Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts

Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, Michalis Polychronakis
Stony Brook University
{nmiramirkhani, mappini, nick, mikepo}@cs.stonybrook.edu

***Abstract*—Malware sandboxes, widely used by antivirus companies, mobile application marketplaces, threat detection appliances, and security researchers, face the challenge of environment-aware malware that alters its behavior once it detects that it is being executed on an analysis environment. Recent efforts attempt to deal with this problem mostly by ensuring that well-known properties of analysis environments are replaced with realistic values, and that any instrumentation artifacts remain hidden. For sandboxes implemented using virtual machines, this can be achieved by scrubbing vendor-specific drivers, processes, BIOS versions, and other VM-revealing indicators, while more sophisticated sandboxes move away from emulation-based and virtualization-based systems towards bare-metal hosts.**

We observe that as the fidelity and transparency of dynamic malware analysis systems improves, malware authors can resort to other system characteristics that are indicative of artificial environments. We present a novel class of sandbox evasion techniques that exploit the "wear and tear" that inevitably occurs on real systems as a result of normal use. By moving beyond how realistic a system looks like, to how realistic *its past use* looks like, malware can effectively evade even sandboxes that do not expose any instrumentation indicators, including bare-metal systems. We investigate the feasibility of this evasion strategy by conducting a large-scale study of wear-and-tear artifacts collected from real user devices and publicly available malware analysis services. The results of our evaluation are alarming: using simple decision trees derived from the analyzed data, malware can determine that a system is an artificial environment and not a real user device with an accuracy of 92.86%. As a step towards defending against wear-and-tear malware evasion, we develop statistical models that capture a system's age and degree of use, which can be used to aid sandbox operators in creating system images that exhibit a realistic wear-and-tear state.

I. Introduction

As the number and sophistication of the malware samples and malicious web pages that must be analyzed every day constantly increases, defenders rely on automated analysis systems for detection and forensic purposes. Malware scanning based on static code analysis faces significant challenges due to the prevalent use of packing, polymorphism, and other code obfuscation techniques. Consequently, malware analysts largely rely on dynamic analysis approaches to scan potentially malicious executables. Dynamic malicious code analysis systems operate by loading each sample into a heavily instrumented environment, known as a *sandbox*, and monitoring its operations at varying levels of granularity (e.g., I/O activity, system calls, machine instructions). Malware sandboxes are typically built using API hooking mechanisms [1], CPU emulators [2]–[5], virtual machines [6], [7], or even dedicated bare-metal hosts [8]–[10].

Malware sandboxes are employed by a broad range of systems and services. Besides their extensive use by antivirus companies for analyzing malware samples gathered by deployed antivirus software or by voluntary submissions from users, sandboxes have also become critical in the mobile ecosystem, for scrutinizing developers' app submissions to online app marketplaces [11], [12]. In addition, a wide variety of industry security solutions, including intrusion detection systems, secure gateways, and other perimeter security appliances, employ sandboxes (also known as "detonation boxes") for on-the-fly or off-line analysis of unknown binary downloads and email attachments [13]–[17].

As dynamic malware analysis systems become more widely used, online miscreants constantly seek new methods to hinder analysis and evade detection by altering the behavior of malicious code when it runs on a sandbox [8], [9], [11], [18]–[20]. For example, malicious software can simply crash or refrain from performing any malicious actions, or even exhibit benign behavior by masquerading as a legitimate application. Similarly, exploit kits hosted on malicious (or infected) web pages have started employing evasion and cloaking techniques to refrain from launching exploits against analysis systems [21], [22]. To effectively alter its behavior, a crucial capability for malicious code is to be able to recognize whether it is being run on a sandbox environment or on a real user system. Such "environment-aware" malware has been evolving for years, using sophisticated techniques against the increasing fidelity of malware analysis systems.

In this paper, we argue that as the fidelity and non-intrusiveness of dynamic malware analysis systems continues to improve, an anticipated next step for attackers is to rely on other environmental features and characteristics to distinguish between end-user systems and analysis environments. Specifically, we present an alternative, more potent approach to current VM-detection and evasion techniques that is effective *even if we assume that no instrumentation or introspection artifacts of an analysis system can be identified by malware at run time*. This can be achieved by focusing instead on the *"wear and tear"* and *"aging"* that inevitably occurs on real systems as a result of normal use. Instead of trying to identify artifacts characteristic to analysis environments, malware can determine the plausibility of its host being a real system used by an actual person based on system usage indicators.

The key intuition behind the proposed strategy is that, in contrast to real devices under active use, existing dynamic analysis systems are typically implemented using operating system images in almost pristine condition. Although some further configuration is often performed to make the system look more realistic, after each binary analysis, the system is rolled back to its initial state [23]. This means that any possible wear-and-tear side-effects are discarded, and the system is always brought back to a pristine condition. As an indicative example, in a real system, properties like the browsing history, the event log, and the DNS resolver cache, are expected to be populated as a result of days' or weeks' worth of activity. In contrast, due to the rollback nature of existing dynamic analysis systems, malicious code can easily identify that the above aspects of a system may seem unrealistic for a device that supposedly has been in active use by an actual person.

Even though researchers have already discussed the possibility of using specific indicators, such as the absence of "Recently Open Files" [24] and the lack of a sufficient number of processes [25] as ways of identifying sandbox environments, we show that these individual techniques are part of a larger problem, and systematically assess their threat to dynamic analysis sandboxes. To assess the feasibility and effectiveness of this evasion strategy, we have identified a multitude of wear-and-tear artifacts that can potentially be used by malware as indicators of the extent to which a system has been actually used. We present a taxonomy of our findings, based on which we then proceeded to implement a probe tool that relies on a subset of those artifacts—ones that can be captured in a privacy-preserving way—to determine a system's level of wear and tear. Using that tool, we gathered a large set of measurements from 270 real user systems, and used it to determine the discriminatory capacity of each artifact for evasion purposes.

We then proceed to test our hypothesis that wear and tear can be a robust indicator for environment-aware malware, by gauging the extent to which malware sandboxes are vulnerable to evasion using this strategy. We submitted our probe executable to publicly available malware scanning services, receiving results from 16 vendors. Using half of the collected sandbox data, we trained a decision tree model that picks a few artifacts to reason whether a system is real or not, and evaluated it using the other half. Our findings are alarming: for all tested sandboxes, the employed model can determine that the system is an artificial environment and not a real user device with an accuracy of 92.86%.

We provide a detailed analysis of the robustness of this evasion approach using different sets of artifacts, and show that modifying existing sandboxes to withstand this type of evasion is not as simple as artificially adjusting the probed artifacts to plausible values. Besides the fact that malware can easily switch to the use of another set of artifacts, we have uncovered a deeper direct correlation of many of the evaluated wear-and-tear artifacts and the reported age of the system, which can allow malware to detect unrealistic configurations. As a step towards defending against wear-and-tear malware evasion, we have developed statistical models that capture the age and degree of usage of a given system, which can be used to fine-tune sandboxes so that they look realistic.

In summary, our work makes the following main contributions:

- We show that certain, seemingly independent, techniques for identifying sandbox environments are in fact all instances of a broader evasion strategy based on a system's wear-and-tear characteristics.
- We present an indicative set of diverse wear-and-tear artifacts, and evaluate their discriminatory capacity by conducting a large-scale measurement study based on data collected from real user systems and malware sandboxes.
- We demonstrate that using simple decision trees derived from the analyzed data, malware can robustly evade *all* tested sandboxes with an accuracy of 92.86%.
- We present statistical models that can predict the age of a system based on its system usage indicators, which can be used to "age" existing artificial-looking sandboxes to the desired degree.

II. Background and Related Work

A. Virtualization and Instrumentation Artifacts

Antivirus companies, search engines, mobile application marketplaces, and the security research community in general largely rely on dynamic code analysis systems that load potentially malicious content in a controlled environment for analysis purposes [1]–[6], [8]–[10], [21], [26]–[34]. Given that virtual machines and system emulators are very convenient platforms for building malware sandboxes, evasive malware often relies on various artifacts of such systems to alter its behavior [8], [9], [11], [18]–[20], [35]. Probably the simplest approach is to use static heuristics that check for certain system properties, such as VM-specific device drivers and hardware configurations, fixed identifiers including MAC addresses, IMEIs (for mobile malware analysis systems), user/host names, VM-specific loaded modules and processes (e.g., VMware Tools in case of VMware), and registry entries [20], [36].

Even when dynamic analysis systems are carefully configured to avoid expected values and configurations, lower-level properties of the execution environment can be used to identify the presence of a virtual environment. These include emulator intricacies that can be observed at runtime using small code snippets, timing properties of certain virtualized instructions, cache side effects, and many others [8], [20], [37]–[43].

B. Environmental and User Interaction Artifacts

The increasing sophistication of environment-aware malware has prompted many research efforts that focus on either detecting the "split personality" of malware by running each sample on multiple and diverse analysis systems [18], [44], [45], or increasing the fidelity of the analysis environment by avoiding the use of emulation or virtualization altogether, and opting for "bare-metal" systems [8]–[10], [46].

These recent developments, however, along with the proliferation of virtual environments in production deployments and end-user systems, have resulted in yet another change of

tactics from the side of malware authors. Given that a VM host may be equally "valuable" as a non-VM host, the number of malicious samples that rely on VM detection tricks for evasion has started to decrease [47].

