SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion

Akira Yokoyama[1], Kou Ishii[1], Rui Tanabe[1], Yinmin Papa[1], Katsunari Yoshioka[1], Tsutomu Matsumoto[1], Takahiro Kasama[2], Daisuke Inoue[2], Michael Brengel[3], Michael Backes[3], and Christian Rossow[1 3]

[1] Yokohama National University
{yokoyama-akira-bs|ishii-kou-yf|tanabe-rui-nv}@ynu.jp
yinminpapa@gmail.com
yoshioka@ynu.ac.jp
tsutomu@mlab.jks.ynu.ac.jp

[2] National Institute of Information and Communications Technology
{kasama|dai}@nict.go.jp

[3] Center for IT-Security, Privacy, and Accountability, CISPA, Saarland University
{mbrengel|crossow}@mmci.uni-saarland.de
backes@cs.uni-saarland.de

Abstract. To cope with the ever-increasing volume of malware samples, automated program analysis techniques are inevitable. Malware sandboxes in particular have become the *de facto* standard to extract a program's behavior. However, the strong need to automate program analysis also bears the risk that anyone that can submit programs to learn and leak the characteristics of a particular sandbox.
We introduce SandPrint, a program that measures and leaks characteristics of Windows-targeted sandboxes. We submit our tool to 20 malware analysis services and collect 2666 analysis reports that cluster to 76 sandboxes. We then systemically assess whether an attacker can possibly find a subset of characteristics that are *inherent* to all sandboxes, and not just characteristic of a single sandbox. In fact, using supervised learning techniques, we show that adversaries can automatically generate a classifier that can reliably tell a sandbox and a real system apart. Finally, we show that we can use similar techniques to stealthily detect commercial malware security appliances of three popular vendors.

1 Introduction

Malicious software poses one of the major security challenges nowadays. In its various forms, malware is equally a threat to consumers (e.g., banking trojans, ransomware), businesses (e.g., targeted attacks, denial-of-service bots), and society in general (e.g., spambots). In 2014, Symantec faced 65 million previously-unseen malicious files targeting Windows [51]. Similarly, PandaLabs reports on a daily flood of over 200,000 new unknown, potentially malicious programs [49]. This trend of increasing malware samples is a consequence of polymorphism, but is also caused by new threats that are discovered almost on a daily basis.

To cope with the volume of malware, defenders started to improve technology and their organization within the community. On the technological side, researchers introduced several complementary approaches to analyze unknown programs in an automated way. Windows-based malware sandboxes in particular have become the *de facto* standard for automated malware analysis [24], equally for academia and industry. Sandboxes are excessively leveraged to obtain threat information, such as previously-unseen malware, inputs for supervised detection mechanisms, malware C&C servers, targets of banking trojans, intelligence on spreading campaigns, or simply to assist in the manual process of reverse engineering. Finally, sandboxes are also used as part of commercial malware security appliances that aim to protect organizations by dynamic malware analysis.

The requirement to automate the analysis of unknown programs ("samples") also bears the risk that the analysis is unattended. That is, oftentimes the entire process from receiving a sample, scheduling it for analysis, executing it, and possibly even returning an analysis result to the sample submitter is embedded in a fully-automated processing chain. Anyone that can submit samples to the input feed of a sandbox can possibly learn and leak the characteristics of a particular sandbox. While it may seem non-trivial to send programs to any sandbox, as we will show, it is typically sufficient to submit a sample to automated malware analysis services, which then redistribute the samples for other sandboxes, easily generating a massive source of insights about the internals of global sandboxes.

In this paper, we follow this general idea and introduce SandPrint, a Windows-based program that measures and leaks characteristics of the sandbox, such as precise OS information, network configuration or installed (or emulated) hardware. Over a period of 2 weeks, we continuously submit our tool to 20 malware analysis services ([1,2,3,20,5,25,38,6,7,28,8,35,48,52,50,11,12,55,13,14])and collect 2666 analysis reports from eleven of these services. In an attempt to fingerprint sandboxes, we use unsupervised learning mechanisms to cluster the SandPrint reports and their various features into groups to identify sandboxes. This process exposes 76 sandboxes, many of which presumably obtain their samples via automated sample exchanges with malware analysis services.

We then turn to our next research question: Is it possible to *detect* a sandbox from the perspective of a (potentially malicious) program? By now, many malware families already follow ad-hoc procedures to identify individual sandbox artefacts, such as detecting virtualization or avoiding specific sandbox configurations. Instead, we assess a more systematic approach and explore whether an attacker can possibly find characteristics (e.g., using a tool like SandPrint) that are *inherent* to all sandboxes, and not just characteristic of a single sandbox. We leverage supervised learning based on the collected features to train an automated classifier that can reliably tell sandboxes apart from normal systems.

Finally, we turn our attention to the possibility of detecting malware appliances of popular vendors. We follow the intuition that these appliances internally also use sandboxing technology and are thus likely susceptible to similar evasion attacks. In fact, by training a classifier just on the aforementioned sandboxes, we show that adversaries can even evade appliances—undermining the entire security concept of such installations.

Summarizing, the contributions of this paper are:

– We present SandPrint, a tool to exfiltrate characteristics of malware sandboxes. We submit it to 20 public malware analysis services and use unsupervised learning techniques to identify the characteristics of 76 sandboxes.
– We leverage the resulting SandPrint reports to train an automated classifier that can reliably distinguish between a sandbox and a user system.
– We show that we can use characteristics that we learned from public sandboxes to detect malware security appliances, even without *a priori* insights on the internals of the appliance's sandbox.

2 Background

We first describe the terminology that we use throughout the paper. We use the term *sandbox* to refer to a dynamic analysis environment that executes unknown programs (called *samples*). Sandboxes are widely used to gain insights on malicious software (*malware*), such as its current campaigns [19], recent C&C servers and traffic patterns [46,36,44] or attack targets [26]. Similarly, sandboxes can be used to group behavior into malware families [16,45], or to identify suspicious behavioral patterns [32]. Egele et al. give a comprehensive overview of known sandbox implementations [24].

By now, malware sandboxes are not used only by academics. In contrast to manual analysis, sandboxes are highly automatized. As a consequence, the antivirus industry and security companies that offer anti-malware appliances heavily rely on sandboxes as part of their daily business. In fact, sandboxes have become the industry standard to cope with the daily feed of hundreds of thousands of previously-unseen malware samples. In times where manually analyzing each malware does not scale anymore, sandboxes are a vital component in the fight against malware.

Virtualization: To scale the analysis, most sandboxes rely on some form of virtualization (with the notable exception of bare-metal sandboxes [30,31]). To this end, sandboxes rely on various virtualization techniques such as VMWare [15] and VirtualBox [10] or CPU emulators [17,4]. Cuckoo Sandbox [8] is a popular open-source sandbox. However, as we will see, many security organizations operate other sandboxes, either choosing from commercial sandboxes or designing their own solution. Virtualization offers the benefit that many virtual machines (VMs) can run in parallel on a single system, each analyzing one sample. In addition, virtualization software makes it possible to take so called *snapshots* of VMs, which freeze the state of a VM and allow reversion back to this state. Snapshots help to reset the system state once a piece of malware was executed, so that future executions do not suffer from side effects.

