Detecting Environment-Sensitive Malware

Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti

Secure Systems Lab, Vienna University of Technology
{mlindorfer,ck,pmilani}@seclab.tuwien.ac.at

Abstract. The execution of malware in an instrumented sandbox is a widespread approach for the analysis of malicious code, largely because it sidesteps the difficulties involved in the static analysis of obfuscated code. As malware analysis sandboxes increase in popularity, they are faced with the problem of malicious code detecting the instrumented environment to evade analysis. In the absence of an "undetectable", fully transparent analysis sandbox, defense against sandbox evasion is mostly reactive: Sandbox developers and operators tweak their systems to thwart individual evasion techniques as they become aware of them, leading to a never-ending arms race.
The goal of this work is to automate one step of this fight: Screening malware samples for evasive behavior. Thus, we propose novel techniques for detecting malware samples that exhibit semantically different behavior across different analysis sandboxes. These techniques are compatible with any monitoring technology that can be used for dynamic analysis, and are completely agnostic to the way that malware achieves evasion. We implement the proposed techniques in a tool called DISARM, and demonstrate that it can accurately detect evasive malware, leading to the discovery of previously unknown evasion techniques.

Keywords: Malware, Dynamic Analysis, Sandbox Detection, Behavior Comparison

1 Introduction

Dynamic analysis of malicious code has increasingly become an essential component of defense against Internet threats. By executing malware samples in a controlled environment, security practitioners and researchers are able to observe its malicious behavior, obtain its unpacked code [17, 21], detect botnet command and control (C&C) servers [30] and generate signatures for C&C traffic [27] as well as remediation procedures for malware infections [24]. Large-scale dynamic malware analysis systems (DMAS) based on tools such as Anubis [6] and CWSandbox [35] are operated by security researchers[1] and companies[2,3]. These services are freely available to the public and are widely used by security

[1] Anubis: Analyzing Unknown Binaries (http://anubis.iseclab.org/)
[2] SunbeltLabs (http://www.sunbeltsecurity.com/sandbox/)
[3] ThreatExpert (http://www.threatexpert.com/)

practitioners around the world. In addition to these public-facing services, private malware analysis sandboxes are operated by a variety of security companies such as Anti-Virus vendors. Like most successful security technologies, malware analysis sandboxes have therefore attracted some attention from miscreants.

One way for malware to defeat dynamic analysis is to detect that it is running in an analysis sandbox rather than on a real user's system and refuse to perform its malicious function. For instance, code packers that include detection of virtual machines, such as Themida, will produce executables that exit immediately when run inside a virtual machine such as VMWare [20]. There are many characteristics of a sandbox environment that may be used to fingerprint it. In addition to using "red pills" that aim to detect widely deployed emulation or virtualization technology [29, 28, 25, 10, 11], malware authors can detect specific sandboxes by taking advantage of identifiers such as volume serial numbers or IP addresses. As we will discuss in Section 2, sandbox detection is not a theoretical problem; Table 1 holds a number of concrete examples of how malware samples have evaded analysis in our Anubis sandbox in the past.

One approach to defeating sandbox evasion is to try to build a *transparent* sandbox. That is, to construct an analysis environment that is indistinguishable from a real, commonly used production environment. This is the goal of systems such as Ether [9]. However, Garfinkel et al. [12] argue that it is fundamentally unfeasible to build a fully transparent virtual machine monitor, particularly if code running in the sandbox has access to the Internet and can therefore query a remote time source. In fact, Ether does not defend against timing attacks that use a remote time source, while Pek et al. [26] have introduced a tool called nEther that is able to detect Ether using local attacks. Even if transparent sandbox technology were available, a specific sandbox installation could be detectable based on the particular configuration of software that happens to be installed on the system, or based on identifiers such as the product IDs of installed software [4] or the universal identifiers of disk partitions.

Another approach relies on running a sample in multiple analysis sandboxes to detect deviations in behavior that may indicate evasion [8, 18, 2, 15]. This is the approach we use in this paper. For this, we run a malware sample in several sandboxes, obtaining a number of behavioral profiles that describe its behavior in each environment. We introduce novel techniques for normalizing and comparing behavioral profiles obtained in different sandboxes. This allows us to discard spurious differences in behavior and identify "environment-sensitive" samples that exhibit semantically different behavior. We implement the proposed techniques in a system called Disarm: DetectIng Sandbox-AwaRe Malware.

Disarm detects differences in behavior regardless of their cause, and is therefore completely agnostic to the way that malware may perform sandbox detection. Furthermore, it is also largely agnostic to the monitoring technologies used in the analysis sandboxes, since it does not require heavyweight, instruction-level instrumentation. Any monitoring technology that can detect persistent changes to system state at the operating system level can take advantage of our techniques.

Previous work on detecting and remediating analysis evasion has required fine-grained, instruction-level instrumentation [18, 15]. In our experience operating Anubis, a DMAS that processes tens of thousands of samples each day, we have found that large-scale deployment of instruction-level instrumentation is problematic. This is because it leads to an extremely slow emulated environment, to the point that some malware fail to perform network communication because of server-side timeouts. Furthermore, the produced log files are unmanageably large (up to half a Gigabyte for a single execution according to Kang et al. [18]). DISARM does not suffer from this limitation. This allows us to apply our techniques to a significant number of malware samples, revealing a variety of anti-analysis techniques.

Chen et al. [8] also performed a large-scale study of analysis-resistant malware. However, their work assumes that an executable is evading analysis whenever its executions differ by even a single persistent change. This assumption does not seem to hold on a dataset of modern malware: as we will show, about one in four malware samples we tested produced different persistent changes between multiple executions *in the same sandbox*. DISARM executes malware samples multiple times in each sandbox to establish a baseline for a sample's variation in behavior. Furthermore, we introduce behavior normalization and comparison techniques that allow us to eliminate spurious differences that do not correspond to semantically different behavior.