At the same time, malware authors have started employing other heuristics that focus mostly on how realistic the state and configuration of a host is, rather than whether it is a real or a virtual one. Such heuristics include checking whether the mouse cursor remains motionless in the center of the screen [48], observing the absence of "Recently Open Files" [24] or an unusually low number of processes [25], testing whether unrestricted internet connectivity is available, and trying to resolve a known non-existent domain name. As some malware analysis systems blindly serve all DNS requests, a positive answer to a request for a non-existent domain is an indication of a malware analysis environment and not a real system [9]. Correspondingly, malware sandboxes have started emulating user behavior (e.g., by generating realistic-looking mouse movements and actions) and exposing a more realistic network environment. Unfortunately, as we show in this work, a neglected aspect of this effort towards making malware sandboxes as realistic as possible, is the wear and tear that is expected to occur as a result of normal use.

C. Sandbox Fingerprinting

A few research efforts have focused on deriving configuration profiles from sandboxes used by malware scanning services and security appliances [23], [49]. By fingerprinting the sandboxes of different vendors, malware can then identify the distinguishing characteristics of a given sandbox, and alter its behavior accordingly. Maier et al. developed Sand-Finger [49], a sandbox fingerprinting tool for Android-based analysis systems. The extracted fingerprints capture properties that are mostly specific to mobile devices, such as saved WiFi networks, paired Bluetooth devices, or whether a device is connected to a host via a cable. The authors show that the derived fingerprints can be used by malicious apps to evade Google's Bouncer and other mobile app sandboxes.

Recently, Blackthorne et al. presented AVLeak, a tool that can fingerprint emulators running inside commercial antivirus (AV) software, which are used whenever AVs detect an unknown executable [50]. The authors developed a method that allows them to treat these emulators as black boxes and use side channels as a means of extracting fingerprints from each AV engine.

Yokoyama et al. developed SandPrint [23], a sandbox fingerprinting tool tailored for Windows malware analysis systems. The tool considers a set of 25 features that were identified to exhibit characteristic, high-entropy values on malware sandboxes used by different vendors. These include hardware configuration parameters, which tend to have unusually low values to facilitate the parallel operation of multiple sandbox images (e.g., by lowering the amount of RAM or disk space), or features specific to the malware sample invocation mechanism employed by each sandbox (e.g., Cuckoo Sandbox's `agent.py` launch script, or the fact that the file name of the analyzed executable is changed according to its MD5 sum or using some other naming scheme). The authors then performed automated clustering to classify fingerprints collected after the submission of the tool to 20 malware analysis services, resulting in the observation of 76 unique sandboxes. They also used some of the features to build a classifier that malware can use to reliably evade the tested sandboxes.

As the authors of SandPrint note, *"most of the selected features are deterministic and their values discrete and reliable,"* and *"a stealthy sandbox could try to diversify the feature values."* Inspired by this observation, our aim in this paper is to explore what the next step of attackers might be, once sandbox operators begin to diversify the values of the above features, and to configure their systems using more realistic settings, rendering sandbox-fingerprinting ineffective. To that end, instead of focusing on features characteristic to sandboxes, we take a radically different view and focus on artifacts characteristic to real, *used* machines. Previous works have shown that sandboxes are configured differently than real systems, and can thus be easily fingerprinted. The programs used to analyze malware are designed to be as transparent as possible, to prevent malware from detecting that it is being analyzed. In this work, we show that even with completely transparent analysis programs, the environment outside the analysis program is enough for malware to determine that it is under analysis.

III. Wear and Tear Artifacts

To determine the extent to which a system has been actually used, and consequently, the plausibility of it being a real user system and not a malware sandbox, malicious code can rely on a broad range of artifacts that are indicative of the wear and tear of the system. In this section, we discuss an indicative set of the main types of such artifacts that we have identified, which can be easily probed by malware.

Our strategy for selecting these artifacts was based on identifying what aspects of a system are affected as a result of normal use, and *not* on system features that may be affected by certain sandbox-specific implementation intricacies—a realistically configured sandbox may not exhibit any such features at all. We located these artifacts by using a method similar to the snowball method for conducting literature surveys. Specifically, upon identifying a potentially promising artifact, we would search in that same location of the system for more artifacts. Besides intuition, to identify additional sources of artifacts, we also studied various Windows "cleaner" and "optimizer" programs, as well as Windows forensics literature.

Our list of artifacts is clearly non-comprehensive and we are certain that there are many more good sources of artifacts that were not included in this study. At the same time, although it is expected that one can find a multitude of wear-and-tear artifacts, e.g., as a result of the customization, personalization, and use of different applications on an actual user system, our selection was constrained by an important additional requirement.

A. Probing for Artifacts while Preserving User Privacy

We can distinguish between two main types of wear-and-tear artifacts, depending on their source: artifacts that stem from *direct* user actions, versus artifacts caused by *indirect* system activity. For instance, a browser's history contains the URLs that a user has explicitly visited. Expecting to find a popular search engine, a social networking service, and a news website among the recently visited URLs could constitute a heuristic that captures a user's specific interests. In contrast, the number of application events in the system event log is an indirect artifact which, although it depends on user activity, it does not capture any private user information.

Direct artifacts are qualitatively stronger indicators of a system's degree of use by a person. Conceivably, malware could rely on a broad and diverse range of such artifacts to decide whether a system has been under active use, e.g., by inspecting whether document folders contain plausible numbers of files with expected file extensions, checking for recently opened documents in popular document viewing applications, or looking for the most recently typed online search engine queries, system-wide ("spotlight-like") search queries, instant messaging or email message contents, and so on.

For the purposes of our study, however, such artifacts are out of question, as collecting them from real user systems would constitute a severe privacy violation. Consequently, we limit our study mainly to indirect, system-wide artifacts, which do not reveal any private user information. To get a glimpse on the evasion potential of direct artifacts, we do consider a limited set of browser-specific artifacts, which though are strictly limited to aggregate counts (e.g., number of cookies, number of downloaded files and, number of visited URLs) that can be collected without compromising user privacy and anonymity.

B. Artifact Categories

Many aspects of a system are susceptible to wear and tear, from the operating system itself, to the various file, network, and other subsystems. Considering, for instance, a typical Microsoft Windows host, a multitude of different wear-and-tear artifacts are available for estimating the extent to which the system has been used. In the following, we provide a brief description of the main types of artifacts that we have identified. The complete set of artifacts that we used in our experiments is shown in Table I.

1) System: Generic system properties such as the number of running processes or installed updates are directly related to the "history" of a system, as more installed software typically accumulates throughout the years.

The event log of Windows systems is another rich source of information about the current state and past activity of the system. Various types of events are recorded into the event log, including application, security, setup, and system events of different administrative ratings (critical, error, warning, information, audit success or failure). Of particular interest are events that denote abnormal conditions, such as warnings about missing files, inaccessible network links, unexpected service terminations, and access permission errors. The more a system has been used, the higher the number of records that will have been accumulated in the event log. Besides the simple count of system events, we also consider additional aspects indicative of past use, such as the number of events from user applications, the number of different event sources, and the time difference between the first and last event.

The event log will be (almost) empty only if the operating system is freshly installed, or if the user deleted the recorded entries. The vast majority of users, however, are not even aware of the existence of this log, so it is unlikely that its contents will be affected (e.g., deleted) by explicit user actions.

2) Disk: User activity results in file system changes, exhibited in various forms, including the generation of temporary files, deleted files, actual user content, cached data, and so on. Given the sensitive nature of accessing user-generated content, even in terms of just counting files, we limit the collection of disk artifacts directly related to user activity only to the number of files on the desktop and in the recycle bin. The rest of the disk artifacts capture counts of files related to non-user-generated temporary or cached data (e.g., generated file-content thumbnails, or process crash "minidump" files).

3) Network: The network activity of a system, especially from a historical perspective, is a strong indicator of actual use. Every time a host is about to send a packet to a remote destination, it consults the system's address resolution protocol (ARP) cache to find the physical (MAC) address of the gateway, or the address of another host in the local network. Similarly, every time a domain name is resolved to an IP address, the operating system's DNS stub resolver includes it in a cache of the most recent resolutions. The use of public key encryption also results in artifacts related to the certificates that have been encountered. For instance, once a certificate revocation list (CRL) is downloaded, it is cached locally. The list of the URLs of previously downloaded CRLs is kept as part of the cached information, so the number of entries in that list is directly related to past network activity (not only due to browsing activity, but also due to applications that perform automatic updates over the network). We also consider the number of cached wireless network SSIDs, which is a good indicator of past use for mobile devices, and the number of active TCP connections.

4) Registry: The Windows registry contains a vast amount of information about many aspects of a system, including many that would fit in the aforementioned categories. We chose, however, to consider the Registry separately both because it is privy to information that does not exist elsewhere, as well as because, in Section V, we investigate whether sandbox-detection is possible if an attacker has to constrain himself to a single type of wear-and-tear artifacts.

TABLE I: Complete list of wear-and-tear artifacts.