Operating System: Regardless of precise sandbox implementations, the most common and popular sandboxes execute samples on commodity operating systems, such as MS Windows. Due to the prevalence of Windows-based malware,

we focus on Windows-based sandboxes, although our results likely also apply to Android-based [37] or Linux-based [39] sandboxes.
Malware Analysis Services: In contrast to a sandbox, a *malware analysis service* (or simply "service" hereafter) receives submissions of samples (e.g., via a web interface), analyzes the submitted samples in various ways, and normally provides analysis results to the users. As we will show, these services typically use one or more sandboxes to analyze the sample. In addition, services such as VirusTotal [14] and Jotti's malware scan [28] provide anti-virus scans. Also, it is quite common that services share samples with other sandbox operators. In fact, some services offer to search for submissions of other users with various key words, such as the hash value of the sample or anti-virus labels, and parties can use paid APIs to automate searches and downloads. Table 2 summarizes the analysis results provided by each service. There are nine services that accept the submission of samples but do not provide any analysis results to the users. Following the feedback by the vendors, the submitted samples were manually analyzed in an isolated environment without Internet access.
Malware Security Appliances: Sandboxes have become an integral part of commercial malware security appliances (or simply "appliances" hereafter). Such appliances protect endpoints by dynamically analyzing an unknown program and inspecting its behavior for suspicious actions. Appliances are frequently deployed at the network layer and are used orthogonally to anti-virus scanners, e.g., to protect endpoints from opening malicious email attachments or malicious file downloads. Internally, appliances also use sandbox technologies to analyze the program behavior.

3 Sandbox Fingerprinting

Sandboxes are a vital tool for malware analysis, which highlights the importance of having a thorough understanding of their live deployment and characteristics. In this section, we will *fingerprint* sandboxes to investigate the landscape of Internet-connected sandboxes. In our context, a fingerprint reveals artifacts that are specific to individual sandboxes. In Section 3.1, we will describe 24 features, i.e., attributes that reveal certain characteristics of a sandbox. We will then present SandPrint, a tool that exfiltrates characteristics from any sandbox. In Section 3.2, we will use the tool to collect fingerprints by submitting it to 20 public malware analysis services, and describe the dataset obtained.

3.1 Sandbox Fingerprinting Features

We first introduce fingerprinting features that we use to discriminate sandboxes from each other, or put differently, to describe characteristics of individual sandboxes. We propose a (non-exhaustive) list of 24 features, as shown in Table 1. We group these characteristics into the following five categories:

1. **System Installation**: Sandboxes require an operating system (OS) to run samples. Typically, to minimize manual effort, sandbox operators install and

Table 1: Sandbox Fingerprinting Features (Clustering distance metrics: ED = Edit Distance, EU = Euclidean Distance, EQ = Equality Distance, JD = Jaccard Distance)

Category	Feature	Clustering	Category	Feature	Clustering
System	Host name	✓ED	**Network config**	Default gateway	✓EQ
Installation	Installation date	✓EQ		External IP address	✓EQ
	OS information			ARP list	
	Organization name	✓EQ		MAC address	
	Owner name	✓EQ		DNS servers	✓EQ
	Windows product ID	✓EQ	**Activity**	Clipboard	
	System Manufacturer			Desktop icons	
Hardware	Disk space	✓EU		Event log	
	Display resolution			Recent files	✓JD
	Mouse devices		**Execution start**	Sample name	✓EQ
	RAM	✓EQ		Sample path	✓EQ
	Processor			Time from boot to start	

configure the OS only once, and then take a snapshot of the system. Assuming that all parallel instances of the sandbox (e.g., VMs) use the same system snapshot, this results in many features that are persistent across executions.

2. **Hardware**: The underlying hardware, whether emulated or physical, can also reveal unique characteristics of a sandbox. All features are agnostic to whether the hardware is emulated or actually physically present.
3. **Network Configuration**: Sandboxes are typically configured such that the sample can communicate with the Internet, e.g., with C&C servers. We thus collect network features and also local configurations.
4. **Activity**: The system snapshot reveals certain events that have taken place in the past (i.e., during installation time). In general, these features can measure whether the system is close to default settings.
5. **Execution Start**: Once the fresh, non-infected sandbox has been started, it needs to obtain a sample that the analyst wants to analyze. There are multiple different ways to automate this process.

This list can easily be extended with further features. We favor features that potentially show a high entropy and are specific to a certain sandbox—the more a feature can differentiate a sandbox from others, the better. In addition, most of the selected features are deterministic and their values discrete and reliable. Note that a stealthy sandbox could try to diversify the feature values.

3.2 Extracting Sandbox Fingerprints with SandPrint

We implemented SandPrint, a tool to exfiltrate all above-mentioned features from a system. SandPrint is a Windows 32-bit PE binary written in C, which uses the Windows API and custom functions to reveal the features. Once SandPrint is executed, it uses HTTP to communicate with the SandPrint server. A unique identifier is assigned to each SandPrint sample and after the HTTP session is established, this ID, embedded in an HTTP POST request, is sent to the server. In this way we can track which sample is executed in which system. After the ID is sent, a challenge-response authentication is done in order to detect replayed requests.

After this initial handshake, SandPrint starts collecting features of the system. For the implementation of feature collection, we avoided using commands

Table 2: Malware Analysis Services and summary of SandPrint submissions

Malware Analysis Service	Dyn.	Stat.	AV Scan	report/ submission	reports	sandboxes
Service #1	✓	✓		0/20	0	0
Service #2	✓			14/20	14	1
Service #3			✓	0/20	0	0
Service #4	✓			0/20	0	0
Service #5			✓	0/20	0	0
Service #6				6/20	85	25
Service #7		✓	✓	2/20	8	6
Service #8			✓	20/20	21	1
Service #9			✓	0/20	0	0
Service #10			✓	20/20	378	36
Service #11	✓	✓	✓	20/20	134	28
Service #12				19/20	25	1
Service #13	✓	✓		20/20	427	36
Service #13 (win7 64 bit)	✓	✓		20/20	399	49
Service #13 (win7 32bit stealthy)	✓	✓		20/20	424	35
Service #14	✓			0/20	0	0
Service #15				0/20	0	0
Service #16	✓	✓	✓	20/20	268	26
Service #17	✓			0/20	0	0
Service #18			✓	20/20	162	20
Service #19		✓		0/20	0	0
Service #20	✓	✓	✓	20/20	321	31

like `systeminfo`, `netstat`, and `ipconfig`, as they are often used by adversaries to collect system features, and indeed we have confirmed that some sandboxes restrict them. Moreover, to avoid potential deadlocks caused by collecting individual features (due to e.g., slow disk I/O), we balanced all feature collection functions across multiple threads. In addition, to estimate the overall execution time, a heartbeat thread periodically notifies the server that SandPrint is still executing. Each thread sends features to the server after the feature collection process is completed. Note that all SandPrint traffic imitates HTTP protocol and so it seems as if it is communicating with a Web server.