DISARM does not, however, automatically identify the root cause of a divergence in behavior. Samples we detect could therefore be further processed using previously proposed approaches to automatically determine how they evade analysis. For instance, the techniques proposed by Balzarotti et al. [2] can be used to automatically diagnose evasion techniques that are based on CPU emulation bugs. Differential slicing [15] is a more general technique that can likewise identify the root cause of a divergence, but it requires a human analyst to select a specific difference in behavior to be used as a starting point for analysis.

We evaluate DISARM using four sandboxes with two different monitoring technologies: In-the-box monitoring using a Windows device driver, and out-of-the-box monitoring using Anubis. We tested the system on a dataset of over 1,500 samples, and identified over 400 samples that exhibit semantically different behavior in at least one of the sandboxes considered. Further investigation of these samples allowed us to identify a number of previously unknown techniques for evading our two monitoring technologies. Most of these evasion techniques can be trivially defeated with small changes to our analysis sandboxes. Furthermore, DISARM helped us to discover several issues with the configuration of software installed inside our sandboxes that, while unrelated to evasion, nonetheless prevent us from observing some malicious behavior.

To summarize, our contributions are the following:

- We introduce a system called DISARM for detecting environment-sensitive malware by comparing its behavior in multiple analysis sandboxes. DISARM is entirely agnostic to the root cause of the divergence in behavior, as well as to the specific monitoring technologies employed.

- We develop a number of novel techniques for normalizing and comparing behavior observed in different sandboxes, discarding spurious differences that do not correspond to semantically different behavior.
- We tested DISARM by running over 1,500 malware samples in four different analysis sandboxes based on two monitoring technologies, and show that it can accurately detect environment-sensitive malware.
- As a result of these experiments, we discovered a number of previously unknown analysis evasion techniques. Concretely, these findings will allow us to improve the analysis capabilities of the widely used Anubis service.

2 Motivation and Approach

To make the case for DISARM, we will provide a brief history of analysis evasion against Anubis. Anubis is a dynamic malware analysis system (DMAS) that is based on an instrumented Qemu [7] emulator. The main output of Anubis analysis is a human-readable report that describes the operating system level behavior of the analyzed executable. Our lab has been offering malware analysis with Anubis as a free service since February 2007. This service has over 2,000 registered users, has received submissions from 200,000 distinct IP addresses, and has already analyzed over 10,000,000 malware samples.

Public-facing analysis sandboxes such as Anubis are particularly vulnerable to detection, because attackers can probe the sandbox by submitting malware samples specifically designed to perform reconnaissance. Such samples can read out characteristics of the analysis sandbox and then use the analysis report produced by the sandbox to leak the results to the attacker. These characteristics can later be tested by malware that wishes to evade analysis. However, because of sharing of malware samples between sandbox operators, private sandboxes may also be vulnerable to reconnaissance [36], so long as they allow executed samples to contact the Internet and leak out the detected characteristics.

The first instance of Anubis evasion that we came across in the wild was a packer called `OSC Binder` that was released in September 2007 and advertised "anti-Anubis" features. Since then, we have become aware of a number of techniques used by malware to thwart Anubis analysis.

Chen et al. [8] have proposed a taxonomy of approaches that can be used by malware for the detection of analysis sandboxes. These are not limited to techniques that aim to detect virtualized [29] or emulated [28, 25] environments, but also include application-level detection of characteristic features of a sandbox, such as the presence of specific processes or executables in the system.

Table 1 shows a number of Anubis evasion techniques that we have become aware of over the years, classified according to an extended version of this taxonomy. Specifically, we added one abstraction (`Network`) and two classes of artifacts (`Connectivity` and `Unique identifier`) to the taxonomy. The unique identifier class is required because many of the detection techniques that have been used against Anubis are not targeted at detecting the monitoring technology used by Anubis, but a specific instance of that technology: The publicly accessible Anubis service. The connectivity class is required because the network

Table 1. Anubis evasion techniques according to taxonomy [8] (extended).

Abstraction	**Artifact**	**Test**
Hardware	unique id	disk serial number [4]
Environment	execution	MOD R/M emulation bug [25]
		AAM instruction emulation bug
Application	installation	C:\exec\exec.exe present
		username is "USER" [4]
		executable name is "sample.exe" [4]
	execution	popupkiller.exe process running
	unique id	windows product ID [4]
		computer name [4]
		volume serial number of system drive
		hardware GUID
Network	connectivity	get current time from Yahoo home page
		check Google SMTP server response string
	unique id	server-side IP address check [36, 19, 16]

configuration of a DMAS faces a trade-off between transparency and risk. It is typically necessary to allow malware samples some amount of network access to be able to observe interesting behavior. On the other hand, we need to prevent the samples from causing harm to the rest of the Internet. A malware sample, however, may detect that it is being provided only limited access to the Internet, and refuse to function. For instance, a DMAS needs to stop malware from sending SPAM. Rather than blocking the SMTP port altogether, it can redirect SMTP traffic to its own mail server. Some variants of the Cutwail SPAM engine detect this behavior by connecting to Gmail's SMTP servers and verifying that the server replies with a specific greeting message.

In the past we have mostly become aware of analysis evasion techniques "by accident". Some samples that evade Anubis have been brought to our attention by Anubis users, while a few Anubis evasion techniques have been discussed in hacker forums and security blogs. In a few instances the Anubis developers have made more deliberate efforts to identify evasion techniques. In one case, a collection of code packers were tested to determine whether and how they evaded Anubis. In another instance, we obtained a number of "red pills" generated by a fuzzer for CPU emulators [25], and fixed the bugs they identified.