Category	Name	Description	User	Sandbox	Baseline
System	totalProcesses	# of processes	94.4	35	41
	winupdt	# installed Windows updates	794	19	2
	sysevt	# system of system events	27K	8K	334
	appevt	# of application events	18K	2.4K	184
	syssrc	# sources of system events	78	49	48
	appsrc	# sources of application events	40	26	23
	sysdiffdays	Elapsed time since the first system event (days)	370	1.7K	0
	appdiffdays	Elapsed time since the first application event (days)	298	943	0
Disk	recycleBinSize	Total size of the recycle bin (bytes)	2.5G	50M	0
	recycleBinCount	# files in the recycle bin	109	3	0
	tempFilesSize	Total size of temporary system files (bytes)	302	24	8.2
	tempFilesCount	# temporary system files	411	60	10
	miniDumpSize	Total size of process crash minidump files (bytes)	3M	409K	0
	miniDumpCount	# of process crash minidump files	9	4	0
	thumbsFolderSize	Total size of the system's thumbnails folder (bytes)	63M	8M	2.6M
	desktopFileCount	# files on the desktop	34	6	3
Network	ARPCacheEntries	# entries in the ARP cache	19	4.5	5
	dnscacheEntries	# entries in the DNS resolver cache	151	4	3
	certUtilEntries	# URLs of previously downloaded CRLs	1.7K	210	6
	wirelessnetCount	# of cached wireless SSIDs	8	0	0
	tcpConnections	# of active TCP connections	77	27	16
Registry	regSize	Size of the registry (in bytes)	144.8M	53M	35M
	uninstallCount	# registered software uninstallers	177	58	18
	autoRunCount	# programs that automatically run at system startup	9	3	1
	totalSharedDlls	Legacy DLL reference count	242	176	26
	totalAppPaths	# registered application paths	54	35	23
	totalActiveSetup	# Active Setup application entries	24	32	25
	orphanedCount	# leftover registry entries	11	10	9
	totalMissingDlls	# registered DLLs that do not exist on disk	21	23	13
	usrassistCount	# of entries in the UserAssist cache (frequently opened applications)	222	98	11
	shimCacheCount	# entries in the Application Compatibility Infrastructure (Shim) cache	191K	98K	38K
	MUICacheEntries	# Multi User Interface (MUI) cache entries	66	83	163
	FireruleCount	# of rules in the Windows Firewall	511	332	358
	deviceClsCount	# previously connected USB devices (DeviceInstance IDs)	81	29	60
	USBStorCount	# previously connected USB storage devices	1.7	0	0
Browser	browserNum	# installed browsers (Internet Explorer, Firefox, Chrome)	2.9	1.4	1
	uniqueURLs	# unique visited URLs	28K	13.8	1.64
	totalTypedURLs	# URLs typed in the browser's navigation bar	1.6K	4	1
	totalCookies	# of HTTP cookies	3K	135	2
	uniqueCookieDomains	# unique HTTP cookie domains	1003	23	1
	totalBookmarks	# bookmarks	274	20	1
	totalDownloadedFiles	# downloaded files	340	3	0
	urlDiffDays	Time elapsed between the oldest and newest visited URL (days)	225	370	0
	cookieDiffDays	Time elapsed between the oldest and newest HTTP cookie (days)	303	256	0

Every time an application or driver is installed on a Windows system, it typically stores many key-value pairs in the registry. This effect is so extensive that Windows is known from suffering from "registry bloat," since programs often add keys during installation, but neglect to remove them when uninstalled. This is exemplified by the multitude of "registry cleaning" tools which aim to clean the registry from stale entries [51], [52]. Many of the registry artifacts we consider are thus related to the footprints of the applications that have been installed on the system (e.g., number of installed applications, uninstall entries, shared DLLs, applications set to automatically run upon boot), as well those that have not been fully uninstalled (e.g., "orphaned" entries left from removed programs and listed DLLs that do not exist on disk).

We also consider various other system-wide properties, including: the size of the registry (in bytes); the number of firewall rules, as new applications often install additional rules; the number of previously connected USB removable storage devices (a unique instance ID key is generated for each device and stored in a cache); the number of entries in the UserAssist cache, which contains information about the most frequently opened applications and paths; the number of entries in the

Application Compatibility Infrastructure (Shim) cache, which contains metadata of previously run executables that used this facility; and the number of entries in the MUICache, which are related to previously run executables and are generated by the Multi User Interface that provides support for different languages.

5) Browser: For many users, an operating system does little more than housing their web browser and a few other applications. Web browsers have become an all-inclusive tool for accessing office suites, games, e-shopping, e-banking, and social networking services. Due to the indispensable use of a browser, it would be really uncommon to find a browser in a pristine, out-of-the-box condition on a real user system.

In fact, the older a system is, the longer the accumulated "history" of its main browser will likely be. This history is captured in various artifacts related to past user activity, including previously visited URLs, stored HTTP cookies, saved bookmarks, and downloaded files. As noted earlier, we consider only aggregate counts of such artifacts, instead of more detailed data, to respect users' privacy. Given that a person may use more than one browser, or a different one than the operating system's default browser, the browser artifacts that we consider correspond to combined values across all browsers found on the system (our tool currently supports Internet Explorer, Firefox, and Chrome). For example, the number of HTTP cookies corresponds to the total number of cookies extracted from all installed browsers. The number of installed browsers is also considered as a standalone artifact.

IV. Data Collection

A. Probe Tool Implementation

To gather a large and representative dataset of the identified wear-and-tear artifacts from real user systems and malware sandboxes, we implemented a probe tool that collects artifact information and transmits it to a server. The probe tool was implemented in C++ and consists of a single Windows 32-bit PE file that does not require any installation, and which does not have any dependencies besides already available system APIs. Although many artifacts are easy to collect by simply accessing the appropriate log files and directories, probably the most challenging aspect of our implementation was to maximize compatibility with as many Windows versions as possible, from Windows XP to Windows 10. Besides differences in the paths of various system files and directories across major versions, in many cases we also had to use different system APIs for Windows XP compared to more recent Windows versions for various system-related artifacts (e.g., event log statistics). Browser artifacts are collected separately for each of the three supported browsers (IE, Firefox, Chrome) that are found on a given system, and are synthesized into compound artifacts at the server side.

The collected information is transmitted to our server through HTTPS. The tool uses both GET and POST requests, in case either of the two is blocked by a sandbox. Besides the collected data, additional metadata for each submission include the IP address of the host, OS version and installation date, and BIOS vendor. The latter is used as a simple VM-detection heuristic, to prevent poisoning of our real-user dataset with any results from virtual machines (we explicitly asked users to run the tool only on their real systems).

Each executable submitted to a public sandbox is generated with a unique embedded ID, which allows us to identify multiple submissions by the same vendor. Since a given unique instance of the tool may run on more than one sandboxes (e.g., due to the use of multiple different sandboxes by the same vendor, or due to sample sharing among vendors), we additionally distinguish between different systems based on the combination of the following keys: reported IP address, OS installation date, Windows version, BIOS vendor, number of installed user applications, and elapsed time since the first system event.

B. IRB Approval and User Involvement

Our experiments for the collection of artifact values from real user systems involved human subjects, and thus we had to obtain institutional review board (IRB) approval prior to conducting any such activity. Our IRB application included a detailed description of the information to be collected, the process a participant would follow, and all the measures that were taken to protect the participants' privacy and anonymity.

Our activities did not expose users to any risk. The collected data does not include any form of personally identifiable information (PII), nor any such information can be derived from it. Due to the nature of our experiment (collecting simple system statistics), our probe tool is small and simple enough so that we can be confident about the absence of any flaws or bugs that would cause damage or mere inconvenience to a user's system and data. The tool does not get installed on the system, and thus users can simply delete the downloaded executable to remove all traces of the experiment. The collected data merely reflects the wear and tear of a user's system as a result of normal use, rather than some personal trait that could be considered PII. Even when considering the full information that is recorded in the transmitted data, no personal information about the user of the system can be inferred.

Based on the above, the IRB committee of our institution approved the research activity on April 11, 2016.

All participants (colleagues, friends, and Amazon's Mechanical Turk workers) were directed to a web page that includes an overview of the experiment, a detailed description of the collected information (including a full sample report from one of the authors' laptop), and instructions for downloading and executing our probe tool (as well as deleting it afterwards). The page was hosted using our institution's second-level domain and TLS certificate, and included information about the IRB approval, as well as the authors' full contact details.

Given that our email requests to friends and colleagues for downloading and executing a binary from a web page could be considered as spear phishing attempts, we pointed to proof for the legitimacy of the message, which was hosted under the authors' institutional home pages.

TABLE II: Distribution of BIOS vendors of users' machines.

BIOS vendor	Frequency
American Megatrends Inc.	25.77%
Dell Inc.	18.08%
Insyde	15.38%
Lenovo	15.38%
Hewlett-Packard	6.92%
Award Software International Inc.	3.85%
Phoenix Technologies Ltd.	3.85%
Acer	1.92%
AMI	1.92%
Intel Corp.	1.92%
Alienware	1.54%
Toshiba	1.54%
Other	1.93%

C. Data Collection

To quantify the difference of wear-and-tear artifacts between real users and sandbox environments, we use the aforementioned probing tool to compile three datasets. The first dataset contains wear-and-tear artifacts collected from 270 real user machines (D_{real}). The majority of these observations (89.4%) originate from Amazon Mechanical Turk workers, who downloaded and executed our artifact-probing tool, and the remaining from systems operated by friends and colleagues. The users who participated in this study come from 35 different countries, with the top countries being: US (44%), India (18%), GB (10%), CA (8%), NL (1%), PK (1%), RU (1%) and 28 other countries with lower than 1% frequency. Therefore, we argue that our dataset is representative of users from both developed and developing countries, whose machines may have different wear-and-tear characteristics. The BIOS vendor distribution of the user systems is shown in Table II. As the table shows, the users' personal systems include different brands, such as Dell, HP, Lenovo, Toshiba, and Acer, as well as game series systems such as Alienware.