We submitted SandPrint to 20 malware analysis services to collect fingerprints. Table 2 summarizes the public services and includes both popular academic and non-academic services. We periodically submitted SandPrint from January 5, 2016 to January 18, 2016.

For each service, we created a unique SandPrint instance so that we could map which file was uploaded where. That is, while the semantic functionality is unaltered, the resulting file hashes are distinct. In addition, we use a unique identifier that is computed during runtime, report this identifier to our server, and aim to expose it in the public analysis report that the service generates. This way, we can later match the identifier in a report with the corresponding identifier of the analysis report, revealing that a report was generated by a particular service. In total, we collected 2666 SandPrint reports from 221 of our 440 submissions. Thus, on average, we received 6 reports per submission. The reports came from 395 IP address including 33 countries. As we will discuss later, this already indicates that there is a strong tendency to (i) re-execute the same sample multiple times (on the same or a slightly different sandbox) and (ii) share samples across sandboxes/services. We will now study this observation in more detail and group similar SandPrint reports for further analyses.

4 Clustering Sandboxes

The fingerprint collection revealed over 2500 reports. But are there really that many sandboxes, or are some sandboxes responsible for multiple reports? To answer this question, in this section, we introduce a clustering technique to group similar reports and identify which reports were sent to us by which sandboxes.

4.1 Clustering

Initial observations have shown that subsets of the entire list of reports actually share similar characteristics. As soon as a sandbox sends multiple reports, this is intuitive, as there are likely features that remain unchanged across two sample executions. Naïvely, one could even check which reports contain equal features. However, we found that sandboxes indeed (intentionally or not) diversify parts of the features. Instead, we thus propose to use unsupervised learning techniques to group *similar* reports together. Lacking any labels and ground truth for sandboxes, we face a classical unsupervised problem here. We chose to use agglomerative hierarchical clustering to group reports. Hierarchical clustering has the advantage that it allows specifying a custom distance function and does not require determining the number of expected clusters in beforehand. The *distance function* determines how different two reports are. We define a distance function that spans all "clustering" features in Table 1 (see checkmark). That is, for a pair of reports R1 and R2, we sum the distances of all pairwise features and divide by the number of features to achieve the average distance. More formally, the distance function between $R1$ and $R2$ is: $dist(R1, R2) = \frac{1}{N} * \sum_{k=0}^{N} dist_k(R1, R2)$ where $dist_k$ is the distance between the values of a particular feature k. When comparing a feature between two reports, we expect equality (EQ), and otherwise assume maximum dissimilarity. That is, $dist_k(R1, R2)$ is zero if the feature k is equal in both reports, or 1 otherwise. For selected features which we observed to vary in individual sandboxes, we do not expect equality. That is, we compare the host name using a normalized edit distance (ED), deploy the Euclidean distance (EU) to compare the disk space and length of the sample name, and use the Jaccard distance (JD) to compare the recently opened files. Table 1 categorizes the features accordingly. All distance functions have been normalized in the range $[0, 1]$ so that a single feature does not introduce bias.

In some cases, features are not present in one of the reports to compare. SandPrint may have failed to collect some features, e.g., if the sandbox analysis time was too short to complete all measurements (e.g., tracking all files in the Programs directory may take a long time). To tackle sparse features, we focus on those features that are included in the majority of the reports, as indicated by the checkmark in Table 1. If a report does not have characteristics for a remaining feature, we still cannot judge if two features are similar. To tackle this problem, we ignore features that are not present in both reports and decrement N (the number of features) accordingly to avoid biases in the average.

We then compute the distance between all reports and group the most similar ones together, using agglomerative single linkage clustering. This process results in a dendrogram, a tree-like structure that represents how the reports are clustered together. After the clustering, we consider groups that have a distance less than 0.5 as clusters. The intuition for this threshold is that we expect that at least half of the features are similar for reports of a single sandbox.

4.2 Clustering Results and Validation

Clustering helped to reduce the 2666 reports down to 76 clusters. Of these, 16 are singleton clusters, i.e., sandboxes that only contributed one report to our dataset. The largest cluster spans 233 reports, while the average cluster consists of 35 reports, or 44 reports if we exclude the singleton clusters.

To verify the clustering output, we divided our research team in two disjoint groups. While one group independently designed and performed the automated clustering, the other group validated the clustering output. To this end, we manually grouped similar sandboxes based on unique characteristics that we identified for a particular sandbox, explicitly also those that slightly varied information across different executions. For each such outstanding feature, we defined a regular expression that matches all reports of the sandbox. We only selected features whose entropy was large enough to avoid coincidental collisions and define at least two characteristic features per sandbox.

We then compared the clustering result with the outcome of the manual "clustering" done by the validation group. The outcome of the manual assignments was equal to the clustering result, except in one case where our clustering merged two sandboxes that we did not group manually. In this case, while the user name, working group name, and host name were similar, the OS installation date was more than three years apart. Other than that, we did not find any further inconsistencies, which shows that our clustering methodology can accurately map SandPrint reports (and their features) to a smaller number of sandboxes.

4.3 Sandbox vs. Service

Table 2 summarizes the results of SandPrint submissions to the 20 malware analysis services. At a glance, the number of SandPrint reports received from these services varies widely. We did not receive any reports from nine services, which implies that sandboxes deployed by these services do not have Internet connectivity, or the services simply did not conduct any dynamic analysis on the submitted samples. Note that SandPrint is implemented such that it first reports back to our server before collecting any features. As we also did not see the initial connection for the nine services, we argue that the lack of reports is not caused by sandboxes that are trying to avoid being fingerprinted. Due to the lack of data, we exclude these nine services and will focus on the remaining eleven services in the following.

4.4 Mapping Malware Analysis Services to Sandboxes

Next, we aim to map the SandPrint reports to malware analysis services. In other words, did our fingerprinting help to expose internals of the sandbox(es) used by a service? To map sandboxes to services, we followed a two-fold approach.

First, we studied the analysis reports (i.e., those provided by the services, and not by SandPrint) that were returned by a service. These reports include the behavior of the submitted samples. Recall that we encoded a unique identifier in each SandPrint submission, which became visible in the analysis reports. We found this identifier in the analysis reports of services #2, #11, #13, and #20.