In the arms race between malware analysis systems and malware samples that evade analysis, we need to be able to rely on more automation. For this, we require scalable tools to screen large numbers of malware samples for evasive behavior, regardless of the class of evasion techniques they employ. This is the role that DISARM aims to fill.

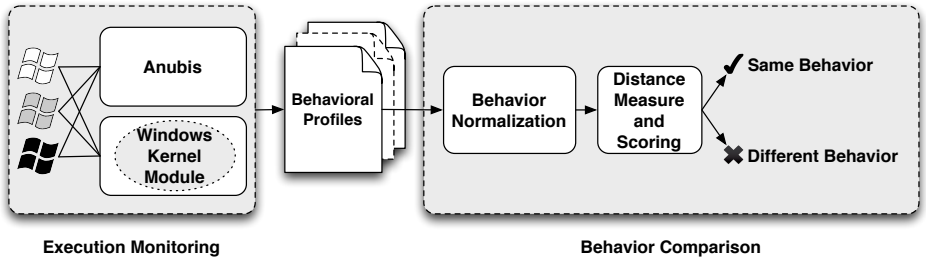

Fig. 1. System Architecture of DISARM.

2.1 System Architecture

DISARM works in two phases, illustrated in Fig. 1. In the execution monitoring phase, a malware sample is executed in a number of analysis sandboxes. For the purpose of this paper, we define a sandbox as a combination of a monitoring technology with a system image: That is, a specific configuration of an operating system on a virtual disk. We execute a sample multiple times in each sandbox. The output of this execution monitoring provides us with the malware's behavior represented as a number of behavioral profiles (one for each execution). In the behavior comparison phase, we normalize the behavioral profiles to eliminate spurious differences. We then compute the distances between each pair of normalized behavioral profiles. Finally, we combine these distances into an evasion score, that is compared against a threshold to determine whether the malware displayed different behavior in any of the sandboxes. Samples that are classified as showing signs of evasion can then be further analyzed in order to identify new evasion techniques and make our sandboxes resilient against these attacks.

3 Execution Monitoring

We analyze malware behavior using two different monitoring technologies. The first is Anubis [6], which is an "out-of-the-box" monitoring technology that captures an executable's behavior from outside the Windows environment using an instrumented full system emulator. The second system uses "in-the-box" monitoring based on system call interception from inside the Windows environment. The idea is that by using two completely different monitoring technologies we are able to reveal sandbox evasion that targets a specific instrumentation technique. Furthermore, we employ sandboxes that use different Windows installations in order to detect evasion techniques that rely on application and configuration characteristics to identify analysis systems.

3.1 In-the-Box Monitoring

The Anubis system has been extensively described in previous work [6, 3]. For in-the-box monitoring, on the other hand, we use a custom-built system that

provides lightweight monitoring of a malware's behavior at the system call level. To this end, we implemented a Microsoft Windows kernel module that intercepts system calls by hooking the entries of the System Service Dispatch Table (SSDT) [13]. This driver records the system call number, a timestamp and selected input parameters, before forwarding the call to the actual system call. After execution, the driver further records the output parameters and the return value. To log only relevant data, the driver maintains a list of processes related to the analyzed malware, and only logs calls originating from these processes. Events such as process creation, service creation, injection of threads into foreign processes, foreign memory writes, and mapping of memory into a foreign process, trigger the inclusion of new processes into the analysis. To maintain the integrity of our system, we prohibit the loading of any other drivers by not forwarding calls to `NtLoadDriver` and `NtSetSystemInformation`.

3.2 Behavior Representation

The analysis of samples with either monitoring technology leads to the creation of a number of analysis artifacts such as a human-readable report summarizing the observed behavior, a detailed log of system calls, a network traffic trace of all network communication performed by the malware, the malware's standard output and error as well as the content of any files generated during analysis. For the purpose of this work we chose to represent malware's system and network-level behavior as a behavioral profile [3, 5]. A behavioral profile is extracted from system call and network traces and represents behavior as a set of features. Each feature represents an action on an operating system (OS) resource, and is identified by the type and name of the resource, the type of action and a boolean flag representing the success or failure of the action. For example, a feature could represent the successful creation of a file called `C:\Windows\xyz.exe`. For network-related features the resource name is a tuple $< IP, domain\ name >$, representing the network endpoint that the malware sample is communicating with. We consider two network resources to be the same if *either one* of the IP or the domain name used to resolve the IP are the same. The reason is that fast-flux service networks [32] or DNS-based load balancing may lead malware to contact different IPs in different executions. Finally, each feature is tagged with a timestamp, representing the offset into the analysis run when the feature was first observed [5]. As we will see, this is essential to be able to compare behavior across monitoring technologies with vastly different performance overheads. The behavioral profiles used in [3] also include features that represent data-flow between OS resources. To maintain compatibility with lightweight monitoring technologies that cannot track the flow of data within the monitored programs, we do not consider such features in this work.

4 Behavior Comparison

When comparing behavioral profiles produced by different monitoring technologies, it is highly unlikely that they will contain the same amount of features. The

reason is that each monitoring technology is likely to have significantly different runtime overheads, so a sample will not be able to execute the same number of actions on each system within a given amount of time. Nor can we simply increase the timeout on the slower system to compensate for this, since monitoring overheads may vary depending on the type of load. Thus, given two sandboxes α and β and the behavioral profiles consisting of n_α and n_β features respectively, DISARM only takes into account the first $min(n_\alpha, n_\beta)$ features from each profile, ordered by timestamp. In a few cases, however, this approach is not suitable. If the sample terminated on both sandboxes, or it terminated in sandbox α and $n_\alpha < n_\beta$, we have to compare all features. This is necessary to identify samples that detect the analysis sandbox and immediately exit. Samples that detect a sandbox may instead choose to wait for the analysis timeout without performing any actions. We therefore also compare all features in cases where the sample exhibited "not much activity" in one of the sandboxes. For this, we use a threshold of 150 features, that covers the typical amount of activity performed during program startup. This is the threshold used by Bayer et al. [4], who in contrast observed 1,465 features in the average profile.

