AndroNeo: Hardening Android Malware Sandboxes by Predicting Evasion Heuristics

Yonas Leguesse, Mark Vella, Joshua Ellul

AndroNeo: Hardening Android malware sandboxes by predicting evasion heuristics

Yonas Leguesse, Mark Vella, and Joshua Ellul

University of Malta, Msida MSD 2080, Malta
yonas.leguesse.05@um.edu.mt, mark.vella@um.edu.mt, and joshua.ellul@um.edu.mt

Abstract. Sophisticated Android malware families often implement techniques aimed at avoiding detection. Split personality malware for example, behaves benignly when it detects that it is running on an analysis environment such as a malware sandbox, and maliciously when running on a real user's device. These kind of techniques are problematic for malware analysts, often rendering them unable to detect or understand the malicious behaviour. This is where sandbox hardening comes into play. In our work, we exploit sandbox detecting heuristic prediction to predict and automatically generate bytecode patches, in order to disable the malware's ability to detect a malware sandbox. Through the development of AndroNeo, we demonstrate the feasibility of our approach by showing that the heuristic prediction basis is a solid starting point to build upon, and demonstrating that when heuristic prediction is followed by bytecode patch generation, split personality can be defeated.

Keywords: Android, Malware sandbox hardening, Sandbox evasion heuristics, Bytecode patching

1 Introduction

Android powers most mobile devices, and has recently surpassed Windows to become the Internet's most used operating system [1]. AntiVirus company McAfee reported [2] that in Q4 (2016) they witnessed a 72% increase of unique mobile malware samples collected in Q3, with over 2.4 million detections in Q4 alone. These staggering numbers are rendering automated malware analysis tools essential for analysts. A popular approach for automated malware analysis involves the use of malware analysis sandboxes, where the analyst sets up an environment in which a malware sample can run whilst its relevant operations and behaviour is collected for analysis. Neuner et al. [3] provided an interesting comparison of available Android malware sandboxes, where system emulation plays a central role in the provision of a safe inspection environment. Sophisticated malware often use techniques that allow them to detect and thwart the sandbox's analysis. One popular approach is commonly referred to as split-personality malware [4], where the generic evasion approach is as follows:

```
if(isSandbox()){
    System.exit();
} else{
    continueMaliciousOperations();
}
```

This code pattern allows malware to behave in two different manners, i.e. executing `System.exit()` or `continueMaliciousOperations()`, depending on whether the environment is a sandbox or not. The distinguishing factors are established within `isSandbox()`, which is where the sandbox detection techniques are implemented. These checks can come in the form of a simple one-line verification or more complex checks such as performance analysis. Further still, malware can shift malicious operations to event handlers that are only likely to be triggered on real devices (e.g. SMS received events).

In this work, through the development of AndroNeo, we demonstrate the ability to automatically generate emulator based sandbox detection techniques, whilst providing the sandbox with the means to avert and disable the malware's split personality capabilities. In other words, we managed to predict the checks being made within `isSandbox()`, and ensure that the heuristic checks fail even when running in a sandboxed environment. This ensures that the behaviour of `continueMaliciousOperations()` is exposed to the sandbox probes. Moreover, we demonstrate that Jing's [5] heuristic prediction basis is a solid starting point for our work. We analysed a set of a number of real malware samples, exposing the presence of the identified heuristic data. The results demonstrated that numerous instances of the malware samples exhibit the use of the discovered heuristic, indicating the potential use of some sort of emulator or sandbox detection. We also developed a prototype implementation that takes advantage of data collected throughout the heuristic prediction process in order to automatically generate sandbox hardening capabilities. Our evaluation demonstrates that our proposed technique is able to predict and disable sandbox detecting capabilities without prior knowledge of the employed heuristic checks.