As mentioned earlier, our tool does not collect any PII and does not affect the user's machine in any way, i.e., it does not modify or leave any data. After running the tool, the user can simply delete the executable.

While analyzing the collected data from Mechanical Turk to filter out those users who, despite our instructions, executed our tool inside a virtual machine (identified through the collected BIOS information), we observed that our server had collected artifacts that could not be traced back to Mechanical Turk workers. By analyzing the artifacts and the logs of our server, we realized that, because the public Human Intelligence Task (HIT) web page for our task on Amazon's website was pointing to our tool download page through a link, the crawlers of a popular search engine followed that link, downloaded our probing tool, and executed it. Based on the IP address space of the clients who reported these artifacts, as well as custom identifiers found in the reported BIOS versions, we are confident that these were sandbox environments that are associated with a popular search engine that downloads and executes binaries as a way of proactively discovering malware. After removing duplicate entries, we marked the rest as belonging to "crawlers" and incorporated them in our sandbox artifact dataset (described in the next paragraph).

The second dataset (D_{sand}) consists of artifacts extracted by sandboxes belonging to various malware analysis services. We identified these sandboxes through prior work on sandbox evasion [23], as well as by querying popular search engines for keywords associated with malware analysis. Most sandboxes are not available for download, but do allow users to submit suspicious binaries which they then analyze and report on the activity of the analyzed binary, in varying levels of detail. Even though we were able to find 23 sandboxes that provide users the ability to upload suspicious files, our artifact-gathering server was able to collect artifacts from 15 different sandbox environments (not counting the aforementioned search-engine sandbox). We reason that the remaining analysis environments are either non-operational, rely on static analysis to scan executables, or completely block any network activity of each analyzed binary.

Table III lists the names of sandboxes from which our server received the results of our probing tools. We uploaded our probing tool multiple times in a period of a month and we also witnessed that some sandboxes executed our probing tool multiple times on different underlying environments (e.g., multiple versions of Microsoft Windows). As mentioned in Section IV-A, each uploaded executable had a unique ID embedded in its code which was sent to our artifact-gathering server, so as to perform accurate attribution.

The third and final dataset (D_{base}) is our baseline dataset which consists of artifacts extracted from fresh installations of multiple versions of Microsoft Windows. Next to setting up local VMs and installing Microsoft Windows XP, Windows Vista, Windows 7, and Windows 8, we also make use of all the Windows server images available in two large public cloud providers, Azure and AWS. Overall, D_{base} contains the artifacts extracted from 22 baseline environments.

D. Dataset Statistics

Table IV shows the number of different Microsoft Windows versions in our three collected datasets. There we see that the vast majority of users are utilizing Windows 7, 8 and 10, with only 4.1% of users still utilizing Windows XP and Windows Vista. Contrastingly, we see almost no sandboxes utilizing Windows 10, and the majority of them (83.7%) utilizing Windows 7. Note that because we extract the version of Windows programmatically, the same version number can correspond to more than one Windows versions, such as, Windows Server 2012 R2 and Windows Server 2016 having the same version with Windows 8.1 and Windows 10. Even though it is not the central focus of our paper, we find it troublesome that sandboxes do not attempt to follow the distribution of operating systems from real user environments.

Figure 1 shows boxplots that reveal the distribution of the values of each wear-and-tear artifact across all three datasets

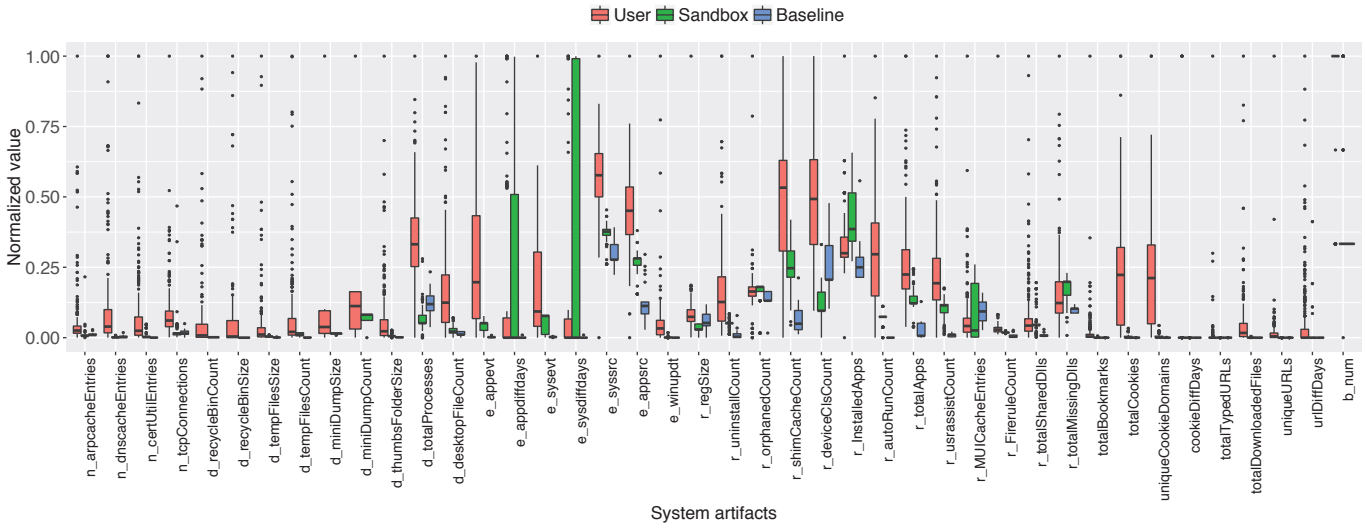

Fig. 1: Distribution of artifacts across user machines, sandboxes, and baselines (normalized values).

TABLE III: Sandboxes from which artifacts were successfully collected.

ID	Name	#Instances
1	Anubis	2
2	Avira	2
3	Drweb	3
4	Fortiguard	3
5	Fsecure	2
6	Kaspersky	15
7	McAfee	7
8	Microsoft	1
9	HybridAnalysis	1
10	Pctools	1
11	Crawlers	50
12	Malwr	2
13	Sophos	4
14	ThreatTrack	3
15	Bitdefender	1
16	Minotaur	1

TABLE IV: Distribution of operating systems in each dataset.

	v5.1	v6.0	v6.1	v6.2	v6.3	Total
Users (D_{real})	8	3	105	7	147	270
Sandboxes (D_{sand})	14	-	82	-	2	98
Baseline (D_{base})	1	1	2	1	17	22

v5.1: Windows XP
v6.0: Windows Vista / Win Server 2008
v6.1: Win7, Win Server 2008 R2
v6.2: Win8 / Win Server 2012
v6.3: Win 8.1/Win 10/ Win Server 2012 R2/ Win Server 2016

(D_{real}, D_{sand} and, D_{base}). There we can observe multiple patterns that will be of use later on in this paper. For example, we see that the vast majority of artifacts have larger values in real user systems, than they have in sandboxes and baselines. This observation, by itself, indicates that these artifacts can be used to differentiate between real machines and sandboxes and can thus be used for malware evasion purposes. Similarly, we also observe that the distributions of artifact values is wider for real systems than it is for systems that are not used by real users. This could indicate that these artifacts could also be used to predict the age of any given machine.

At the same time, we do see certain cases where the opposite behavior manifests. For instance, the appdiffdays and sysdiffdays artifacts are clearly more diverse in sandboxes than they are in real user systems. We argue that the explanation for this behavior is two-fold. As Table IV suggests, on average, sandboxes are "older" compared to real-user systems. As such, it is entirely within reason that the events in Windows' event log are further apart than those in real user systems. Moreover, assuming that the Windows log will only store events up to a maximum number, systems with more real-user activity are more likely to need to delete older events and hence reduce the time span observable in the Windows event log.

To verify, from a statistical point of view, the difference that we can visually identify between the distributions of artifacts, we need to use a metric that will allow us to differentiate between different distributions. Even though a normal distribution is usually assumed, by conducting a Shapiro-Wilk test we find that none of our extracted artifacts matches a normal distribution. For that reason, we use the Mann-Whitney U test to compare distributions, which is a non-parametric alternative of a t-test and does not assume that the data is normally distributed. We handle missing values by replacing them with dummy-coded values because the fact that an artifact cannot be extracted can be useful information for differentiating between systems.

Figure 2 shows the effect size of the differences between the distribution of D_{real} and D_{sand} as reported by the Mann-

Whitney U test. Using standard interpretations of effect sizes, an effect between 0.3 and 0.5 is considered to be characteristic of a "medium" effect while one higher than 0.5 is considered to be a "strong" effect. This confirms that the vast majority of our identified wear-and-tear artifacts have different distributions in sandbox and real-user environments.

V. Evading Malware Sandboxes

In the previous sections, we described the type of OS artifacts that we hypothesized would be indicative of wear and tear, and collected these artifacts from real user machines, sandboxes, and fresh installations of operating systems in virtualized environments. As our results showed, most artifacts have sufficiently different distributions, allowing them to be used to predict whether a given system is an artificial environment or a real user system, and evade the former while executing on the latter. In this section, we take advantage of these artifacts and treat them as features in a supervised machine-learning setting, allowing the implementation of simple decision trees that can be employed by malware for evasion purposes.