Second, to map the remaining services, we analyzed whether some sandboxes were exclusively used when submitting a sample to a particular service. That is, we identify sandboxes that are seemingly attached to a certain service. Figure 3 (see Appendix) depicts the mapping between all samples submitted to eleven malware analysis services (y-axis) and 76 sandboxes according to the report clustering (x-axis). Some mappings could be confirmed by the analysis reports. Next to these confirmed mappings, we find that some sandboxes are frequently and exclusively used by the same service. For example, Sandbox 69 is constantly used by Service #11 and no other services. In such a case, we can with some likelihood conclude that the sandbox is exclusively used by the service. In total, we revealed the dedicated sandboxes for four of the eleven services in this way.

After we mapped services to sandboxes, we were left with 71 sandboxes that do not directly belong to one of the services. This is also shown in Figure 3, which lists many sandboxes that are commonly used to analyze samples from various services. The degree of activity per sandboxes is an indicator for the coverage, i.e., how many samples a sandbox receives and executes. But foremost, it highlights that samples are actively shared among the services.

4.5 Empirical Sandbox Analysis

After determining the sandboxes, we will highlight some insights obtained from the collected features. First, we inspected the system installation features. We found that the most popular OS for these sandboxes is still Windows XP, counting 37 out of 71 sandboxes for which we could identify the OS. 29 sandboxes were Windows 7. The other 5 sandboxes run Windows 8. The installation date can approximate the age of a sandbox. Assuming the other installation dates are not faked, we can see that all of the obtained OS installation dates are between the years 2008 and 2016. We also see that more than half of the sandboxes are at least three years old. As of 2014, 10 sandboxes were installed and already 18 sandboxes in 2015 or 2016. It is notable that the Windows product ID of 41 sandboxes is static, while 18 sandboxes vary the value. We presume this serves for diversification purposes, as malware has been observed to use the Windows product ID as a feature for sandbox identification.

The distribution of sandbox host names and owner names falls into two extreme cases, namely, they are either highly diversified or completely static. We deduce that some sandbox developers take countermeasures against being fingerprinted, while many others do not. Among the sandboxes that diversify host and owner names, the randomized names of most sandboxes still exhibit common patterns, such as common prefixes and/or fixed length of the strings.

In some cases, we can infer sandbox implementations. Namely, Cuckoo Sandbox includes a particular file named *agent.py*, which must be running upon the analysis of a sample. We can infer that Cuckoo Sandbox is installed and running by checking if the recent files list includes *agent.py*. We infer that five sandboxes are implemented with Cuckoo Sandbox. Note that the sandboxes are not clustered together, although they use the same technology. This is mainly due to the fact that sandbox operators have to set up their own VM image, regardless of whether they use common frameworks like Cuckoo. Although some sandboxes use the same virtualization technology, these sandboxes can still be distinguished based on their installation features (such as OS installation date or product ID).

Next, we inspected the Internet uplinks used by the sandboxes. 64 sandboxes use external IP addresses of a single country according to GeoIP. Among them, the US comes first with 22 sandboxes, Germany ranks second with six sandboxes, China ranks third with five sandboxes, and Ireland ranks fourth with four sandboxes. Three sandboxes each are in Sweden, Russia, and Korea, and Romania, Japan, and Britain host two sandboxes. We note that there are two sandboxes that we cannot geolocate due to their high diversity of external IP addresses. These sandboxes use Tor to diversify the IP address. We also note that 29 sandboxes use a single fixed IP address, which makes them trivially detectable from the server side. For instance, if a malware sample sends a command to a C&C server, the IP address could be checked against a black list on that server, which then tells the client to stop executing.

The MAC addresses show the highest diversity in all features we collected. Only 12 sandboxes use a single fixed MAC address, as confirmed by multiple SandPrint reports. The majority of MAC addresses are at least partly diversified (e.g., the first three octets, namely the vendor ID, are often fixed but the rest are diversified). We speculate this is due to the fact that the sandboxes actually consist of multiple VMs running in parallel, sharing the same VM image, but all having unique MAC addresses to avoid collisions on the Ethernet layer. Of those sandboxes that did not hide the vendor prefix, we could reveal 6 VMware-based (prefix: 00-50-56) and 21 VirtualBox-based (prefix: 08-00-27) sandboxes.

5 Sandbox Classification

We have shown that the fingerprints can be used to discover that certain reports belong to the same sandboxes. We now explore whether we can leverage the extracted features to judge if a system is a sandbox. Intuitively, we explore features that are *inherent* to sandboxes due to hardware constraints, their snapshot-based operations, or lack of user interactions. We will show how we can use those inherent features to detect sandboxes using supervised machine learning techniques. We will first describe the feature selection for this task and

Table 3: Sandbox classifier features.

	Feature	Observation	Transf.
Hardware	Display resolution	uncommon	`id`
	Display width	small	`id`
	RAM size	small/uncommon	`id`
	PS/2 mouse	uncommon	`{0,1}`
	#Cores	small	`id`
	Disk size	small	`id`
History	System uptime	small	`id`
	Last login	long ago	`id`
	Last file access	long ago	`id`
Execution	Image name	uncommon	`{0,1}`
	Clipboard	empty	`len`
	System manufacturer	uncommon	`len`

then outline how we design and evaluate a classifier for sandboxes with those features using Support Vector Machines (SVMs) [21].

5.1 Feature Selection

The key idea behind the feature selection is to find patterns which are characteristic for a sandbox operation but unlikely for a machine under human control. Instead of identifying specific fingerprints for particular sandboxes, we strive to find sandbox-inherent features that are common to all sandboxes.

Feature Selection Process To establish a ground truth for user PCs, we execute SandPrint on 50 commodity Windows workstations which are not used as sandboxes and are under the control of human operators. We then manually examined the reports to identify inherent and meaningful patterns which we observed in the sandbox reports but which were not as characteristic for the user reports. Table 3 summarizes the selected features, which we divide in the three categories hardware, history and execution. The second column contains the feature name, the third one describes our observations from the sandbox reports, and the last column shows how we transform the feature value to an integer before we pass it to the SVM (as we will discuss in Section 5.2).

The observations mentioned in Table 3 are a vague description of the feature characteristic. A naïve approach would be to derive sandbox signatures for concrete values, such as searching for reports with a display resolution of 4:3. This observation was made for the vast majority of sandboxes, but was uncommon for a real user. However, there are several problems when choosing such *concrete* values. First, the feature value is not necessarily so precise that such a solution would make sense. The screen resolution, for example, was not 4:3 for all sandbox reports, but 5:4 or some other suspicious value which we did not observe in the user reports. Thus, instead of figuring out concrete values and checks for each feature, we leave this task to the training process of our SVM classifier.

Similarly, we also refrain from detecting *virtualization* techniques, and rather focus on inherent sandbox techniques. Technically, we could check for artifacts that indicate the presence of a virtualization solution such as VMWare or Virtual Box, which is frequently used by sandboxes. However, we would bias our classifier towards detecting virtual machines, which is not the objective. While virtualization is definitely a hint toward the presence of a sandbox, it is also definitely not a guarantee. For instance, we found one user report which indicated that the execution was taking place in a VMWare virtual machine. Our classifier should be able to classify this machine as a user machine and not as a sandbox. Conversely, a sandbox does not necessarily use virtualization as, for example, in the case of bare-metal sandboxes. Our classifier should be able to classify those systems as sandboxes despite the absence of virtualization, as our features are based on the observation that sandboxes use snapshots, have restricted resources, and uncommon user interaction.