2 Background and Related work

2.1 Sandbox Detection

The use of emulation underpins sandbox construction (e.g. Ananas [6] and Mobile-sandbox [7]), since it easily provides isolation and efficient system state restoration. However, emulation provides an easy target for malware to evade sandbox analysis through sandbox detection. For example, Jing et al. [5] developed a tool (Morpheus) that is able to automatically generate heuristics that can detect Android emulators. Pestas et al. [8] outlined three categories of evasion techniques based on static properties, dynamic sensor information, and VM-related intricacies of the Android emulator that can be used for sandbox detection. Maier et al. [9] developed a tool called Sand-Finger that uses a fingerprinting approach which gathers information on several sandboxes and

uses this information to identify which particular sandbox is being used. Vidas et al. [10] categorized four classes of sandbox detection techniques based on differences in behaviour, performance, hardware components, and software components. These sandbox detecting techniques are problematic for malware analysts, since they expose potential weaknesses in their sandboxes, and that is where sandbox hardening comes into play.

2.2 Sandbox Hardening

Hardened sandboxes are ones that are not so easily bypassed. Gajrani et al. [11] took this threat into account, and set out to develop techniques that help malware analysts build a hardened sandbox analysis environment by identifying commonly used sandbox detecting techniques and patching them by applying emulator modifications, system image modifications, and by applying runtime hooks. Another sandbox hardening approach involves the use of bare-metal devices which drops the need of emulation altogether. The bare-metal approach was applied by Mutti et al. with the tool BareDroid [12]. Kirat et al. [13] mention how there is a constant tension between the quality, stealthiness, and efficiency of a malware sandbox. Trying to improve one of the factors often results in compromises in the remaining two. A bare-metal approach does bring about many negative efficiency implications in terms of cost and time. For example, in the experimentation of BareDroid, the authors mention that a full restore which is required after every analysis takes 141 seconds, thus causing a scalability issue. On the other hand, system restore time on an emulator is almost negligible. They also mentioned that the quality of the analysis is often adversely affected since probes that rely on system emulation, such as taint analysis on native code, cannot be used. Moreover, the use of bare-metal sandboxes can create new sandbox detecting heuristics when attempting to address these issues. It is evident that because of the advantages provided by system emulation, bare-metal is not a fix-all approach, and emulator based sandboxes will not be fully replaced by bare-metal ones. In our work, we will be focusing on hardening those that are still based on emulation.

Sandbox hardening can be applied to different components of a sandbox. One approach involves the modification of emulator properties [11]. Certain properties are easy to modify, however others are not modifiable out of the box and require hardware emulation tweaking. The modifications are sensitive in nature since any misconfiguration could easily corrupt the system. Besides modifying the emulator, one may also choose to directly modify the application's bytecode. By modifying the sandbox detecting parts of the code directly in the application, it is possible to force the application into failing sandbox detecting checks, even when running in a sandbox. Another hardening technique involves patching the Android Framework. By modifying the Android Framework APIs, one can control the results of the class method invocations, thus manipulating the sandbox detection verification checks. Even though this approach can be very effective, the downside is that it would require custom modifications for every Android OS version.

3 AndroNeo

AndroNeo builds upon an evasion heuristic generation technique, Morpheus [5]. While Morpheus focuses on the automatic generation of sandbox detecting heuristics, AndroNeo aims to provide automated sandbox hardening capabilities. The proposed technique involves utilising the generated heuristics, as well as properties obtained throughout the generation process, in order to produce sandbox hardening patches for the malware's Dalvik bytecode.

The notion of a distinguisher is a central component: A **Distinguisher** refers to a distinctive characteristic that can be used to classify an environment as a sandbox or an actual device. These distinguishers can fall under one of two categories: Sandbox Profile Distinguishers, or Device Profile Distinguishers. **Sandbox Profile Distinguishers** refer to a set of distinguishers that contain properties found in most sandboxes environments. In other words, if the system on which the malware is being analysed contains a property within the Sandbox Profile, then it is likely that it is in fact a sandbox. On the other hand **Device Profile Distinguishers** refer to a set of distinguishers that contain properties found in most mobile devices. In other words, if the system on which the malware is being analysed contains a property within the Device Profile, then it is likely that it is an actual device and not an emulator. As Figure 1 depicts, the distinguisher generation stage is split in two phases. The first phase starts off with a recon task that operates upon samples of real devices and sandboxes in order to generate recon datasets that contain candidate distinguishers. The second step of this phase produces the device and sandbox profiles as characterized by the computed profile distinguishers that identify them as such. The resulting distinguisher profiles are used during a second stage to automatically harden the sandbox. The distinguisher profiles themselves provide the required information to locate the patch points and to generate the code patches that deactivate evasion. The following sections present the individual steps in detail.