A. Setup and Classifier

When we were considering which classifier to use, we came to the conclusion that explainability of the results is a desirable property for our setup. It is important for the security community to understand how these seemingly benign features are different between real user systems and sandbox environments. Moreover, attackers can also benefit from a classifier with explainable results since they can, in real time, reason whether the result of their classifier makes intuitive sense given the execution environment. As such, even if an SVM-based classifier may perform slightly better than most explainable supervised learning algorithms, attackers have no way of evaluating the truthfulness of the result, especially in environments that are constantly evolving to bypass their evasion techniques.

For these reasons, we decided to use decision trees as our algorithm of choice. Next to their explainable results, a decision-tree model is also straightforward to implement as a series of if-else statements, which can be beneficial for attackers who wish to keep the overall footprint of their malware small. Before training our classifier, we inspected the correlation of pairs of features by calculating the Pearson correlation coefficient (r) for each possible pair. The results showed that four pairs of features have a strong correlation ($r > 0.7$). We decided, however, to include all features in our classifier, since from the point of view of defenders, they would have to address all correlated features in order to stop malware from successfully evading their analysis systems.

The training set of the classifier is composed as follows. For positive examples, we randomly sample 50% of our sandbox observations (49 instances) and add to them 100% (22 instances) of our fresh OS installations. We consider it important to augment the sandbox dataset (D_{sand}) with our baseline dataset (D_{base}) to cover a wider range of possible sandbox environments, starting from the most basic virtualization environments, to state-of-the-art sandboxes belonging to the security industry. To balance this data, we randomly sample 71 instances from real-user environments collected through Amazon Mechanical Turk (D_{real}). Therefore, our *training set* consists of 142 observations, balanced between the two classes. Similarly, our *testing dataset* comprises the remaining 50% of sandbox observations (49 instances) and a random sample of 49 real user machines which were not part of our training set. We handle missing feature values by setting them equal to zero.

To improve the accuracy of our classifier, we utilize two orthogonal techniques. First, we use a 10-trial adaptive boosting where the algorithm creates multiple trees, with each tree focusing on improving the detection of the previous tree [53]. Specifically, while constructing $tree_i$, the algorithm assigns larger weights to the observations misclassified by $tree_{i-1}$, aiming to improve the detection rate for these observations. Through the construction of multiple sequential trees, the adaptive boosting decision tree algorithm can learn boundaries that a single decision tree could not. Note that this choice does not really complicate the implementation of this algorithm by malware, as it can still be implemented as multiple sets of if-else rules, and use a weighted sum of all predictions according to the weights specified by the trained model.

In addition to adaptive boosting, we specify our own cost matrix where false negatives are ten times more expensive than false positives. The rationale for this decision is that we assume it is preferable for malware, from the point of view of the malware author, to miss an infection of a real user's machine (false positive), rather than to execute in a sandbox environment where its detection would mean the creation of signatures that would jeopardize its ability to infect systems protected with antivirus software.

B. Evaluation

Using the aforementioned training dataset and parameters, we trained an adaptive boosting decision tree and evaluated it on the testing set. Figure 3 shows three of the ten trees that our classifier generated, as well as the fraction of misclassification of each final decision which, as described earlier, drives the construction of the next tree.

Note that the testing set was not provided to our algorithm during training, in an attempt to quantify how well our model generalizes to previously unseen data. Our model achieved an accuracy of 92.86% on the testing set, with a false-negative rate (FNR) of 4.08% and a false positive rate (FPR) of 10.20%. The ten constructed trees had an average tree size of 4.6 splits, with the shortest tree having 3 splits and the longest one 5 splits. Table V shows the features that were the most useful to the algorithm, and the percentage of trees using any given feature.

To assess the robustness of our algorithm against incremental changes in sandbox environments, we repeated our aforementioned experiment 30 times, each time removing an additional feature from the original datasets, compiling new, randomly

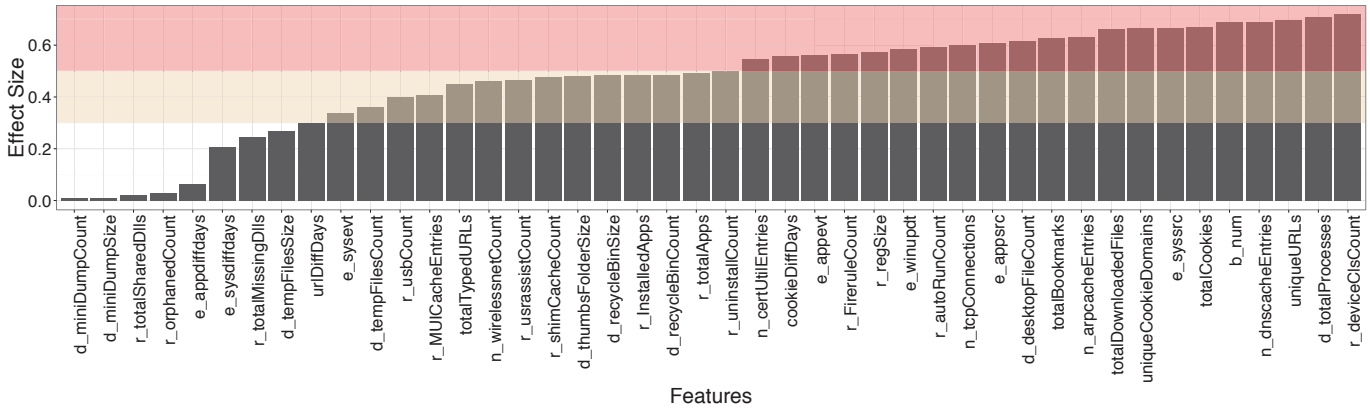

Fig. 2: The difference of the distributions of a given artifact's values between real user systems and sandboxes, based on the Mann-Whitney U test. An effect size above 0.3 (highlighted with light yellow) can be interpreted as a medium difference, while an effect size above 0.5 (highlighted with red) corresponds to a high difference between the two distributions.

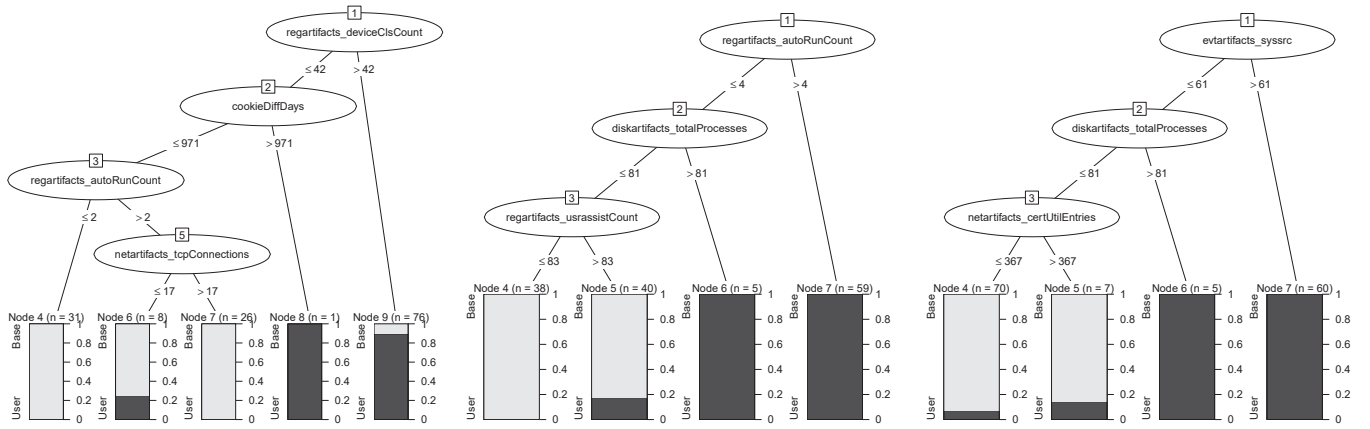

Fig. 3: Trees 1, 3, and 5, calculated by the C5.0 decision tree algorithm with adaptive boosting and a custom cost matrix that favors false positives over false negatives.

sampled, training and testing datasets using the splits described earlier, and training and testing the new classifier. Specifically, we remove the feature that was the most used by the previous model, i.e., the most important one. For example, the first feature that we removed was the number of entries in a system's DNS cache, as it was the most commonly used feature of our first classifier according to Table V. Through this process we aim to quantify an attacker's ability of distinguishing sandboxes from real users, in the face of dedicated sandbox operators who can try to mimic a real user system by populating our identified wear-and-tear artifacts with realistic values.

We argue that this setup captures the worst-case scenario for an attacker, because not only are there fewer and fewer features to train on, but also because the remaining features are of lower discriminatory power than the removed ones. Figure 4 shows how the accuracy, false positive rate, and false negative rate vary, as we keep on removing features from our dataset. As one can notice, the overall accuracy of the classifier remains high (greater than 90%) even if we remove as many as 20 discriminatory features.