Feature Description We now describe the features in more detail. The *hardware* features are motivated by the fact that sandbox operators restrict resources in order to leverage parallel computation. Therefore, it is quite common that sandboxes are single core, use little RAM, and have small disk sizes, whereas these quantities are much larger on the average user PCs. Second, since sandboxes are usually not interactively used by a human, the operators often do not customize the hardware configurations. We argue that a small display size and uncommon display resolution as well as a PS/2 mouse are all indicators for a sandbox. It is worth mentioning that this is not equivalent to virtualization detection, where these configurations are usually the default as well. A user interactively using her VM likely customizes its screen resolution and increases its computation power by using more cores and more RAM. The *history* features mainly originate from the observation that sandboxes leverage snapshot technology. Prior to a malware sample being analyzed, the sandbox usually restores the system state to a previously captured clean state, which is called a snapshot. A snapshot is typically taken once when the sandbox is set up and is then used for the rest of the operation time of the sandbox (unless it is occasionally updated). As a consequence, it is likely to show history artifacts. For example, if a snapshot was taken months ago, every time the snapshot is restored, the login history would reveal that the last login was at that time. Similarly, the file access history would reveal that the last file access happened suspiciously long ago. In addition, we observed that many sandboxes had just been started, whereas user PCs usually have a longer uptime. Sandbox reports frequently show system uptimes on the order of seconds, whereas a vulnerable system that is about to be infected (e.g., via a drive-by download) likely has a significantly higher uptime.

The *execution* features stem from the sandbox showing uncommon execution patterns. We noticed, for example, that sandboxes tend to change the image executable name to something which is easier to handle in terms of automation. It is quite common that sandboxes uses MD5 hashes or generic names such as `virus.exe`, whereas the user reports indicate that such renaming is unlikely. We also found that the clipboard of the sandboxes was empty or contained seemingly-random strings, whereas users' clipboards tended to contain more meaningful values such as links, text, or file objects. Finally, we observed that sandboxes returned suspicious values for the system manufacturer, such as empty or random strings, possibly to hide real names—which we did not observe in the user reports.

5.2 Classification

We use the previously-described features to train a classifier that can automatically learn a model to predict if an unknown feature report was taken on a sandbox or a user PC. To this end, we build up a training data set that consists of all 50 user reports and up to three random samples from each sandbox cluster. In total this gives us a training set of 202 reports, 50 of which are user reports and 152 of which are sandbox reports.

For building the classifier, we use an SVM with a radial basis function kernel. To normalize the feature vector that we pass to the SVM, we need to transform the feature values into numerical values. This is done according to the last column in Table 3. Here, `id` means that we simply take the number as is, `len` means that we consider the length of the string feature, and `{0,1}` is a boolean value (in our case, to show if the report indicates a PS/2 mouse or not). Similarly, for the image name feature we check if the file image name has been altered. Since not every feature is available for every report, for reasons explained in Section 4, we decided to use mean imputation to estimate missing values. Finally, we normalize values to the $[0, 1]$ range using Min-Max Scaling.

To build a classifier, we need to specify an effective combination of the SVM regularization constant C and the kernel parameter γ. For this purpose, we use hyperparameter tuning with grid search and 10-fold cross validation to compute the accuracy of our classifier. We use 10-fold cross validation on top of this methodology to ensure that we get unbiased results. In an initial step, to evaluate the strength of each individual feature, we built a classifier for each single feature. The results of this experiment are depicted in Figure 1. As we can see, even a single feature can be used to detect sandboxes with high accuracy, with the RAM feature being the best, at an accuracy of 98.06%. However, a single feature is easier to fix for a sandbox operator than multiple features. We thus also created a classifier that trains on all features. The rightmost bar in Figure 1 shows that this classifier has a perfect accuracy of 100% (i.e. 0 false positives and 0 false negatives), illustrating the strength of combining multiple detection features.

5.3 Comparison to Existing Solutions

In order to evaluate how well our classifier performs, we decided to compare our methodology to existing work. For this purpose, we use Paranoid Fish (Pafish), a popular framework consisting of a collection of several well-known sandbox detection techniques used by malware in the wild. We encoded 45 detection techniques used by Pafish in SandPrint and performed them during each run. Using those 45 detection results, we then built a classifier in the same fashion as before. We consider each detection a feature for which we build a single classifier, and we also build a classifier for the combination of all 45 features. The accuracy results for these classifiers are depicted in Figure 2. Again, each light colored bar shows the accuracy for a single feature, and the black bar shows the accuracy for the classifier which combines all the features. As we can see, the majority of the single-feature classifiers are not much better than guessing. Two features are

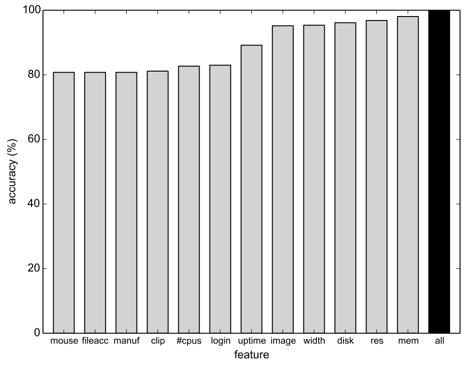

Fig. 1: Classifier accuracy (larger is better).

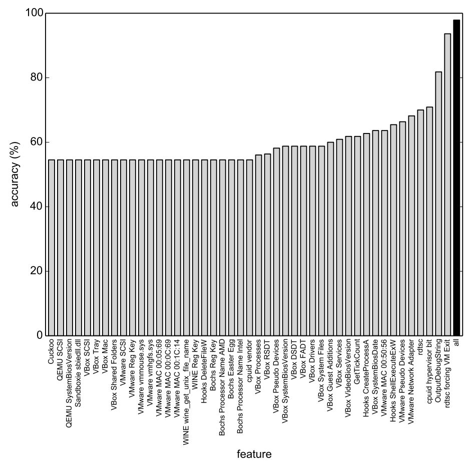

Fig. 2: Pafish classifier accuracy.

above 80% accuracy, with the best individual feature (`rdtsc` time measurements to detect virtualization [18]) having 93% accuracy. The combined version has an accuracy of 97.8%.

Besides having a better accuracy, we argue that our methodology is superior to Pafish for two additional reasons. First, Pafish mainly checks for virtualization artifacts, from which we refrain for reasons explained before. Second, the majority of the checks performed by Pafish are not stealthy by any means, since it heavily queries information from the registry, network adapters, and other sources which are likely to be monitored by the sandbox. By doing so, Pafish risks being detected as an environment-sensitive malware. In contrast, we argue our method's information extraction is stealthier. In fact, as we will see in Section 6 our approach is not even detected by state-of-the-art security appliances, which highlights the stealthiness of our approach.