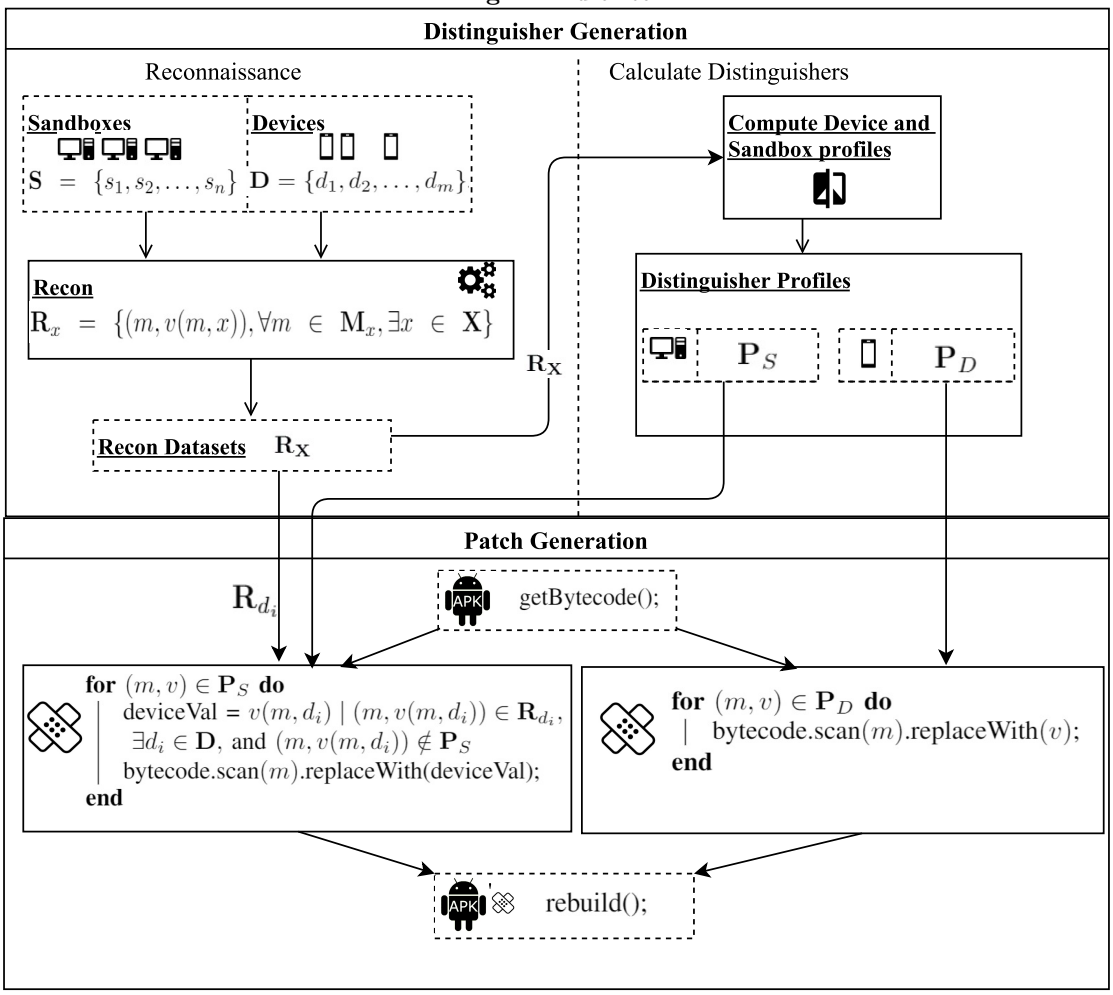
3.1 Reconnaissance

Let $\mathbf{S}$ be the set of n sample sandboxes, and $\mathbf{D}$ be the set of m sample devices, $\mathbf{S} = \{s_1, s_2, \ldots, s_n\}$ and $\mathbf{D} = \{d_1, d_2, \ldots, d_m\}$. Let $\mathbf{X}$ represent the set of all sample sandboxes and devices, i.e. $\mathbf{X} = \mathbf{S} \cup \mathbf{D}$.