Not all features are of equal value for characterizing a malware's behavior. DISARM only takes into account features that correspond to persistent changes to the system state as well as features representing network activity. This includes writing to the file system, registry or network as well as starting and stopping processes and services. This is similar to the approach used in previous work [1, 8] and, as we will show in Section 5.1, it leads to a more accurate detection of semantically different behavior.

4.1 Behavior Normalization

In order to meaningfully compare behavioral profiles from different executions of a malware sample, we need to perform a number of normalization steps, mainly for the following two reasons: The first reason is that significant differences in behavior occur even when running an executable multiple times within the same sandbox. Many analysis runs exhibit non-determinism not only in malware behavior but also in behavior occurring inside Windows API functions, executables or services. The second reason is that we compare behavioral profiles obtained from different Windows installations. This is necessary to be able to identify samples that evade analysis by detecting a specific installation. Differences in the file system and registry, however, can result in numerous differences in the profiles. These spurious differences make it harder to detect semantically different behavior. Therefore, we perform the following normalizations on each profile.

Noise Reduction. In our experience even benign programs cause considerable differences when comparing profiles from different sandboxes. As a consequence, we captured the features generated by starting four benign Windows programs (notepad.exe, calc.exe, winmine.exe, mspaint.exe) on each sandbox, and consider them as "noise". These features are filtered out of all behavioral profiles. Similarly, we filter out the startup behavior of explorer.exe, iexplore.exe, cmd.exe, and

Dr. Watson. This normalization eliminates a number of differences that are not directly related to malware behavior.

User Generalization. Programs can write to the user's home directory in `C:\Documents and Settings\<username>` without needing special privileges. Malware samples therefore often write files to this directory. In the registry, user specific data is stored in the key `HKEY_CURRENT_USERS`, which actually points to `HKEY_USERS\<SID>`. The SID is a secure identifier created by the Windows setup program. It is unique for every user and system. Profiles from different systems certainly differ in the users SID and may also contain different usernames. We therefore generalize these values.

Environment Generalization. Other system specific values include hardware identifiers and cache paths. Furthermore, names of folders commonly accessed by malware, e.g. `C:\Documents and Settings` and `C:\Program Files` and their respective subfolders, depend on the language of the Windows installation. We generalize these identifiers and paths to eliminate differences not caused by malware behavior when comparing profiles from different Windows installations.

Randomization Detection. Malware samples often use random names when creating new files or registry keys. Since DISARM executes each sample multiple times in each sandbox, we can detect this behavior by comparing profiles obtained in the same sandbox. Like the authors of MIST [33], we assume that the path and extension of a file are more stable than the filename. As a consequence, we detect all created resources (in the filesystem or registry) that are equal in path and extension but differ in name. If the same set of actions is performed on these resources in all executions, we assume that the resource names are random. We can thus generalize the profiles by replacing the random names with a special token.

Repetition Detection. Some types of malware perform the same actions on different resources over and over again. For instance, file infectors perform a scan of the filesystem to find executables to infect. This behavior leads to a high number of features, but in reality only represents one malicious behavior. Furthermore, these features are highly dependent on a sandbox's file system and registry structure. To generalize these features, we take into account actions that request directory listings or enumerate registry keys. We also consider the arguments that are passed to the enumeration action, for example queries for files with extension ".exe". For each such query, we examine all actions on resources that match the query. If we find any actions (such as a file write) that are performed on three or more such resources, we create a generalized resource in the queried directory and assign these actions to it.

Filesystem and Registry Generalization. For each sandbox, we create a snapshot of the Windows image's state at analysis start. This snapshot includes a list of all files, a dump of the registry, and information about the environment. We use this information to generalize the user and the environment. We can also use this information to view a profile obtained from running on one image in the context of another image. This allows us to remove actions that would be impossible or unnecessary in the other image. That is, we ignore the creation of a

resource that already exists in the other image and, conversely, the modification or deletion of a resource that doesn't exist in the other image.

4.2 Distance Measure and Scoring

The actions in our behavioral profiles are represented as a set of string features. We thus compare two behavioral profiles using the Jaccard distance [14]:

$$J(a,b) = 1 - |a \cap b| / |a \cup b|. \quad (1)$$

Balzarotti et al. [2] observed that two executions of the same malware program can lead to different execution runs. Our own experiments reveal that about 25 % of samples execute at least one different persistent action between multiple executions in the same sandbox. Because of this, we cannot simply consider a high distance score as an indication of evasion. Instead, we consider the deviations in behavior observed within a sandbox as a baseline for variations observed when comparing behavior across different sandboxes. We therefore calculate an evasion score defined as:

$$E = \max_{1<i<n} \left\{ \max_{1<j<n, i \neq j} \left\{ \text{distance}(i,j) - \max\{\text{diameter}(i), \text{diameter}(j)\} \right\} \right\}. \quad (2)$$

Here, diameter(i) is the full linkage (maximum) distance between executions in sandbox i, while distance(i, j) is the full linkage (maximum) distance between all executions in sandboxes i and j. The evasion score is thus the difference between the maximum *inter-sandbox distance* and the maximum *intra-sandbox distance*. The evasion score is in the interval [0,1], with 0 representing the same behavior and 1 representing completely different behavior. If this score exceeds an evasion threshold, DISARM declares that the malware has performed semantically different behavior in one of the sandboxes.