5.4 Summary

As we have shown, we can reliably distinguish between a sandbox and a user machine based on sandbox-inherent features. Although the number of features seems quite small, we argue that hiding those features takes a lot of effort for the sandbox operator. While changing the screen size and switching to a USB emulated mouse is configurable, removing the parallel computation artifacts is not as simple. Increasing the number of cores and the amount of memory is likely not to be an option for the operator, as this would decrease the productivity of the sandbox. This could be solved through a solution which gives the running programs the impression of more resources. Similarly, avoiding history artifacts introduced by snapshots also requires engineering effort. For example, the sandbox operator could make sure that all the relevant history information on the system appears to be normal. A solution could be to customize sandbox snapshots and keep them up-to-date like non-sandbox systems. Unfortunately, such customization it is high effort, might be prone to errors, and likely needs to be reimplemented for every operating system under analysis. For other countermeasures which could be applied by sandbox operators, we refer to Section 7.1, where we combine this aspect with an ethical discussion of our work.

6 Malware Appliance Detection

Seeing that one can detect publicly-exposed sandboxes, we wondered if we could use the classifier trained on public knowledge to evade closed malware analysis appliances. Appliances are different from sandboxes in that their main objective is not to *analyze* the complete behavior of malware, but rather to *detect* malware in order to protect a sensitive infrastructure against cyber attacks. An advanced attacker may thus have strong incentives to detect an appliance in order to fly under the radar. That is, if an attacker can detect an appliance, she could hide her program's malicious behavior to avoid triggering any alert in the appliance.

Looking at the feature selection in Section 5, we realized that we can possibly assume that appliances could share the same feature characteristics as sandboxes. To verify this, we run SandPrint on three popular appliances from well-known vendors[4]. For this purpose, we gained access to various instances (Windows 7, Windows XP, 32/64 bit, different service packs, etc.) of the appliances. We ran SandPrint four times on each instance and collected 40 reports. Obtaining the features was not as trivial as in the case of publicly available sandboxes, since the appliances did not allow for network communication. To overcome this issue, we encoded the extracted features in the analysis report which was produced by the appliance after executing SandPrint.

When manually inspecting the feature reports, we found out that our assumption about the feature characteristics was correct. Similar to sandboxes, appliances also exhibit hardware, history and execution characteristics that indicate non-human and non-interactive usage. To our surprise, some features were even stronger than in the sandbox case. For example, all 40 reports contained a small screen width and a 4:3 screen resolution.

For each appliance, we then measure how accurately the classifier that we trained on the sandboxes and user report performs on the appliance reports. With an accuracy of 100%, the classifier detected all appliances as non-user machines. However, the main priority in this setting is not evasion *per se*, but rather *stealth evasion*. That is, while an attacker aims to detect an appliance, she does not want her detection method to be unveiled. We thus had a look at the reports produced by the appliances and found out that SandPrint created many security alerts by reading information such as motherboard information, BIOS information, or serial numbers. We then checked if the features used for the classifier were also on the list of alerts, which would essentially negate the stealthiness of the detection. For example, many Pafish checks were detected

[4] We omit the vendor names not to pinpoint to weaknesses of individual appliances.

by the appliances. Although the majority of the sandbox-inherent features did not trigger an alert by any appliance, we discovered that one appliance considers reading the disk information as suspicious behavior. To counter this, we removed the disk feature from the feature vector and evaluated the classifier again on the appliance reports, resulting again in 100% accuracy—even for stealth evasion.

To summarize, an attacker can reveal characteristics of publicly available sandboxes and use the gathered information to build a classifier that can perfectly distinguish between a user PC and an appliance. With insider knowledge on security appliances, an advanced attacker could tweak her classifier such that the evasion is stealthy and remains undetected by the appliance.

7 Discussion and Limitations

This section discusses ethical aspects and potential limitations of our work. As part of the ethical discussion, we also describe a responsible disclosure process in which we informed the sandbox and appliance operators about our findings.

7.1 Ethical Considerations

Our research may seem offensive in the sense that we reveal fingerprints of malware sandboxes that adversaries can use to evade them. Note, however, that the information we presented can be gathered by any other person reproducing our (conceptually simple) fingerprinting method. We thus consider the information shown in this paper as public knowledge. Still, we present data only in aggregated form and refrain from revealing any internals of particular sandboxes.

Using our insights, sandbox operators can aim to implement stealthier analysis systems. For example, we have shown that one should periodically update features that are inherent to the snapshot of a sandbox. While it will always be possible to find artifacts that can identify an individual sandbox, it is significantly harder to build a classifier that works for *all* sandboxes, especially if more people randomize characteristics. We have shown which features are particularly characteristic of sandboxes, giving sandbox operators hints on where to significantly improve the stealthiness of their systems.

7.2 Responsible Disclosure

Organizations developing sandboxes and/or appliances are immediately affected by our research results and we thus considered them as the main target of our responsible disclosure process. To notify these organizations, we contacted them 90 days prior to the publishing date of this paper, detailing the proposed attack and including hints on how to protect against potential adversaries in the future. We used direct contacts whenever possible and available. Alternatively, we resorted to contact details stated on the organization's websites, notably including Web-based contact forms. If we did not receive a response after 2 weeks, we retried to contact the organization, if possible using alternative communication channels (e.g., using generic email addresses like `info@organization.com` or email addresses found in the WHOIS database for the organization's website domain). If we did not hear back from the organization after 4 weeks, we contacted the national CERT(s) that are in the same country as the affected organization in order to notify the party via the CERT as trusted intermediary.

We handed to each organization an executive summary of our research results as well as a full description of our research methodology (i.e., a copy of this paper in the pre-print version). We made sure to highlight the implications of our work with respect to future operations of the sandbox and/or appliance. We also specified our contact details of both research institutions, including physical address, phone number, and the email address of a representative for the research activities. We allowed the organizations to download the latest version of SandPrint and its source code. Such auxiliary data is helpful to build protection mechanisms against sandbox-evasive programs similar to SandPrint. We also remove all organizations' names when referring to individual sandboxes/services.

7.3 Isolated Sandboxes

Most sandboxes allowed the program under analysis to communicate over the Internet, whereas nine services and all three appliances did not do so. To some extent we could also extract features of isolated sandboxes (the appliances) by encoding the features into events of the analysis report. However, this requires access to the isolated sandboxes, which may be hard to obtain for an attacker. Note that our sandbox classification did not use features that depend on the network configuration. In principle, our classification results should also generalize to non-connected sandboxes. Although we cannot rule out the possibility that there are non-connected sandboxes for which our classifier would perform poorly, we argue that the successful detection of appliances supports this claim.