The reconnaissance (or recon) phase involves the extraction of data from the sandboxes and devices, that can potentially be used to identify sandboxes. The recon works by parsing all of the Android API classes [14] in each element in $\mathbf{S}$ and $\mathbf{D}$, and invoking all of their available methods and reading of class constants. Every recon execution on a sandbox or device produces what we refer to as a recon dataset. Let $\mathbf{M}_x$ be the set of class constants and methods of all Android API classes accessible from sandbox or device x: $\mathbf{M}_x = \{m | m \in (ApiMethodCalls \cup ApiClassConstants), \exists x \in \mathbf{X}\}$, e.g: $\mathbf{M}_x = \{getDeviceId(), \ldots, getLineNumber(),\ Build.DEVICE,$
$\ldots, Build.SERIAL\}$. The function $v(m, x)$ denotes the value returned when calling/reading m within the context of $x \in \mathbf{X}$, e.g: $v(\texttt{getDeviceID()}, s_1) =$

"000000000000000". The set $\mathbf{R}_x$ represents the resulting set of key value pairs obtained by invoking all methods and constants available for the device or sandbox x i.e.: $\mathbf{R}_x = \{(m, v(m, x)), \forall m \in \mathbf{M}_x, \exists x \in \mathbf{X}\}$ e.g: $\mathbf{R}_{s_1} = \{($`getDeviceId()`, "000000000000000"$), \ldots, ($`getLineNumber()`, "15555215554"$)\}$. The set $\mathbf{R_X}$ represents the set of all recons from the set of devices and/or sandboxes in $\mathbf{X}$. i.e: $\mathbf{R_X} = \{\mathbf{R}_x | \forall x \in \mathbf{X}\}$.

Fig. 1. AndroNeo



3.2 Calculating Distinguishers

Let $r = (m, v(m, x)) \in \mathbf{R}_x$ where $m \in \mathbf{M}_x$ and $x \in \mathbf{X}$. We are interested in the ratios $\frac{|r_\mathbf{S}|}{|\mathbf{S}|}$ and $\frac{|r_\mathbf{D}|}{|\mathbf{D}|}$ where $|r_\mathbf{S}|$ and $|r_\mathbf{D}|$ represent the number of times

r is present in $\mathbf{S}$ and $\mathbf{D}$ respectively. For example, if `getDeviceId()` returns `"000000000000000"` for every sandbox in our set of sample sandboxes then we are interested in the ratio $\frac{|(getDeviceId(), \text{“000000000000000”})_{\mathbf{S}}|}{|\mathbf{S}|} = 1$. This ratio tells us that all sandboxes returned the value of `"000000000000000"`, and therefore points towards a property that could be used to identify a sandbox, i.e. a sandbox profile distinguisher.
On the other hand, if for example the value of the class constant `Build.TAGS` is `"release-keys"` for every device in our set of sample devices then we are interested in the ratio $\frac{|(Build.TAGS, \text{“}release-keys\text{”})_{\mathbf{D}}|}{|\mathbf{D}|} = 1$. This ratio tells us that all devices returned the value of `"release-keys"`, and therefore points towards a property that could be used to identify a device, i.e. a device profile distinguisher.

Sandbox Profile The set $\mathbf{P}_S$ represents all sandbox profile distinguishers: $\mathbf{P}_S = \{(m, v) \mid (m, v) = (m, v(m, s_i)) \in \mathbf{R}_{s_i}, \frac{|r_{\mathbf{S}}|}{|\mathbf{S}|} > \tau > \frac{|r_{\mathbf{D}}|}{|\mathbf{D}|}, \exists s_i \in \mathbf{S}\}$. The elements in $\mathbf{P}_S$ provide us with a list of properties that can be used to identify a sandbox. Let us assume that we have a device or sandbox x_0, and we want to determine whether or not x_0 is a sandbox. For every $(m, v(m, x_0)) \in \mathbf{R}_{x_0}$ if $(m, v(m, x_0)) \in \mathbf{P}_S$ then this indicates that x_0 is a sandbox.

