysis was complete. One explanation could be that the analysis report returned by VirusTotal only includes insights gained from static analysis, which is much faster. Then, the submitted app is only dynamically analysed at a later time.

Now we will discuss each app in more detail.

6.3.1 Always Leak

Always Leak leaked data. This was expected. From this we can infer that the apps have a working internet connection while being analysed – although this be inferred from any other app that leaked data as well.

6.3.2 Accelerometer

Accelormeter managed to evade analysis. This gives us a strong indication that the dynamic analysis was performed on an emulator. Furthermore, this emulator only performs basic sensor emulation.

6.3.3 Battery Level

Battery Level also evaded analysis. This does not guarantee that their battery emulation is poor. This evasion could also be due to Battery Level behaving as a logic bomb; even if the device was real or had proper battery level emulation, for the app to leak data, the battery would have to be drained or charged for at least 20% for the leak to trigger. This in and of itself is a powerful logic bomb, regardless of whether the device has

a real battery or an emulated one. Real devices used for dynamic analysis are often permanently being charged, making it unlikely that the batter level changes at all, much less 20%.

6.3.4 CPU Properties 1 & 2

CPU properties 1 did not evade analysis. This app checks CPU properties from `/proc/cpuinfo`. Since we already deduced that the analysis must have happened in an emulator, it seems that the CPU properties were succesfully changed to non-suspicious values. What is surprising, is that CPU properties 2 did not leak data. This app checked for the existance of files indicating the max and minimum CPU frequencies, which must have been missing. This is surprising because according to our own findings, these files were added to newer versions of the Google emulator. This leads us to believe that either a different emulator or an older version of the Google emulator is used during analysis.

6.3.5 Hardware Properties

Hardware Properties did not manage to evade analysis. Just as with CPU Properties 1, it seems that the values in the `Build` class were given realistic values.

6.3.6 Subnet

Subnet also failed to evade analysis. The device was not running on one of the default subnets during any dynamic analysis.

6.3.7 Uptime

Uptime did not work. It seems that our assumption that devices are reset after each analysis was wrong. Perhaps such a reset is not strictly necessary. Moreover, resetting the sandbox takes time, which is disadvantageous. This may be another reason why no reset appears to happen between analyses.

6.3.8 Sleeptime

Sleeptime did manage to evade analysis. Sleeptime is a stronger version of Uptime since we require the device not only to have been booted for a time, but this time should also have been spent sleeping. It seems that our assumption that devices used for dynamic analysis are unlikely to sleep was correct, since the uptime technique did not work and this technique did work.

6.3.9 Wait after Start Up

Waiting after start up was also succesful. It seems our assumption was correct.

6.3.10 Startup Counting

Startup counting also evaded analysis. Also here it seems like our assumption was correct; the app was not restarted more than 10 times.

6.3.11 Dynamic Code Loading

Dynamic Code Loading also evaded analysis and did not send data to our server. This is surprising because strictly speaking, this app does not try evade analysis at all. If the button is pressed twice, the app will leak data. We know the button was not triggered twice. This makes us suspect that the dynamic analysis used to analyse the apps was quite primitive, since it does not have proper UI nagivation.

It is easy to adapt dynamic analysis to trigger the button twice. However, the idea of dynamic code loading is still very powerful. The extension disussed in technique W, where the malicious code is swapped out during analysis is impossible to deal with for analysis tools since the behaviour the tool is looking for does not exist at that moment.

Do realise that for this to work, the actor in control of the remote server from where the dynamic code is being fetched must know when the analysis is going to occur.

6.3.12 Popular App Installed

Lastly, Popular App Installed. In our case the popular app was WhatsApp. This app did leak data. It seems that the dynamic analysis environment was not devoid of any apps; an attempt was done to make it look like a real phone and WhatsApp was installed.

7 Limitations

While we managed to get to know more about the way VirusTotal conducts dynamic analysis by

examining what benchmark apps did or did not leak data, the analysis reports from VirusTotal did not mention any dangerous or even suspicious behaviour. If the benchmark apps employed more aggresive data leaks that are caught more easily, we might be able to find that some techniques still get flagged as suspicious despite them succesfully evading the dynamic analysis.

Furthermore, the benchmark currently only consists of relative few apps; there are only thirteen. Quite some techniques that might succesfully evade analysis are not included in DynaBench. Increasing the number of techniques clearly helps decrease this limitation. There are many techniques that were identified during the literature review in the approach or theorised, but were not implemented. In particular, all are reverse Turing tests were not implemented.

Moreover, we tested DynaBench on anti-virus engines only. These dynamic analysis tools are not focused on privacy leaks. This is why it is not surprising that they did flag any apps as suspicious or dangerous. Perhaps, we would get different results if we tried to evaluate DynaBench with analyses tools that are focused on privacy leaks, such as Agrigento [6], to better evaluate the benchmark.

Lastly, we know that most of the dynamic analysis our apps were subjected to happened after VirusTotal returned the analysis report. It could be that the analysis reports get updated by VirusTotal when different anti-virus engines return their findings from dynamic analysis to VirusTotal. It could be the case that some apps were flagged as suspicious by one of the anti-virus engines, but the report was updated after we retrieved it.

8 Future Works

The first next step would be to deal with the limitations. They are all relatively easy to deal with, but require time and effort. A different data leak could be used in the benchmark apps that might result in gaining more information about the analysis tool.

Another limitation that should be dealt with is to test the benchmark on an analysis tool that focusses on privacy leaks, instead of malware in general. This might yield different results.