Due to our assumption of Internet-connected sandboxes, the number of in-the-wild sandboxes is likely higher than our findings in the clustering results suggest. We argue, though, that our analyses are based on a statistically significant set of sandboxes, including those of the most popular analysis services.

8 Related Work

Evasion Techniques: Seeing the increasing popularity of sandboxes, malware authors try to find a way to evade sandbox analysis. Egele et al. give an overview of sandbox implementations [24]. Most sandboxes use virtual machine (VM) technology or CPU emulators. Such virtualization eases the process of analyzing multiple samples in parallel. Accordingly, studies show how to distinguish between a real machine and virtual environment. RedPill [47] determines whether it is executed on VMware using the `sidt` instruction. Many other detection methods have also been developed for not only VMware [43,29], but also for famous system emulators such as QEMU [43,40,22,29] and BOCHS [40,34]. There are also some detection methods for emulation-based Android sandboxes [54,42,27].

The fundamental difference between our approach and the above techniques is that we do not aim to detect virtualization or emulation, as VMs and sandboxes are not equivalent. In addition, as shown with PAFISH, most of these checks are not stealthy, whereas our approach even managed to detect security appliances without triggering alerts. It is also likely that our approach could work for bare metal sandboxes. We argue that bare metal sandboxes conceptually share many sandbox-inherent features with traditional sandboxes as the major difference is only the absence of virtualization and emulation—not the snapshot mechanism.

The work closest to our approach has been done by Maier et al. [33]. They gathered several features about Android sandboxes and showed that Android malware can bypass the existing sandboxes by using the fingerprints. However, they do neither perform automated clustering and classification of *sandbox-inherent* features, nor do they test their approach against security appliances. Furthermore, the feature selection of Maier et al. is very specific to smartphones. Features such as "the device needs at least n saved WiFi-networks" or "the device must have a paired Bluetooth device" cannot be used in our (non-mobile) context in a meaningful way. However, we also use some similar features like special hardware artifacts or system uptime. Regarding sandboxes for Windows malware, Yoshioka et al. [56] clustered and detected sandboxes by their external IP addresses. We were inspired by these works and performed a study in greater detail, collecting 25 features and identifying 76 sandboxes with an unsupervised machine learning technique.

Transparent Sandboxes: Seeing the threat of VM evasion, researchers started to explore transparent sandboxes that are stealthy against detection. Vasudevan et al. proposed Cobra [53], which was the first analysis system countering anti-analysis techniques. Dinaburg et al. proposed Ether [23], a transparent sandbox using hardware virtualization extensions such as Intel VT. Those systems focus on how to conceal the existence of analysis mechanisms from malware. Pek et al. introduced a timing-based detection mechanism to detect Ether [41]. In addition, as we have shown, the majority of sandboxes, including VT-based sandboxes, are susceptible to evasion due to sandbox-inherent features.

Kirat et al. proposed to use actual hardware to analyze malware [31,9]. The proposed system, called BareBox, is based on a fast and rebootless system restore technique. Since the system executes malware on real hardware, it is not vulnerable to any type of VM/emulation-based detection attacks. Still, as it is snapshot-based, it falls for the methods described in Section 5.

9 Conclusion

Our real-world malware sandbox investigations have shown it is quite straightforward to fingerprint malware sandboxes. We identified 76 sandboxes by uploading a measurement binary to 20 services, all of which can be rather trivially detected and evaded just based on sandbox-inherent characteristics. Our findings also suggest detecting and evading malware appliances is similarly possible. This calls into question how we can protect against the threat of sandbox evasion in the future, and should serve as a heads-up for sandbox operators to inform them about threats that may actually be already silently misused by malware.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable comments. Special thanks goes to our shepherd Michael Bailey, who supported us during the process of finalizing the paper. This work was supported by the MEXT Program for Promoting Reform of National Universities and by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and for the BMBF project 13N13250.

References

1. Amnpardaz Sandbox - File Analyzer. `http://jevereg.amnpardaz.com/`.
2. Anubis: Malware Analysis for Unknown Binaries. `https://anubis.iseclab.org/`.
3. Bkav - Scan virus online. `http://quetvirus.vn/default.aspx?lang=en`.
4. bochs: The Open Source IA-32 Emulation Project. `http://bochs.sourceforge.net`.
5. Dr.Web Online Check. `http://online.drweb.com/?lng=en`.
6. FortiGuard Center. Online Virus Scanner. `http://www.fortiguard.com/virusscanner`.
7. Gary‘s Hood. Online Virus Scanner. `http://www.garyshood.com/virus/`.
8. Malwr - Malware Analysis by Cuckoo Sandbox. `https://malwr.com/`.
9. NVMTrace: Proof-of-concept Automated Baremetal Malware Analysis Framework. `https://code.google.com/p/nvmtrace/`.
10. Oracle VM VirtualBox. `https://www.virtualbox.org`.
11. #totalhash. `https://totalhash.cymru.com/upload/`.
12. Vicheck.ca.
13. Virusblokada. `http://anti-virus.by/en/index.shtml`.
14. VirusTotal - Free Online Virus, Malware and URL Scanner. `https://www.virustotal.com/en/`.
15. VMware. `http://www.vmware.com/`.
16. Bayer, U., Milani Comparetti, P., Hlauschek, C., Kruegel, C., and Kirda, E. Scalable, Behavior-based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)* (2009).
17. Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), ATEC ’05.
18. Brengel, M., Backes, M., and Rossow, C. Detecting Hardware-Assisted Virtualization. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).
19. Caballero, J., Grier, C., Kreibich, C., and Paxson, V. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *USENIX Security* (2011).
20. Comodo. Comodo Instant Malware Analysis. `http://camas.comodo.com/`.
21. Cristianini, N., and Shawe-Taylor, J. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods.* Cambridge University Press, 2000.

22. DEXLabs. Detecting Android Sandboxes. `http://www.dexlabs.org/blog/btdetect`, 2012.
23. Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), CCS'08.
24. Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv. 44*, 2 (2008).
25. F-Secure. Sample Analysis System. `https://analysis.f-secure.com/portal/login.html`.
26. Freiling, F., Holz, T., and Wicherski, G. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symposium on Research in Computer Security (ESORICS)* (2005).
27. Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACSAC '14.
28. Jotti. Jotti's Malware Scan. `http://virusscan.jotti.org/en`.
29. Jung, P. Bypassing Sandboxes for Fun. `https://www.botconf.eu/wp-content/uploads/2014/12/2014-2.7-Bypassing-Sandboxes-for-Fun.pdf`.
30. Kirat, D., Vigna, G., and Kruegel, C. Barecloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014), SEC'14.
31. Kirati, D., Vigna, G., and Kruegel, C. BareBox: Efficient Malware Analysis on Bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACSAC'11.
32. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. AccessMiner: Using System-centric Models for Malware Protection. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), CCS '10.
33. Maier, D., Müller, T., and Protsenko, M. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security* (2014), ARES '14.
34. Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (2009), ISSTA'09.
35. Microsoft. Submit a sample - Microsoft Malware Protection Center. `https://www.microsoft.com/security/portal/submission/submit.aspx`.
36. Neugschwandtner, M., Comparetti, P. M., and Platzer, C. Detecting Malware's Failover C&C Strategies with Squeeze. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACSAC '11.
37. Neuner, S., van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., and Weippl, E. Enter Sandbox: Android Sandbox Comparison. `http://arxiv.org/ftp/arxiv/papers/1410/1410.7749.pdf`, 2015.
38. OPSWAT. Metascan Online: Free File Scanning with Multiple Antivirus Engines. `https://www.metascan-online.com/#!/scan-file`.
39. Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., and Rossow, C. IoTPOT: Analysing the Rise of IoT Compromises. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies* (2015), WOOT.
40. Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators.