Device Profile The set $\mathbf{P}_D$ represents all device profile distinguishers: $\mathbf{P}_D = \{(m, v) \mid (m, v) = (m, v(m, d_j)) \in \mathbf{R}_{d_j}, \frac{|r_{\mathbf{D}}|}{|\mathbf{D}|} > \tau > \frac{|r_{\mathbf{S}}|}{|\mathbf{S}|}, \exists d_j \in \mathbf{D}\}$. The elements in $\mathbf{P}_D$ provide us with a list of properties that can be used to identify a sandbox, or rather the lack of a device. Let us assume that we have a device or sandbox x_0, and we want to determine whether or not x_0 is a sandbox. Then for every $(m, v(m, x_0)) \in \mathbf{R}_{x_0}$ if $(m, v(m, x_0)) \notin \mathbf{P}_D$ then this indicates that x_0 is not a device, and therefore a sandbox. The Sandbox and Device profiles correspond to what Morpheus [5] refers to as S-pool and D-pool respectively. Additionally, our profiles make use of a tunable threshold (τ), and retain the obtained Device Dataset values for the patch generation phase.

3.3 Patch Generation

Let s_0 be the sandbox that requires hardening. Since s_0 is in fact a sandbox, then there is a good chance that for every $(m, v) \in \mathbf{P}_S$, it is also the case that $(m, v(m, s_0)) \in \mathbf{P}_S$. These occurrences can provide malware with sandbox detecting capabilities. In order to hide the presence of these values, we will re-use data that was collected during the recon phase. For every $(m, v) \in \mathbf{P}_S$, we need to identify a corresponding $(m, v(m, d_i)) \in \mathbf{R}_{d_i}$, where $d_i \in \mathbf{D}$ and $(m, v(m, d_i)) \notin \mathbf{P}_S$. The bytecode can then be modified to ensure that for every $(m, v) \in \mathbf{P}_S$, $v(m, s_0)$ returns $v(m, d_i)$, thus emulating the value of a real device.

In the case of device profile distinguishers since s_0 is a sandbox, then there is a good chance that for every $(m, v) \in \mathbf{P}_D$, $(m, v(m, s_0)) \notin \mathbf{P}_D$. This time, the bytecode needs to be modified to in such a way that for every $(m, v) \in \mathbf{P}_D$, $v(m, s_0)$ returns v, thus emulating the value of a real device (since this time v

indicates a value that is commonly found on devices). The following process is used to patch the malware's bytecode:

Input: sandbox detecting malware apk
Output: patched malware apk′
bytecode = apk.getBytecode();
for $(m, v) \in \mathbf{P}_S$ **do**
  deviceVal = $v(m, d_i) \mid (m, v(m, d_i)) \in \mathbf{R}_{d_i}$,
   $\exists d_i \in \mathbf{D}$, and $(m, v(m, d_i)) \notin \mathbf{P}_S$
  bytecode.scan(m).replaceWith(deviceVal);
end
for $(m, v) \in \mathbf{P}_D$ **do**
  bytecode.scan(m).replaceWith(v);
end
return apk.rebuild() -> apk′;

4 Experimentation

A number of experiments were carried out in order to evaluate the capabilities of automatically generated sandbox detection heuristics, whilst demonstrating the effectiveness of the hardening process just presented.

4.1 Experiment Setup

We developed an Android application that allowed us to collect data by invoking all possible Android API class methods and constant values through the use of the Reflection API. Apktool was used to decode and re-compile the applications, whilst a number of bash scripts allowed us to patch the smali code generated through the apktool, according to the identified distinguishers. The environment in which the experimentation was conducted, consisted of a number of Android sandboxes and emulators (Sanddroid [15], NVISO ApkScan [16], Droidbox [17], Android 7.0 Emulator, Android 6.0 Emulator), and a set of Android devices (Samsung Galaxy S4, Nexus 5x, Nexus 5, Nexus 6P, OnePlus X). Moreover, we used a set of 7160 real malware samples from VirusShare [18]. In our case studies, we made use of the popular DroidBox sandbox, which is also the underlying dynamic analysis tool of several Android sandboxes [3].