5 Evaluation

To evaluate the proposed approach, we tested our system using our two monitoring technologies and three different operating system images. Table 2 summarizes the most important characteristics of the four sandboxes we employed. To simplify deployment, we ran the driver-based sandboxes inside an unmodified Qemu emulator (version 0.11), rather than on a physical system. This limits our ability to detect evasion techniques targeted against Qemu CPU emulation bugs that may be present in both monitoring technologies. In the future, we plan to extend our driver-based monitoring system to automatically analyze samples on a physical system. For this, we need to be able to reset the system to a fresh state after each analysis. As an alternative, we could instead use an existing DMAS such as CWSandbox, that already provides such functionality.

In the following we will refer to the sandboxes used for evaluation by the names shown in the first column of Table 2. The first image, used in the *Anubis* and *Admin* sandboxes, was an image recently used in the Anubis system.

Table 2. Sandboxes used for evaluation.

Sandbox	Monitoring Technology	Image Characteristics		
		Software	Username	Language
Anubis	Anubis	Windows XP Service Pack 3, Internet Explorer 6	Administrator	English
Admin	Driver	same Windows image as Anubis		
User	Driver	Windows XP Service Pack 3, Internet Explorer 7, .NET framework, Java Runtime Environment, Microsoft Office	User	English
German	Driver	Windows XP Service Pack 2, Internet Explorer 6, Java Runtime Environment	Administrator	German

We selected two additional images that included a significantly different software configuration. The three images differ in the language localization, the username under which the malware is running, as well as the available software and software versions. Each sample is analyzed three times in each of the four different sandboxes, resulting in a total of 12 runs per sample.

5.1 Training Dataset

To develop our techniques and select a threshold for evasion detection, we created a small, labeled training set. For this, we selected 175 malware samples among those submitted to Anubis in August 2010. To ensure this small dataset was as diverse as possible, we selected samples belonging to 175 different malware families, based on virus labels from Kaspersky Anti-Virus. Furthermore, we included ten additional samples based on comments from Anubis users, for a total of 185 samples.

To establish a "ground truth" for the training set we manually inspected the generated behavioral profiles and in a few cases even the binaries themselves. Manual classification of these samples showed that 43 (23.24 %) displayed different behavior in at least one of the sandboxes. These samples were further split into four groups: samples evading Anubis, samples evading our driver, samples requiring the .NET framework and therefore only working in the *User* sandbox (see Table 2), samples not working in the *German* sandbox due to hardcoded paths, and samples showing different behavior due to various other installation characteristics (e.g. the missing installation of Java in the Anubis image). Table 3 summarizes the amount of samples for each class.

Our techniques succeeded in assigning low evasion scores to samples with the same behavior and high scores to samples displaying different behavior. To visualize these results, we plotted the maximum diameter against the maximum distance discussed in Section 4.2 in Fig. 2. The overall score can be seen as the distance of each point from the diagonal. Points close to the diagonal represent samples with low scores, while points farther away from the diagonal represent

Table 3. Observed behavioral differences in the training set.

		Description of behavior
76.76 %	142	same behavior
9.19 %	17	evasion of Anubis
5.41 %	10	.NET environment required
3.24 %	6	evasion of our driver
3.24 %	6	different behavior due to other characteristics
2.16 %	4	not working in the German environment

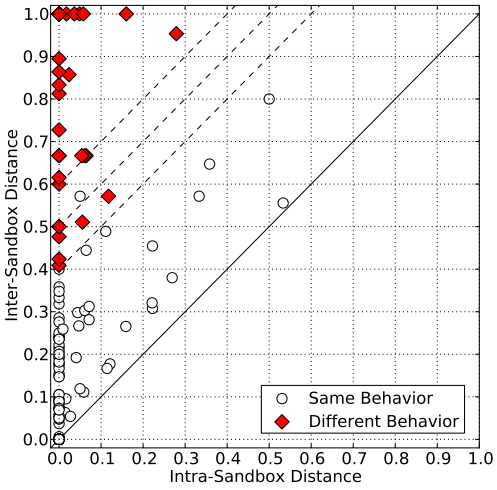

Fig. 2. Maximum diameter (*intra-sandbox distance*) vs. maximum distance (*inter-sandbox distance*) with thresholds (0.4,0.5,0.6).

samples with high scores. Points close to the y-axis are samples exhibiting little variation between analysis runs in the same sandbox. This is the case for the larger part of our training set, confirming the effectiveness of our normalization techniques. Only 8.11 % display a maximum intra-sandbox variation greater than 0.1 as a result of non-deterministic behavior such as crashes that occur only in some executions.

In Fig. 2, the samples classified as exhibiting different behavior are displayed as filled points, while those with the same behavior are displayed as empty points. Threshold candidates are displayed as parallels to the diagonal. For the training set a threshold of 0.4 results in detecting all samples with different behavior, while incorrectly classifying only one sample.

To measure the effect of the various normalization steps on the results, we calculate the proportion of correctly classified samples in the training set for each possible threshold. This metric, called accuracy, is defined as follows:

$$accuracy = \frac{|True\ Positives| + |True\ Negatives|}{|All\ Samples|} \cdot 100. \tag{3}$$

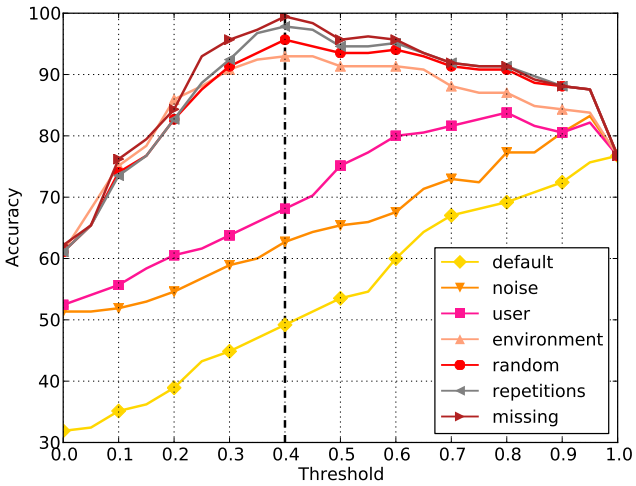

Fig. 3. Overall accuracy for each normalization step at thresholds [0,1].