Moreover, the number of apps in the benchmark could be increased. Testing more techniques increases the breadth of the benchmark and thereby also its effectiveness. The techniques that we described in our approach are an obvious starting point when deciding what techniques should be implemented next.

Furthermore, as mentioned in the limitations it would be interesting to retrieve the reports from VirusTotal after a week or so, to see if the reports have changed.

DynaBench could also disguise malware instead of data leaks. It would be interesting to see how effective this benchmark would be to test anti-virus tools if the data leak was replaced with malicious behaviour as found in malware.

Lastly, to increase the flexibility of DynaBench, some architectural changes could be made. Currently, to change the way the benchmark leaks data, each app would have to be changed individually. If this flexibility would exist, we could test an analysis tool with different data leaks and see if the resulst are different. Moreover, as mentioned above, this would also make it easier to repurpose DynaBench to test anti-virus engines instead of analysis tools focused on data leakage.

9 Conclusions

We set out to create ground truth to test dynamic analysis, since dynamic analysis faces unique challenges that are not present when testing static analysis. We did so by propsing DynaBench, a micro-benchmark for testing and evading dynamic analysis. Moreover, we did literature review to create a list of known dynamic analysis evasion techniques and implemented a subset of those techniques in DynaBench. The benchmark is a set of apps that all implement an evasion technique. This technique attempts to figure out if the app is dynamically being analysed and the app leaks data based on the conclusions drawn by this technique.

Seven out of thirteen of our apps managed to evade analysis. This shows us that it is still quite easy to evade dynamic analysis. Evading analysis can be done with a variety of different techniques; more than half of the techniques managed to evade analysis. Moreover, our benchmark does

what we want it to do. DynaBench succesfully ascertained what techniques dynamic analysis of VirusTotal is vulnernable to.

Our experimental validation also shows that such a benchmark is needed. Detection by a successful commercial product such as VirusTotal can be circumvented by relatively simple evasion techniques. This need becomes even more clear when we consider that these techniques can also be used to evade dynamic analysis for malware.

References

[1] Amir Afianian et al. "Malware Dynamic Analysis Evasion Techniques: A Survey". In: *ACM Comput. Surv.* 52.6 (2019).

[2] Iman M. Almomani and Aala Al Khayer. "A Comprehensive Analysis of the Android Permissions System". In: *IEEE Access* 8 (2020), pp. 216671–216688.

[3] Steven Arzt et al. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps". In: *SIGPLAN Not.* 49.6 (2014), pp. 259–269.

[4] Jacob Boomgaarden et al. "Mobile Konami Codes: Analysis of Android Malware Services Utilizing Sensor and Resource-Based State Changes". In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2016.

[5] Alexei Bulazel and Bülent Yener. "A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web". In: *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*. ROOTS. Vienna, Austria: Association for Computing Machinery, 2017.

[6] Andrea Continella et al. "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis". In: *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, 2017.

[7] Wenrui Diao et al. "Evading Android Runtime Analysis Through Detecting Programmed Interactions". In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. WiSec '16. Darmstadt, Germany: Association for Computing Machinery, 2016, pp. 159–164.

[8] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Transactions on Computer Systems* (2014).

[9] Clint Gibler et al. "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale". In: *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*. TRUST'12. Vienna, Austria: Springer-Verlag, 2012, pp. 291–307.

[10] Michael I. Gordon et al. "Information Flow Analysis of Android Applications in DroidSafe". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[11] Li Li et al. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 280–291.

[12] Dominik Maier, Mykola Protsenko, and Tilo Müller. "A game of Droid and Mouse: The threat of split-personality malware on Android". In: *Computers & Security* 54 (2015). Secure Information Reuse and Integration & Availability, Reliability and Security 2014, pp. 2–15.

[13] *Mobile Operating System Market Share Worldwide*. URL: `https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201912-202106` (visited on 2021-12-05).

[14] Felix Pauck, Eric Bodden, and Heike Wehrheim. "Reproducing Taint-Analysis Results with ReproDroid". In: *Software Engineering 2020*. Ed. by Michael Felderer et

al. Bonn: Gesellschaft für Informatik e.V., 2020, pp. 123–124.

[15] Thanasis Petsas et al. “Rage against the Virtual Machine: Hindering Dynamic Analysis of Android Malware”. In: *Proceedings of the Seventh European Workshop on System Security*. EuroSec ’14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014.

[16] Lina Qiu, Yingying Wang, and Julia Rubin. “Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 176–186.

[17] Z. Qu et al. “DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications”. In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017, pp. 415–426.

[18] Joel Reardon et al. “50 Ways to Leak Your Data: An Exploration of Apps’ Circumvention of the Android Permissions System”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 603–620.

[19] Lars Richter. “Common weaknesses of android malware analysis frameworks”. In: *IT Security Conference, University of Erlangen-Nuremberg during summer term*. 2015, pp. 1–10.

[20] Kimberly Tam et al. “The Evolution of Android Malware and Android Analysis Techniques”. In: *ACM Comput. Surv.* 49.4 (2017).

[21] Timothy Vidas and Nicolas Christin. “Evading Android Runtime Analysis via Sandbox Detection”. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 447–458.

[22] Fengguo Wei et al. “Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps”. In: *ACM Trans. Priv. Secur.* 21.3 (2018).

[23] *What is Data Leakage?* 2021. URL: `https://www.forcepoint.com/cyber-edu/data-leakage` (visited on 2021-12-09).