Similarly, we see that, guided by our custom cost matrix, the classifier constantly prefers to err on the side of false positives instead of false negatives. We even observe that for the first eight features, the accuracy increases and the FPR/FNR rates decrease. This shows that some pattern that was learned by the training data concerning the removed features did not appear in the testing data, and thus removing these features, in fact, made the classifier more accurate. The average accuracy of the first 30 classifiers (removing from 0 to 29 features) is 91.5% with a FPR of 9.05% and a FNR of 7.96%. When the 30th feature is removed (out of 33 in total), the decision tree algorithm fails to train a model with accuracy greater than random chance, and quits.

To better understand how each type of artifact affects the accuracy of our classifier, we retrained our model using only one type of artifacts at a time. Table VI shows the number of artifacts per category and the performance of each classifier on

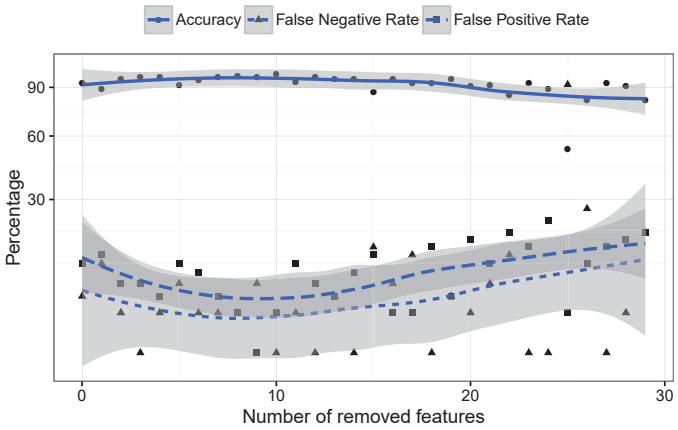

Fig. 4: Variance of accuracy, false negative rate, and false positive rate of our boosted classifier as we keep removing the topmost features. Even after removing tens of the features with the highest discriminatory capacity, the accuracy of the classifier drops only slightly.

the unseen testing dataset. We can observe that, even though the classifier based on registry artifacts had the most features available to it, it performed worse than the Network-based and Disk-based classifiers. This indicates that certain classes of artifacts have higher discriminatory power than the rest. At the same time, all classifiers achieve fairly high accuracy (some even have a 0% FNR), which suggests that malware authors can choose which category of artifacts they can focus on, without compromising the evasion capabilities of their malware. As such, malware authors who, for example, know that their samples will be analyzed by a sandbox that considers registry-related activity suspicious, can use the other types of wear-and-tear artifacts for both detecting the sandbox and avoiding raising suspicion.

Our results demonstrate that, not only is it possible to differentiate between real systems and sandboxes based on non-intrusive, sandbox-agnostic wear-and-tear artifacts, but that incremental patching approaches where one or two artifacts are made to look similar to those in real systems will not be sufficient to prevent sandbox evasion.

VI. Estimating Actual System Age

In the previous section, we showed that wear-and-tear artifacts can be used to accurately differentiate between machines belonging to real users and sandbox environments. The underlying intuition is that sandbox environments do not "age" either at all, or clearly not in the same way as user environments. Based on this observation, in this section, we explore the possibility of predicting the age of a system based on the values of its wear-and-tear artifacts. This possibility would allow malware to assess whether a machine exhibits a realistic wear and tear that matches its reported age. At the same time, it would also enable researchers to artificially age malware sandboxes to match a desired age.

TABLE V: Features that the adaptive boosting decision trees rely on, and the percentage of trees utilizing them.

Category	Feature	% of trees
Network	dnscacheEntries	100.00%
System	sysevt	100.00%
System	syssrc	100.00%
Registry	deviceClsCount	100.00%
Registry	autoRunCount	100.00%
Browser	uniqueURLs	100.00%
Disk	totalProcesses	66.90%
Browser	cookieDiffDays	59.86%
Registry	usrassistCount	54.93%
Network	certUtilEntries	54.23%
System	appevt	54.23%
Browser	uniqueCookieDomains	52.82%
Network	tcpConnections	50.00%
System	winupdt	16.90%
Registry	MUICacheEntries	11.27%
Network	ARPCacheEntries	4.93%

TABLE VI: Classifier performance when trained only on a single type of artifacts (FNR = False Negative Rate, FPR = False Positive Rate).

Artifact Type	#Artifacts	Accuracy	FNR	FPR
Network	4	95.92%	2.04%	6.12%
Disk	5	95.92%	0.00%	8.16%
System	5	92.86%	0.00%	14.29%
Registry	11	93.88%	4.08%	8.16%
Browser	7	92.86%	2.04%	12.24%

A. Correlation Between Age and Artifacts

We begin this process by first investigating the self-reported age of sandboxes and real user machines. Figure 5 shows the cumulative distribution function of the age of user systems and malware sandboxes. We can clearly see that, on average, user systems are much "newer" than sandboxes. Half of the users have systems less than a year old, yet 50% of sandboxes are less than two years old. This makes intuitive sense since most users update their hardware on a regular basis, while many sandboxes were setup once and are just reused over and over for a long period of time. At the same time, we see that the two distributions meet at the top, meaning that there some users have systems that are as old (or even older) than the oldest available sandboxes.

Having established their age, we can now investigate the extent to which each individual wear-and-tear artifact can potentially be used to predict an environment's age. Figure 6 shows the Pearson correlation between the reported age and each wear-and-tear artifact for real user systems. We argue that changing the age reported by the OS is not only something that everyday users cannot do, but that they do not have any interest of doing. As such, we assume that the reported ages collected from real user systems are accurate.

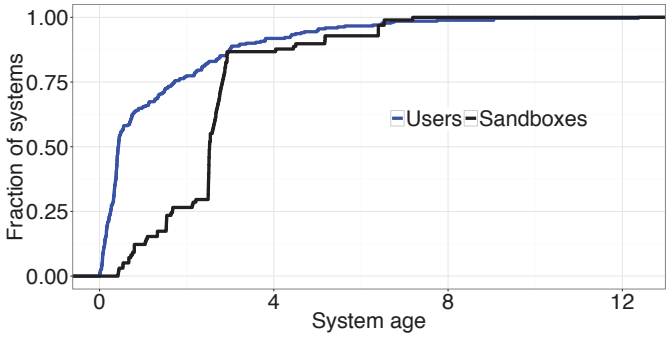

Fig. 5: CDF of the reported ages for sandboxes and real user machines.

We see that many artifacts that were very successful in differentiating real machines from sandboxes do not, in fact, correlate with a machine's age. By focusing on the artifacts that appear towards the middle of Figure 6 (having a correlation coefficient close to zero), one can notice that these artifacts may be characteristic of user activity but they are also under a user's control. Users can delete their cookies, visit more or fewer URLs than other users, delete downloaded files, and, in general, affect the values of these artifacts in unpredictable ways that prevent them from being indicative of system's age. Similarly, there are other artifacts, such as the number of cached wireless networks, TCP connections, and firewall rules, which are not under the direct control of users, clearly do not correlate with age, yet vary sufficiently between sandboxes and real machines to be of use for differentiation.

Towards the right end of Figure 6 we see the artifacts that appear to positively correlate with system age, such as the number of previously connected USBs, the number of install applications, and the elapsed time since the first user event. These artifacts are either "proxies" of age (e.g., the longer a machine is used, the more applications are installed on it), or are direct sources of age information which, unlike HTTP cookies, are buried deeper in an OS's internals, and are thus unlikely to be modified or deleted by users.

B. Regression

Having established that some artifacts positively correlate with age, we now attempt to identify the appropriate way of combining their raw values to estimate a system's actual age. To that end, we first process our dataset to remove artifacts with missing values in many of our observations. For artifacts with a high rate of missing values (more than 80% of their values are missing) we remove them from our dataset and do not attempt to utilize them any further. For the remaining 37 artifacts, we used the rest of the dataset to predict what would have been the most appropriate value, had we been able to extract it. We use a random-forest-based imputation technique, which is a standard way of handling missing values, without the need to remove entire observations and without relying on overly generic statistics, like a feature's mean or mode, which could bias our results.

TABLE VII: Significant coefficients of linear regression for predicting the age of a system based on wear-and-tear artifacts.

Feature	Coeficient	p-Value
diskartifacts:tempFilesCount	0 2.92	0.0041 **
diskartifacts:totalProcesses	-2.12	0.0364 *
evtartifacts:appevt	02.39	0.0186 *
evtartifacts:sysevt	1.71	0.0901 .
evtartifacts:syssrc	-3.09	0.0025 **
evtartifacts:appsrc	2.57	0.0114 *
regartifacts:regSize	2.70	0.0080 **
regartifacts:uninstallCount	-2.07	0.0411 *
regartifacts:deviceClsCount	1.85	0.0664 .
regartifacts:InstalledApps	5.13	1.2e-06 ***
regartifacts:totalApps	1.99	0.0490 *
totalDownloadedFiles	2.11	0.0368 *
browser:num	-2.56	0.0117 *
Significance Codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1		
Multiple R-Squared: 0.68, Adjusted R-Squared: 0.583		

1) Linear Regression: We first attempt to use linear regression, since this is a common regression method that works well in many contexts. In essence, we start with the formula $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \epsilon$ where X_i is the value of any given artifact, and try to calculate the weights β_i that would allow us to sum the artifacts in a way that will provide a prediction of machine's age (Y).