We applied the normalization steps, as described in Section 4.1 in ascending order and calculated the accuracy for each step (see Fig. 3): no normalization (*default*), the removal of noise (*noise*), the generalization of user-specific artifacts (*user*), the generalization of environment-specific artifacts (*environment*), the detection and generalization of random names (*random*), the detection of repetitions (*repetitions*), and the generalization of missing filesystem and registry resources (*missing*).

Overall we achieved an accuracy of more than 95 % for thresholds between 0.3 and 0.6, with the highest accuracy of 99.5 % at the chosen threshold of 0.4. Every normalization step improved the accuracy to some degree, with the removal of noise and the generalization of user and environment yielding the largest improvements. As we discussed in Section 4, DISARM considers persistent actions for the behavioral comparison. If we instead consider all actions, we obtain a maximum accuracy of 89.2 %.

5.2 Large Scale Test

We tested DISARM on a larger dataset of 1,686 samples submitted to Anubis between December 2010 and March 2011. We selected a maximum of five samples per sample family as classified by Kaspersky to make sure this dataset is diverse. We used the evasion threshold of 0.4 selected in the previous section. Note that, since we do not have a ground truth for this dataset, we are unable to provide an accuracy score.

In this dataset, a total of 431 (25.56 %) samples resulted in an evasion score above the threshold. Table 4 breaks these results down for each pair of sandboxes. The sandboxes *Anubis* and *Admin* use the same Windows image. We can

Table 4. Samples with evasion scores above the threshold of 0.4 for each pair of sandboxes.

	Anubis	**Admin**	**User**	**German**
Anubis	-	252 (14.95 %)	333 (19.75 %)	303 (17.97 %)
Admin	-	-	121 (7.18 %)	58 (3.44 %)
User	-	-	-	138 (8.19 %)
German	-	-	-	-

Table 5. Samples with at least one different persistent action for each pair of sandboxes.

	Anubis	**Admin**	**User**	**German**
Anubis	391 (23.19%)	729 (43.24%)	733 (43.48%)	755 (44.78%)
Admin	-	440 (26.10%)	727 (43.12%)	730 (43.30%)
User	-	-	443 (26.28%)	780 (46.26%)
German	-	-	-	435 (25.80%)

therefore attribute these results to evasion of one of the monitoring technologies employed. Conversely, different behavior for any combination of *Admin*, *User* and *German* indicates evasion or environment sensitivity related to differences in the Windows environment. The results for *Anubis-User* and *Anubis-German* are a combination of both factors and therefore produce the highest rates of evasion.

Table 5 demonstrates that simply assuming a malware sample is evading an analysis sandbox whenever two executions differ by as little as a single persistent action (as was done in [8]) leads to misleading results. Close to one quarter of samples display some difference in behavior even among executions in the same sandbox. The problem is further exacerbated when considering different sandboxes: Overall 990 samples (58.72 %) behave differently in at least one sandbox according to this criterion.[4]

5.3 Qualitative Results

To get a feeling for differences causing a high ranking in our behavior comparison, we manually examined randomly selected samples with an evasion score above our threshold. The following examples illustrate how Disarm succeeded in uncovering new methods used by malware to detect the presence of Anubis, as well as pointing out ways to improve the configuration of the Windows images in our analysis sandboxes.

Anubis Evasion. We found several timing-sensitive samples, that evade analysis by detecting that the environment in which they are running is much slower than a real system. For this, they use the `GetTickCount` API call before and

[4] To allow for a fair comparison across different Windows images, we applied our user and environment normalization and ignored network activity.

after executing several benign calls such as `GetCommandLine`. If the difference exceeds a threshold, these samples do not exhibit malicious behavior. We are able to detect these samples because our in-the-box monitoring technology, despite running inside Qemu, is much faster than Anubis and does not exceed the specified thresholds. This further highlights the importance of evasion detection techniques that are compatible with lightweight instrumentation. Among these samples were representatives of several well known malware families such as Zeus, Virut, Agent and Rbot.

Win32.Packed.Krap.ag.5 and *Win32.TrojanSpy.Banker.9* evade Anubis by checking if the parent process is explorer.exe. In Anubis samples are started by an analysis daemon, which terminates right away. Thus, this condition is not met and these samples exit without performing any malicious activity.

To render known evasion techniques ineffective, Anubis already randomizes a number of identifiers at each execution, such as those listed in Table 1. This is implemented by intercepting the calls to known sources for this information and forging the results. Inspecting samples detected by DISARM, however, reveals that this feature needs to be improved. Several samples query a machine GUID stored in `HKLM\SOFTWARE\MICROSOFT\CRYPTOGRAPHY\MACHINEGUID`, that is not yet covered by Anubis randomization.[5] In other cases, the randomization was insufficient to prevent evasion. *Trojan-Clicker.Win32.Wistler.d* detected Anubis by querying the computer name. Currently our computer names have the format "pc" followed by a random number. Clearly, we need to implement stronger randomization of this identifier. Finally, malware can also detect Anubis by checking the hard drive manufacturer information. The randomization feature of Anubis already intercepts the device control code `IOCTL_STORAGE_QUERY_PROPERTY` and the system call `NtQueryVolumeInformationFile` and forges the return information. Some samples, however, were able to bypass this randomization by instead using the device control code `DFP_RECEIVE_DRIVE_DATA` to retrieve the hard drive serial number and manufacturer.