4.2 Distinguisher Profiles

In order to validate the relevance of the identified profile distinguishers and test their potential in identifying evasion checks, we cross-checked our findings against the malware samples. Table 1 enlists the results of the top 10 sandbox profile distinguishers. The results were calculated by identifying the number of

Table 1. Sandbox Profile Distinguishers in malware samples

Method/Field	Count	Value	Count
`Build.MODEL`	2478	`sdk`	2352
`getDeviceId`	2395	`000000000000000`	794
`getNetworkOperatorName`	1522	`Android`	665
`Build.DEVICE`	1150	`generic`	594
`Build.BOARD`	992	`unknown`	561
`Build.MANUFACTURER`	1404	`unknown`	255
`Build.CPU_ABI`	306	`x86`	95
`getSubscriberId`	1830	`310260000000000`	53
`getSimOperatorName`	432	`Android`	23
`Build.TAGS`	378	`test-keys`	17

malware samples that contained instances of $(m, v) \in \mathbf{P}_S$, where both m and v are found in the same class. For example, the results show that out of the 7160 malware samples, we found 2395 applications that invoked the method `getDeviceId()`. The invocation of this method on its own is not necessarily suspicious, however when we see that 794 of these applications also looked for the string `"000000000000000"` within the same class, then this fact increases the likelihood that this call is made for emulator detection purposes. The numbers in Table 1, are somewhat conservative since they only represent cases where both instances of the invocation (e.g: `getDeviceId()`) and the corresponding distinguisher value (e.g: `"000000000000000"`) were found to be in the same class. There may very well be a few additional cases where the distinguisher values are defined in a class different to the distinguisher method/field invocation. Moreover, there is also the possibility that the distinguisher values are encrypted or hashed, and are therefore not identified during the crosscheck against the malware samples. Nevertheless, when it comes to the actual patching, these cases will still be patched using the bytecode modification approach, since it is the field or method call (e.g: `getDeviceId()`) that is being modified. These results show that it is very likely that the automatically generated heuristics are being used to detect sandboxes or emulators by malware families in the wild, providing validity to Jing's [5] assumptions.

4.3 Case Studies

In order to verify that the detected profile distinguishers contain actual evasion checks, and that the proposed patch generation step effectively deactivates them, we chose two representative samples and conducted a more in-depth investigation. We chose the samples on the basis that they form part of two widespread malware families, Crosate and Pincer, as well as the availability of thorough documentation [19] of their behaviour. This information provides the ground truth with which to compare the results obtained by AndroNeo. Through Droidbox, we proceeded with analyzing both the original and patched samples, with the resulting behaviour observed in both instances being compared to the ground

truth. We generated a bytecode hardening class, containing the data necessary for AndroNeo to spoof the return values of the identified distinguishers (i.e. `bytecode.scan`(m)`.replaceWith`(v) step in Section III C). The code below is a snippet from the hardening class.

```
# Field Declarations
const-string v0, "release-keys"
sput-object v0, Lharden/Harden;->FIELD18:Ljava/lang/String;
....
# Method Declarations
.method public static method14()Ljava/lang/String;
....
   const-string v0, "353627074120224"
   return-object v0
.end method
```

Crosate Crosate is a bot with the ability to steal SMSs, call logs, contact information, send SMS, record a call, and makes a phone call. However, when executing in DroidBox, it terminates itself, thus hiding all bot to Command and Control (CNC) communication. The listing below contains Crosate's code where the sandbox detection check is made:

```
public void onCreate() {
....
    String BotID = tm.getDeviceId();
....
if (BotID.indexOf("000000000000000") != -1) {
    System.exit(0);
  }
```

In the last three lines the application checks the value of `BotID`, which returns the value of `getDeviceId()` (i.e. the phone's IMEI). If it finds that the IMEI contains `"000000000000000"` then it calls `System.exit(0)`. The listing below contains the code of the patched version of Crosate:

```
public void onCreate() {
....
  String BotID =  AGHardening.method14();
....
  if (BotID.indexOf("000000000000000") != -1) {
    System.exit(0);
  }
```