To train a linear regression model based on the real users' dataset, we first split the user dataset into two sets, one for training (60%) and one for testing (40%). We keep the test dataset aside to use it later to test how well our model generalizes on previously unseen data. Our linear regression model is trained on the training set, and we use a 10-fold cross validation to evaluate it, which results in a mean square error (MSE) of 1.23. The coefficients of the model and their p-values are reported in Table VII. We see that 13 wear-and-tear artifacts correlate with a machine's age in a statistically significant way. Even though our model is not perfect, it can still explain up to 68% of the variability in a system's age.

Since the training error does not necessarily guarantee the performance of our model when dealing with new data, we evaluate the accuracy of the linear regression model against the previously unseen test dataset. We first apply our model to the unseen test dataset of systems belonging to real users to make sure that it can predict their age with sufficient accuracy, and then proceed to use it to predict the age of sandboxes and compare it with their claimed age. The MSE of the predicted ages for the regular systems is 1.88, while sandboxes have a very high MSE of 6.25. Our results indicate that a model trained on real user data can not only predict the age of other unseen systems based on the values of their wear-and-tear artifacts, but it can also identify that a system's artifacts are unrealistically high (or low) for its claimed age.

Figure 7 shows the CDF of the values of residuals when applying our linear regression model on systems from D_{real}

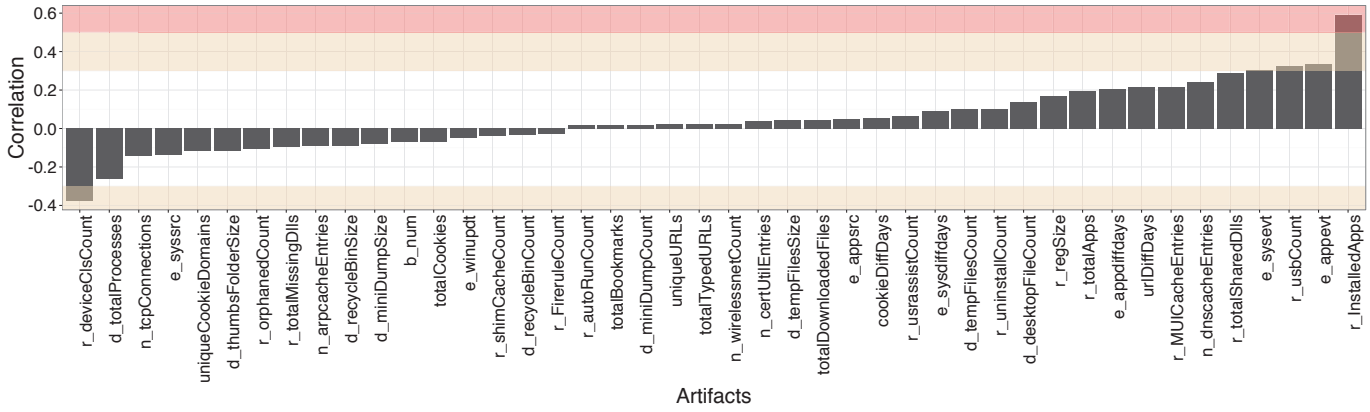

Fig. 6: Correlation between the reported age of user systems and each of the artifacts.

TABLE VIII: Lasso regression coefficients of the features selected as good predictors of system age by the model.

Feature	Coeficient
netartifacts:dnscacheEntries	0.1383
diskartifacts:tempFilesCount	0.0883
evtartifacts:appevt	0.1345
evtartifacts:sysevt	0.1097
regartifacts:regSize	0.1160
regartifacts:InstalledApps	0.6432
regartifacts:totalApps	0.0559
browser:num	-0.0138

and D_{sand}. We see that although the majority of real user systems have a residual error approximately equal to zero (meaning that the predicted age closely matches the system's actual age), the residuals when attempting to predict the age of sandboxes are much higher, i.e., the predicted age is much lower than the reported age, thereby shifting the distribution towards the right part of the graph.

2) Lasso Regression: To explore whether a more involved regression model would result in better prediction accuracy, we also report the results of Lasso regression. Lasso regression penalizes the absolute size of the regression coefficients, and sets the coefficients of unnecessary features to zero, reducing in this way the size of our artifacts set. One of the benefits of Lasso regression is that it typically utilizes less predicting variables than linear regression, which reduces the complexity of the overall model. In the case of malware, a smaller feature set means fewer API calls to the underlying OS, and thereby less chances of triggering suspicious activity monitors.

For our experiments, we split the datasets to three parts: training, evaluation, and testing. We first train a Lasso model on the training set and make use of cross validation to find the optimum λ value. We use the discovered value ($\lambda = 0.145$), as this minimizes the MSE of the model, and train our final model. Table VIII shows the eight features that the Lasso model selected as good predictors of system age and their corresponding coefficients. The only artifact that has a small negative coefficient is the number of browsers installed on a system. One reasonable explanation for this is that owners of older systems may hesitate to install additional browsers, which has an effect on our trained model.

By applying the model on the unseen dataset that consists of artifacts from real user systems (D_{real}), the MSE is 0.749, which is better than that of linear regression. The MSE on the testing sandbox dataset is 4.45, which again shows how the wear and tear of sandboxes does not match their claimed age. Figure 7 shows the residual errors of the trained Lasso regression model, and how these compare to the errors of the linear regression model. We observe that although both models can predict the age of real user systems (residual errors approximately equal zero), the Lasso model can better differentiate some sandboxes based on fewer artifacts compared to the linear regression model.

Overall, we see that wear-and-tear artifacts can predict, with sufficient accuracy, the real age of a system and thus reveal the age mismatch present in current sandboxes. As we mentioned earlier in this section, other than just improving our understanding of the relationship between artifacts and system age, our trained models can be of use to sandbox developers to help them create environments with wear and tear that matches their stated age.

VII. Discussion

A. Ethical Considerations and Coordinated Disclosure

Our research sheds light to the problem of next-generation environment-aware malware based on wear and tear artifacts. By demonstrating the effectiveness of this new evasion technique, we will hopefully bring more attention to this issue, and encourage further research on developing effective countermeasures. To that end, our preliminary investigation on predicting a systems' age based on the evaluated artifact values can help sandbox operators to fine-tune the wear-and-tear characteristics of their systems so that they look realistic. There have been very recent indications that malware authors have already started taking into consideration usage-related artifacts

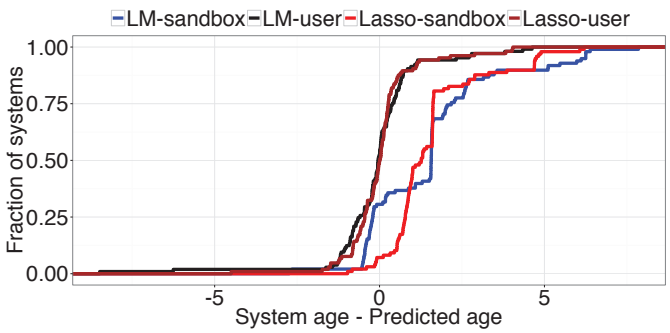

Fig. 7: CDF of the difference between the reported and estimated age for sandboxes and real systems, using the linear and Lasso regression models.

(e.g., whether any documents appear under the recently opened documents menu [54]). Consequently, developing effective countermeasures before malware authors start taking advantage of wear and tear for evasion purposes more broadly, is of utmost importance.

Although our results regarding the analyzed sandboxes used by public malware analysis services showcase their vulnerability to evasion based on wear and tear, adversaries may have already come to the same conclusion even before us, given that our probing technique was reasonably straightforward, and that similar techniques have also been used as part of previous research on public sandbox fingerprinting [23]. Hopefully, by publishing our findings and coordinating with the affected vendors, our research will eventually result in more robust malware analysis systems.

To that end, we contacted all vendors from which our probe tool managed to collect wear-and-tear artifact information, notifying them about our findings several months before the publication of this paper. Some of these vendors acknowledged the issue and requested additional information, as well as the code of our artifact-probing tool so that they can more easily detect future artifact exfiltration attempts.

B. Probing Stealthiness

Depending on the type of artifact, the actual operations that some malware needs to perform in order to retrieve the necessary information range from simple memory or file system read operations, to more complex parsing and querying operations that may involve system APIs and services. Instead of making sandboxes look realistic from a wear-and-tear perspective, an alternative (at least short-term) defense strategy may thus focus on detecting the probing activity itself. At the same time, many environmental aspects can be probed or inferred through non-suspicious system operations commonly exhibited during the startup or installation of benign software (e.g., retrieving or storing state information from the registry or configuration files, or checking for updates), while malware can always switch to a different set of artifacts that are not monitored for suspicious activity.

Another aspect that must be considered is that as the principle of least privilege is better enforced in user accounts and system services, a malicious program may not have the necessary permissions to access all system resources and APIs. We did not encounter any such issues with our probe tool on the tested operating systems, but based on our findings, there is plenty of artifacts that can be probed even by under-privileged programs. This issue may become more important in other environments, such as, in malicious mobile apps, which are much more constrained in terms of the environmental features they can access.

C. OS Dependability

We have focused on the Windows environment, given that it is one of the most severely plagued by malware, and that most malware analysis services employ Windows sandboxes. As a result, many of the identified artifacts are Windows-specific (e.g., the registry-related ones), and clearly not applicable for malware that target other operating systems (e.g., Linux, Mac, Android). Even so, artifacts related to browser usage, the file system, and the various network caches, are likely to be present regardless of the exact operating system.