In *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (2009), WOOT'09.

41. Pék, G., Bencsáth, B., and Buttyán, L. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proceedings of the Fourth European Workshop on System Security* (2011), EUROSEC'11.
42. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security* (2014), EuroSec '14.
43. Raffetseder, T., Kruegel, C., and Kirda, E. Detecting System Emulators. In *Proceedings of the 10th International Conference on Information Security* (2007), ISC'07.
44. Rieck, K., Schwenk, G., Limmer, T., Holz, T., and Laskov, P. Botzilla: Detecting the Phoning Home of Malicious Software. In *proceedings of the 2010 ACM Symposium on Applied Computing (ACSAC '10)* (2010).
45. Rieck, K., Trinius, P., Willems, C., and Holz, T. *Automatic Analysis of Malware Behavior Using Machine Learning.* Journal of Computer Security, 2009.
46. Rossow, C., Dietrich, C. J., and Bos, H. Large-Scale Analysis of Malware Downloaders . In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '12)* (July 2012).
47. Rutkowska, J. Red Pill... Or How To Detect VMM Using (Almost) One CPU Instruction. `http://www.securiteam.com/securityreviews/6Z00H20BQS.html`, 2004.
48. Security, P. Free Automated Malware Analysis Service. `https://www.hyblid-analysis.com/`.
49. Security, P. Blog article. `http://www.pandasecurity.com/mediacenter/press-releases/pandalabs-neutralized-75-million-new-malware-samples-2014-twice-many-2013/`, 2015.
50. Security, T. Free Online Malware Analysis. `http://www.threattracksecurity.com/resources/sandbox-malware-analysis.aspx`.
51. Symantec. Internet Security Threat Report 04/2015. `http://www.symantec.com/de/de/security_response/publications/threatreport.jsp`, 2015.
52. ThreatExpert. Submit Your Sample Online. `http://www.threatexpert.com/submit.aspx`.
53. Vasudevan, A., and Yerraballi, R. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006), S&P'06.
54. Vidas, T., and Christin, N. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (2014), ASIA CCS '14.
55. VirSCAN.org. Free Multi-Engine Online Virus Scanner. `http://www.virscan.org/`.
56. Yoshioka, K., Hosobuchi, Y., Orii, T., and Matsumoto, T. Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. *Journal of Information Processing 52*, 3 (2011).

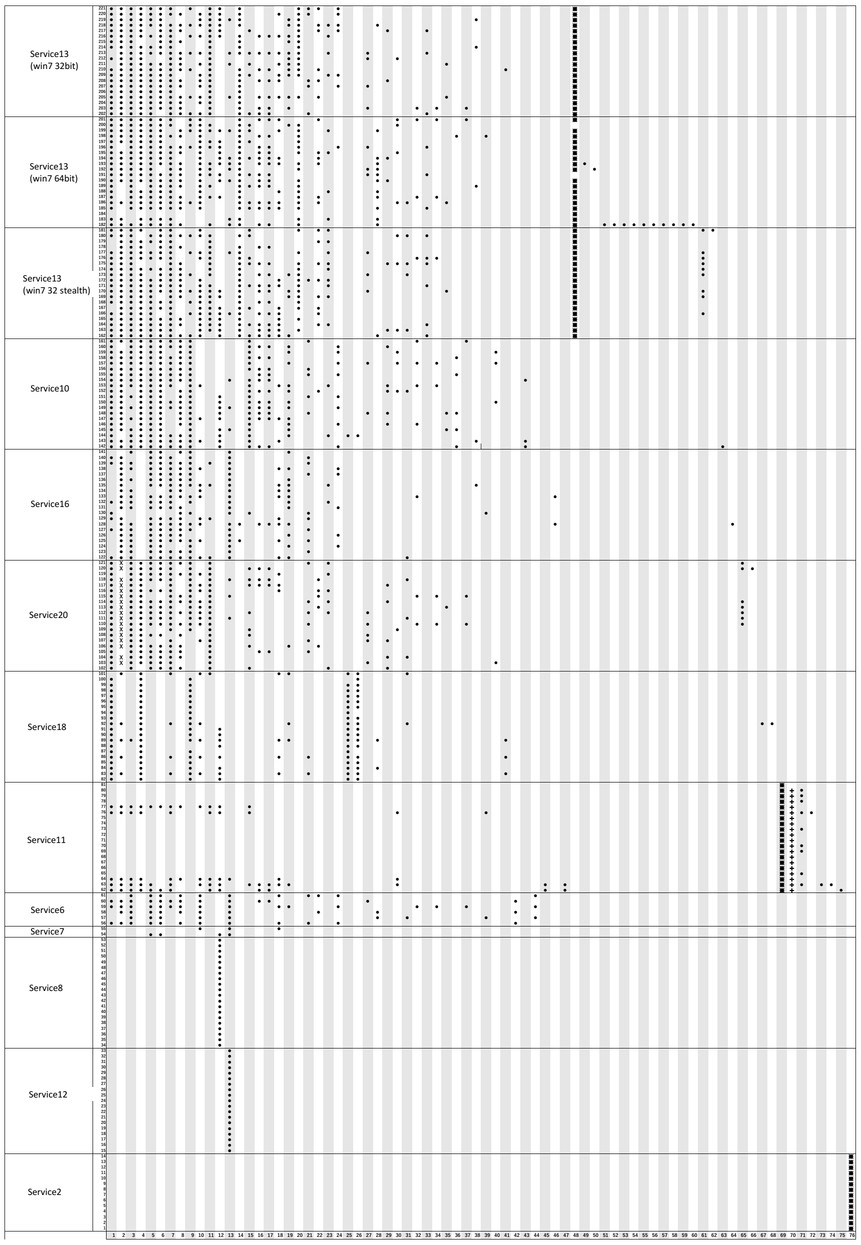

Fig. 3: Mapping between submitted SandPrint instances and sandboxes. The non-circle shapes indicate constant and exclusive use of a sandbox by a particular service and thus are inferred as being a sandbox attached to the service. A cross indicates that the mapping is confirmed by mapping the SandPrint report to the dynamic analysis report provided by the service.