Here one can see that again the code checks the value of `BotID` and compares it with `"000000000000000"`. However, this time `BotID` is not returning `getDeviceId()`, but is instead invoking `method14()` from our hardening class. As we saw earlier, `method14()` now returns `"353627074120224"` instead of the sandbox's IMEI. Therefore, in our patched version the check will fail and the `System.exit(0)` method will not be invoked, thus performing all of the malicious operations as it would on a actual device. The application also calls several other

methods such as `getLineNumber1()` and `getNetworkOperatorName()`, which can also be used to identify a sandbox. All of these instances were replaced with their corresponding device values found in the hardening class. Comparing the analysis report of the original Crosate sample and the hardened version exposed the difference in their behaviour. The original version showed very little activity, and the entire report only produced 21 log entries. On the other hand, the modified version produced 191 log entries, which clearly showed additional activities and services being started, as well as device data being exfiltrated. As confirmed by a review of the malware's code, the results demonstrated that the additional 170 log entries were generated by the services and activities that were only launched if the sandbox detecting code failed, and thus the `System.exit(0)` was not invoked. This also corresponded to the expected behaviour as documented by Nigam et. al. [19]. This demonstrates the successful automatic patching of the malware's sandbox detecting capabilities, exposing the malware's split personalities.

Pincer Pincer is another bot that only communicates with the CNC server when it detects that the device is an actual device and not a sandbox. Therefore, our assumption was that we would only see network communication with the CNC once we patched the malware sample. The listing below outlines the several checks that Pincer invokes in order to distinguish between a sandbox and a device. Fortunately, our tool was able to predict these checks and patch them accordingly.

```
if ( C0024b.m108d(context).toLowerCase().equals("android")
  || C0024b.m106b(context).equals("000000000000000")
  || C0024b.m107c(context).equals("15555215554")
  || AGHardening.FIELD21.toLowerCase().equals("sdk")
  || AGHardening.FIELD21.toLowerCase().equals("generic")) {
    C0018a.m68a(context, true);
} else {C0014a.m50a(context, jSONObject, new C0023d());}
```

The listing below shows the methods from where the distinguishers are invoked, and how they were identified and modified.

```
public static String m106b(Context context) {
  return AGHardening.method14();
}
public static String m107c(Context context) {
  return AGHardening.method6();
}
public static String m108d(Context context) {
  return AGHardening.method8();
}
```

Therefore, all of the methods and constants involved in the sandbox detecting checks were replaced with our methods and fields in the hardening class, forcing them to return actual device values instead of the sandbox's values. Pincer was an interesting case study since initially both the patched and original versions

seemed to return the same logs. The patching itself did not fail. However, the parts of the code where the split personality occur are only executed once a command is received from the CNC. Only once we simulated the receipt of a CNC command could we observe the differences in behaviour. This issue falls outside the scope of AndroNeo, however it is included in the scope extension. Comparing the analysis report of the original Pincer sample and the hardened version exposed the difference in behaviour. The original version did not show any network communication corresponding to our simulated CNC commands. The modified version, on the other hand, reported the network connections and the IP addresses of the CNC, and responded to the commands that we requested. Moreover, the modified version also attempted to access image files on the SD card, a behaviour that was not present in the original version. Once again this case study clearly demonstrates the successful patching of the malware's sandbox detecting capabilities, exposing the malware's split personalities. The patched malware's behaviour corresponded to the expected behaviour as documented by Nigam et. al. [19].

It is important to note that in our tests the detection and patching of distinguishers was done in an automated manner. The identified distinguishers retrieved from our sample sandboxes and devices were able to predict and patch the sandbox detection techniques used by these malware samples without being explicitly told what to look for.