Based on the dependability of an artifact on a given OS, sandbox operators may choose different defensive strategies. For instance, operators of sandboxes tailored to the detection of cross-platform malware (e.g., written in Java) may start addressing OS-independent artifacts first, before focusing on OS-dependent ones. Finally, since end-user systems are the most popular targets of malware authors, our study is focused solely on them. We leave the analysis of wear-and-tear artifacts on other platforms, such as embedded devices, for future work.

D. Defenses

We have demonstrated that removing system instrumentation and introspection artifacts from malware sandboxes is not enough to prevent malware evasion. Since the lack of wear and tear can allow attackers to differentiate between dynamic analysis systems and real user systems, *introducing* wear and tear artifacts to analysis systems becomes an additional necessary step. We outline two different potential strategies to achieve this.

First, a real user system can be cloned and used as a basis for a malware sandbox. In this scenario, the system will already have organic wear and tear which can be used to confuse attackers. Potential difficulties of adopting this approach are: i) privacy issues (how does one scrub all private information before or after cloning, but leaves wear-and-tear artifacts intact?), and ii) eventual outdating of the artifacts (if the cloned system is used over a long period of time, an attacker can use our proposed statistical models to detect that the claimed age does not match the level of wear and tear).

An alternative approach is to start with a freshly installed image of an operating system and artificially age it by automatically simulating user actions. This would include installing and uninstalling different software while changing the system time to the past, browsing popular and less popular

webpages, and, in general, taking actions which will affect the wear-and-tear artifacts on which an attacker could rely for detecting dynamic analysis systems. While this approach does not introduce any privacy concerns and can be repeated as often as desired, it is unclear to what extent this artificial aging will produce realistic artifacts that are similar enough to those of real systems.

We leave the implementation and comparison of these two approaches for future work.

VIII. Conclusion and Future Work

In this paper, we have systematically assessed the threat of a malware sandbox evasion strategy that leverages artifacts indicative of the wear and tear of a system to identify artificial environments, and demonstrated its effectiveness against existing sandboxes used by malware analysis services. As long as this aspect is not taken into consideration when implementing malware sandboxes, malicious code will continue to be able to effectively alter its behavior when being analyzed. Even approaches that move from emulated and virtualized environments to bare-metal systems [8]–[10] will be helpless if evasion tactics shift from how realistic a system looks like, to how realistic its *past use* looks like. The same also holds for existing techniques that identify split personalities in malware by comparing malware behavior on an emulated analysis environment and on an unmodified reference host [44] or multiple sandboxes [18]. As long as all involved systems do not look realistic from the perspective of a real user's activity, they will be easily detectable and, therefore, evadable.

As a step towards mitigating this threat, we have presented statistical models that capture a system's age and degree of use, which can aid sandbox operators in fine-tuning their systems so that they exhibit more realistic wear-and-tear characteristics. Although addressing each and every identified artifact may be a viable short-term solution, more generic automated "aging" techniques will probably be needed to provide a more robust defense, as many more artifacts may be available.

As part of our future work, we plan to explore such approaches based on simulation-based generation, as well as privacy-preserving transformation of system images extracted from real user devices. We also plan to evaluate the effectiveness of sandbox evasion based on wear and tear in other environments, such as, different desktop operating systems as well as mobile devices.

Acknowledgments

We thank our shepherd, Tudor Dumitras, and the anonymous reviewers for their valuable feedback. This work was supported in part by the Office of Naval Research (ONR) under grant N00014-16-1-2264, and by the National Science Foundation (NSF) under grants CNS-1617902 and CNS-1617593, with additional support by Qualcomm. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, NSF, or Qualcomm.

References

[1] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *Security Privacy, IEEE*, vol. 5, no. 2, pp. 32–39, March 2007.

[2] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," in *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR)*, April 2006.

[3] X. Jiang and X. Wang, "'Out-of-the-box' monitoring of VM-based high-interaction honeypots," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007, pp. 198–218.

[4] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 116–127.

[5] "Anubis: Malware analysis for unknown binaries," https://anubis.iseclab.org/.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008, pp. 51–62.

[7] "Automated malware analysis: Cuckoo sandbox," https://cuckoosandbox.org/.

[8] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare metal: Using processor features for binary analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 189–198.

[9] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 287–301.

[10] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, December 2015.

[11] J. Oberheide and C. Miller, "Dissecting the Android Bouncer," 2012, https://jon.oberheide.org/files/summercon12-bouncer.pdf.

[12] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, "Enter sandbox: Android sandbox comparison," in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[13] "FireEye Malware Analysis AX Series," https://www.fireeye.com/products/malware-analysis.html.

[14] "Blue Coat Malware Analysis Appliance," https://www.bluecoat.com/products/malware-analysis-appliance.

[15] "Cisco Anti-Malware System," http://www.cisco.com/c/en/us/products/security/web-security-appliance/anti_malware_index.html.

[16] "FortiNet Advanced Threat Protection (ATP) - FortiSandbox," http://www.fortinet.com/products/fortisandbox/index.html.

[17] "McAfee Advanced Threat Defense," http://www.mcafee.com/us/products/advanced-threat-defense.aspx.

[18] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011, pp. 338–357.

[19] C. Kolbitsch, "Analyzing environment-aware malware," 2014, http://labs.lastline.com/analyzing-environment-aware-malware-a-look-at-zeus-trojan-variant-called-citadel-evading-traditional-sandboxes.

[20] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of mobile malware," in *Proceedings of the 7th European Workshop on System Security (EuroSec)*, April 2014.

[21] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 637–652.

[22] G. Stringhini, C. Kruegel, and G. Vigna, "Shady paths: Leveraging surfing crowds to detect malicious web pages," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013, pp. 133–144.

[23] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, and C. Rossow, "SandPrint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016, pp. 165–187.

[24] D. Desai, "Malicious Documents leveraging new Anti-VM & Anti-Sandbox techniques," https://www.zscaler.com/blogs/research/malicious-documents-leveraging-new-anti-vm-anti-sandbox-techniques, 2016.

[25] ProofPoint, "Ursnif Banking Trojan Campaign Ups the Ante with New Sandbox Evasion Techniques," https://www.proofpoint.com/us/threat-insight/post/ursnif-banking-trojan-campaign-sandbox-evasion-techniques, 2016.

[26] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Proceedings of the 13th Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[27] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[28] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy, "A crawler-based study of spyware on the web," in *Proceedings of the 13th Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[29] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy, "Spyproxy: Execution-based detection of malicious web content," in *Proceedings of the 16th USENIX Security Symposium*, 2007.

[30] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the World Wide Web Conference (WWW)*, 2010.

[31] "Wepawet: Detection and Analysis of Web-based Threats," https://wepawet.iseclab.org/.

[32] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 31–39.

[33] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and detecting malicious flash advertisements," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Honolulu, HI, December 2009.

[34] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008.

[35] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2014, pp. 447–458.

[36] T. Klein, "ScoopyNG – The VMware detection tool," 2008, http://www.trapkit.de/tools/scoopyng/index.html.

[37] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proceedings of the 1st ACM Workshop on Virtual Machine Security (VMSec)*, 2009, pp. 11–22.

[38] P. Ferrie, "Attacks on Virtual Machine Emulators," http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.

[39] ——, "Attacks on More Virtual Machine Emulators," http://pferrie.tripod.com/papers/attacks2.pdf.

[40] Y. Bulygin, "CPU side-channels vs. virtualization rootkits: the good, the bad, or the ugly." ToorCon, 2008.

[41] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction," 2004, https://web.archive.org/web/20071112073957/http:/www.invisiblethings.org/papers/redpill.html.

[42] O. Bazhaniuk, Y. Bulygin, A. Furtak, M. Gorobets, J. Loucaides, and M. Shkatov, "Reaching the far corners of MATRIX: generic VMM fingerprinting." SOURCE Seattle, 2015.

[43] P. Jung, "Bypassing sanboxes for fun!" BotConf, 2014, https://www.botconf.eu/wp-content/uploads/2014/12/2014-2.7-Bypassing-Sandboxes-for-Fun.pdf.

[44] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2010.

[45] M. K. Sun, M. J. Lin, M. Chang, C. S. Laih, and H. T. Lin, "Malware virtualization-resistant behavior detection," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011, pp. 912–917.

[46] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011, pp. 403–412.

[47] C. Wueest, "Does malware still detect virtual machines?" 2014, http://www.symantec.com/connect/blogs/does-malware-still-detect-virtual-machines.

[48] A. Singh, "Defeating Darkhotel Just-In-Time Decryption," 2015, http://labs.lastline.com/defeating-darkhotel-just-in-time-decryption.

[49] D. Maier, T. Müller, and M. Protsenko, "Divide-and-conquer: Why android malware cannot be stopped," in *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*, 2014, pp. 30–39.

[50] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener, "AVleak: Fingerprinting antivirus emulators through black-box testing," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT)*, Austin, TX, 2016.

[51] "CleanMyPC Registry Cleaner," http://www.registry-cleaner.net/.

[52] "CCleaner - Registry Cleaner," https://www.piriform.com/ccleaner/registry-cleaner.

[53] R. E. Schapire and Y. Freund, *Boosting: Foundations and algorithms*. MIT press, 2012.

[54] M. Cova, "Evasive JScript," 2016, http://labs.lastline.com/evasive-jscript.