Environment Sensitivity. The results of our evaluation also exposed various configuration flaws in the image currently used in Anubis. In this image, third party extensions for Internet Explorer are disabled. *AdWare.Win32.InstantBuzz* queries this setting and terminates with a popup asking the user to enable browser extensions. Four samples, e.g. *Trojan.Win32.Powp.gen*, infect the system by replacing the Java Update Scheduler. Clearly, they can only show this behavior in the sandboxes in which the Java Runtime Environment is installed. Microsoft Office is only installed in one of our sandboxes and is targeted by *Worm.Win32.Mixor*. *P2P-Worm.Win32.Tibick.c* queries the registry for the presence of a file-sharing application and fails on images where the Kazaa file-sharing program is not installed. Using this insight we are able to modify the image used in Anubis in order to observe a wider variety of malware behavior.

Driver Evasion. We prevent samples from loading drivers in order to maintain the integrity of our kernel module. Nonetheless, we found samples that not

[5] Note that this is a different identifier than the hardware GUID listed in Table 1, which Anubis already randomizes.

only detect our logging mechanism, but also actively tamper with our SSDT hooks. At least 20 samples employ mechanisms to restore the hooks to their original addresses and therefore disable the logging in the driver. This can be done from user space by directly accessing `\device\physicalmemory` and restoring the values in the SSDT with the original values read from the ntoskrnl.exe disk image [31]. Another ten samples achieve the same effect by using the undocumented function `NtSystemDebugControl` to directly access kernel memory. These techniques are employed by several popular malware families such as Palevo/Butterfly, Bredolab, GameThief and Bifrose, probably as a countermeasure against Anti-Virus solutions. By disabling access to kernel memory and instrumenting additional system calls, it is possible for us to harden our driver against such techniques, so long as the kernel is not vulnerable to privilege-escalation vulnerabilities.

False Positives. False positives were caused by samples from the Sality family. This virus creates registry keys and sets registry values whose name depends on the currently logged in user: `HKCU\SOFTWARE\AASPPAPMMXKVS\A1_0` for "Administrator" and `HKCU\SOFTWARE\APCR\U1_0` for "User". This behavior is not random and not directly related to the user name and therefore undetected by our normalization.

5.4 Limitations

Our results have shown that Disarm is able to detect evasion techniques used in current, real-world malware samples. However, a determined attacker could build samples that evade detection in ways our current system cannot detect. In this section, we describe a few mechanisms an attacker could leverage, as well as possible countermeasures.

First of all, malware could evade detection with Disarm if it were able to evade analysis under *all* of our sandboxes. Therefore, sandbox characteristics that are shared across monitoring technologies are of particular concern. Evasion that is based on a sandbox's network environment, for instance, would currently be successful against Disarm, because all of our sandboxes currently share a similar network environment. Malware authors could identify the public IP addresses used by our sandboxes, and refuse to function in all of them. To address this problem, we plan to configure our sandboxes to employ a large and dynamic pool of public IP addresses. These can be obtained from commercial proxy services or from ISPs that provide dynamic IP addresses to consumers. More sophisticated attacks can try to detect the restrictions placed on a malware's network traffic to prevent it from engaging in harmful activity such as sending SPAM, performing Denial of Service attacks or exploiting vulnerable hosts. Another characteristic that is common to many different monitoring technologies is the fact that they impose a performance overhead, and may thus be vulnerable to timing attacks. As we showed in Section 5.3, our driver-based monitoring technology, even running inside Qemu, was fast enough to escape timing-based detection from some malware samples. However, more aggressive

timing attacks would presumably be able to detect it. We can make timing-based detection considerably harder by running the driver on a physical system instead of in an emulator.

Malware authors aware of the specifics of our system could also attack DISARM by trying to decrease their evasion score. Since the evasion score is the difference between the inter-sandbox distance and the intra-sandbox distance, this can be achieved by decreasing the former or increasing the latter. To increase the intra-sandbox distance, an attacker could add large amounts of *non-deterministic* behavior to the malware program. Here, one must consider two things, however: First, a sandbox that can provide fine-grained instrumentation (such as Anubis) may be able to detect execution that is highly dependent on random values [3], and flag such samples as suspicious. Second, implementing truly randomized behavior without impacting the reliability and robustness of the program can be rather challenging. Unstable malware installations are likely to raise suspicion, lead to fast removal from a system, or increase attention from malware analysts - three outcomes truly unfavorable to an attacker.

Conversely, malware authors could try to decrease their intra-sandbox distance. Since we currently compute the distance between two behavioral profiles using Jaccard index, this can be achieved by adding a number of identical features to the execution on all sandboxes. To defeat this attack, we could experiment with evasion scores calculated from the set difference of each pair of profiles, rather than from their Jaccard distance.

6 Related Work

Transparent Monitoring. To prevent sandbox detection, researchers have tried to develop transparent analysis platforms. Examples include Cobra [34], which is based on dynamic code translation, and Ether [9], which uses hardware assisted virtualization to implement a transparent out-of-the-box malware analysis platform. However Garfinkel et al. [12] have argued that perfect transparency against timing attacks cannot be achieved, particularly if a remote timing source (such as the Internet) is available. Pek et al. [26] have succeeded in defeating Ether using a local timing attack.

Paleari et al. [25] used fuzzing to automatically generate "red pills" capable of detecting emulated execution environments. Their results can be used to detect and fix emulator bugs before malicious code can exploit them. Martignoni et al. [22] proposed to observe malware in more realistic execution environments by distributing the execution between a security lab and multiple end-user's hosts. They thereby improve analysis coverage and are able to observe user input that triggers malicious behavior.

Evasion Detection. Chen et al. [8] were the first to develop a detailed taxonomy of anti-virtualization and anti-debugging techniques. In their experiments, 40 % of samples showed less malicious behavior with a debugger and 4 % of samples exhibited less malicious behavior under a virtual machine. However, their results were based on the comparison of single execution traces from different execution

environments (plain-machine, virtual-machine and debugger) and on considering any difference in persistent behavior to indicate evasion. Lau et al. [20] focused on virtual machine detection and employed a dynamic-static tracing system to identify VM detection techniques in packers.