5 Scope Extension

5.1 Limitations

Even though the evaluation produced promising results, and the prototype successfully patched well known malware samples in an automated manner, one must bear in mind that heavily obfuscated malware within scope is not currently handled. There still exist evasion techniques that may not be identified and patched in the current implementation. For example, malware obfuscation through the use of native code, direct Binder IPC invocations, or malware packers will not be handled by the prototype. Another limitation is that currently our recon implementation only utilises Java reflection in order to invoke the methods and class fields and collect system information. Morpheus's artefact collector on the other hand implements additional techniques, such as the use of a directory walker that identifies the presence of emulator specific files and folders, in order to generate heuristics based on these artefacts.

5.2 Proposed Extensions

AndroNeo is planned to be extended subject to further experimentation in the following ways:

Improve Bytecode Patching In its current state AndroNeo is not able to tackle self-modifying malware that implement techniques such as runtime class loading to hide its malicious code. To overcome this, AndroNeo would need to implement a form of dynamic patching.

Implement all of Morpheus' Artefact retrievers Extending AndroNeo to include other such sources of system information, such as the ones implemented by Morpheus, could generate additional heuristics.

Extend to all Sandbox Detecting techniques The current scope is limited to hardening sandbox detecting techniques based on static emulator properties. However, the proposed methodology can be evolved to tackle event-based, or user presence based techniques. Whether or not heuristic prediction can tackle these types of evasion techniques requires further investigation. Alternatively, an interesting avenue could involve merging our techniques with other techniques, such as the one proposed by Pooryousef et. al. [20], that are aimed at tackling the exposure of event-driven actions. Moreover, one could extend AndroNeo to build upon alternative heuristic generation techniques [21] in order to patch their corresponding generated heuristics.

6 Conclusion

Android malware families demonstrate the ability of detecting malware analysis sandboxes using detection heuristics. To tackle this problem we presented AndroNeo, a tool that automatically hardens malware analysis sandboxes to disable the malware's sandbox detection capabilities. AndroNeo identifies sandbox detecting capabilities within an Android application and alters and disables its functionality. Moreover, we presented a prototype implementation of AndroNeo demonstrating its capabilities on real malware families. AndroNeo would benefit from straightforward extensions in terms of complete reuse of Morpheus, and other extensions requiring further thought, such as the ability to patch event-based, or user presence based sandbox detecting techniques.

References

1. Techcrunch. android overtakes windows as the internets most used operating system. https://techcrunch.com/2017/04/03/statcounter-android-windows. Accessed: 2017-05-01.
2. Mcafee. 2016 mobile threat report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf. Accessed: 2017-02-05.
3. Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar R. Weippl. Enter sandbox: Android sandbox comparison. *CoRR*, abs/1410.7749, 2014.

4. Dominik Maier, Mykola Protsenko, and Tilo Muller. A game of droid and mouse: The threat of split-personality malware on android. *Computers & Security*, 54:2–15, 2015.
5. Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.
6. Thomas Eder, Michael Rodler, Dieter Vymazal, and Melanie Zeilinger. Ananas-a framework for analyzing android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 711–719. IEEE, 2013.
7. Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.
8. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.
9. David Maier, Tim Muller, and Mykola Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 30–39. IEEE, 2014.
10. 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*, pages 447–458. ACM, 2014.
11. Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, Manoj Singh Gaur, and Mauro Conti. A robust dynamic analysis system preventing sandbox detection by android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 290–295. ACM, 2015.
12. Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015.
13. Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.
14. Anrdroid API classes. https://developer.android.com/reference/classes.html. Accessed: 2017-05-01.
15. Sandroid: Android malware sandbox. http://sanddroid.xjtu.edu.cn. Accessed: 2017-03-01.
16. Nviso apkscan, scan android applications for malware. https://apkscan.nviso.be/. Accessed: 2017-05-01.
17. Droidbox: Dynamic analysis of android applications. https://github.com/pjlantz/droidbox. Accessed: 2017-05-01.
18. Virusshare.com - because sharing is caring. https://virusshare.com/. Accessed: 2017-05-01.
19. Ruchna Nigam. A timeline of mobile botnets. *Virus Bulletin, March*, 2015.
20. Shahrooz Pooryousef and Morteza Amini. Enhancing accuracy of android malware detection using intent instrumentation. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 380–388. INSTICC, ScitePress, 2017.

21. Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.