Balzarotti et al. [2] proposed a system that replays system call traces recorded on a real host in an emulator in order to detect evasion based on CPU semantics or on timing. Kang et al. [18] use malware behavior observed in a reference platform to dynamically modify the execution environment in an emulator. They can thereby identify and bypass anti-emulation checks targeted at timing, CPU semantics and hardware characteristics. Moser et al. [23] explore multiple execution paths to provide information about triggers for malicious actions. Differential slicing [15] is able to find input and environment differences that lead to a specific deviation in behavior. The deviation that is to be used as a starting point, however, has to be identified manually. In contrast to these techniques, DISARM is agnostic to the type of evasion methods used in malware, as well as to the monitoring technologies employed. Nevertheless, evasive samples detected by our system could be further processed with these tools to automatically identify the employed evasion techniques.

7 Conclusion

Dynamic malware analysis systems are vulnerable to evasion from malicious programs that detect the analysis sandbox. In fact, the Anubis DMAS has been the target of a variety of evasion techniques over the years.

In this paper, we introduced DISARM, a system for detecting environment-sensitive malware. By comparing the behavior of malware across multiple analysis sandboxes, DISARM can detect malware that evades analysis by detecting a monitoring technology (e.g. emulation), as well as malware that relies on detecting characteristics of a specific Windows environment that is used for analysis. Furthermore, DISARM is compatible with essentially any in-the-box or out-of-the-box monitoring technology. We introduced techniques for normalizing and comparing behavior observed in different sandboxes, and proposed a scoring system that uses behavior variations within a sandbox as well as between sandboxes to accurately detect samples exhibiting semantically different behavior.

We evaluated DISARM against over 1,500 malware samples in four different analysis sandboxes using two different monitoring technologies. As a result, we discovered several new evasion techniques currently in use by malware. We will apply these findings to our widely used Anubis service to prevent these attacks in the future.

Acknowledgments. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec), from the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme European Commission - Directorate-General Home Affairs (project i-Code), and from

the Austrian Research Promotion Agency (FFG) under grant 820854 (TRUDIE). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

1. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated Classification and Analysis of Internet Malware. In: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID) (2007)
2. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS) (2010)
3. Bayer, U., Comparetti, P., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS) (2009)
4. Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., Kruegel, C.: A View on Current Malware Behaviors. In: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET) (2009)
5. Bayer, U., Kirda, E., Kruegel, C.: Improving the Efficiency of Dynamic Malware Analysis. In: Proceedings of the ACM Symposium on Applied Computing (SAC) (2010)
6. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference (2006)
7. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference (2005)
8. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In: Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN) (2008)
9. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2008)
10. Ferrie, P.: Attacks on Virtual Machine Emulators. Tech. rep., Symantec Research White Paper (2006)
11. Ferrie, P.: Attacks on More Virtual Machines (2007)
12. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI) (2007)
13. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows kernel. Addison-Wesley Professional (2005)
14. Jaccard, P.: The Distribution of Flora in the Alpine Zone. The New Phytologist 11(2) (1912)
15. Johnson, N.M., Caballero, J., Chen, K.Z., McCamant, S., Poosankam, P., Reynaud, D., Song, D.: Differential Slicing: Identifying Causal Execution Differences for Security Applications. In: IEEE Symposium on Security and Privacy (2011)

16. Kamluk, V.: A black hat loses control. http://www.securelist.com/en/weblog?weblogid=208187881 (2009)
17. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables. In: ACM Workshop on Recurring Malcode (WORM) (2007)
18. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating Emulation-Resistant Malware. In: Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec) (2009)
19. Kleissner, P.: Antivirus Tracker. http://avtracker.info/ (2009)
20. Lau, B., Svajcer, V.: Measuring virtual machine detection in malware using DSD tracer. Journal in Computer Virology 6(3) (2010)
21. Martignoni, L., Christodorescu, M., Jha, S.: OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (2007)
22. Martignoni, L., Paleari, R., Bruschi, D.: A Framework for Behavior-Based Malware Analysis in the Cloud. In: Proceedings of the 5th International Conference on Information Systems Security (ICISS) (2009)
23. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: IEEE Symposium on Security and Privacy (2007)
24. Paleari, R., Martignoni, L., Passerini, E., Davidson, D., Fredrikson, M., Giffin, J., Jha, S.: Automatic Generation of Remediation Procedures for Malware Infections. In: Proceedings of the 19th USENIX Conference on Security (2010)
25. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In: Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT) (2009)
26. Pek, G. and, B.B., L., B.: nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In: ACM European Workshop on System Security (EUROSEC) (2011)
27. Perdisci, R., Lee, W., Feamster, N.: Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In: USENIX Conference on Networked Systems Design and Implementation (NSDI) (2010)
28. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting System Emulators. In: Information Security Conference (ISC) (2007)
29. Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction. http://invisiblethings.org/papers/redpill.html (2004)
30. Stone-Gross, B., Moser, A., Kruegel, C., Almaroth, K., Kirda, E.: FIRE: FInding Rogue nEtworks. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (2009)
31. Tan, C.K.: Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration. Tech. rep., SIG2 G-TEC Lab (2004)
32. The Honeynet Project: Know Your Enemy: Fast-Flux Service Networks. http://www.honeynet.org/papers/ff (2007)
33. Trinius, P., Willems, C., Holz, T., Rieck, K.: A Malware Instruction Set for Behavior-Based Analysis. Tech. Rep. 07–2009, University of Mannheim (2009)
34. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In: IEEE Symposium on Security and Privacy (2006)
35. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy 5(2) (2007)
36. Yoshioka, K., Hosobuchi, Y., Orii, T., Matsumoto, T.: Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. Journal of Information Processing 19 (2011)