MALREC: Compact Full-Trace Malware Recording for Retrospective Deep Analysis

Giorgio Severi[1(✉)], Tim Leek[2], and Brendan Dolan-Gavitt[3]

[1] Sapienza University of Rome, Rome, Italy
severi.1462794@studenti.uniroma1.it
[2] MIT Lincoln Laboratory, Lexington, USA
tleek@ll.mit.edu
[3] New York University, New York, USA
brendandg@nyu.edu

Abstract. Malware sandbox systems have become a critical part of the Internet's defensive infrastructure. These systems allow malware researchers to quickly understand a sample's behavior and effect on a system. However, current systems face two limitations: first, for performance reasons, the amount of data they can collect is limited (typically to system call traces and memory snapshots). Second, they lack the ability to perform *retrospective analysis*—that is, to later extract features of the malware's execution that were not considered relevant when the sample was originally executed. In this paper, we introduce a new malware sandbox system, MALREC, which uses whole-system deterministic record and replay to capture high-fidelity, whole-system traces of malware executions with low time and space overheads. We demonstrate the usefulness of this system by presenting a new dataset of 66,301 malware recordings collected over a two-year period, along with two preliminary analyses that would not be possible without full traces: an analysis of kernel mode malware and exploits, and a fine-grained malware family classification based on textual memory access contents. The MALREC system and dataset can help provide a standardized benchmark for evaluating the performance of future dynamic analyses.

Keywords: Malware analysis · Record and replay
Malware classification

1 Introduction

As the number of malware samples seen each day continues to grow, automated analyses have become a critical part of the defenders' toolbox. Typically,

Tim Leek's work is sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract #FA8721-05-C-0002 and/or #FA8702-15-D-0001. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

C. Giuffrida et al. (Eds.): DIMVA 2018, LNCS 10885, pp. 3–23, 2018.
https://doi.org/10.1007/978-3-319-93411-2_1

these analyses can be broadly divided into static and dynamic analyses. Static analyses, which do not need to actually execute a sample, have the benefit of being often highly scalable, but are easily foiled by obfuscation and packing techniques. On the other hand, dynamic malware analysis systems can quickly extract behavioral features of malware by running them inside an instrumented virtual machine or full-system emulator, but are less scalable since each sample must run for a certain amount of real (wall-clock) time.

However, current sandbox systems suffer from several limitations. First, they must choose between deep analysis (in the form of heavyweight monitoring) and transparency. The sandbox may take a very light hand and only observe what malware does at a gross level, such as files read or written, configuration changes, network activity, etc. Or the sandbox may attempt to gather very detailed information about behavior, such as the sequence of function calls and their arguments. However, the instrumentation we may wish to perform while the malware is running might be invasive and slow down the sandbox to much slower than real time performance, which may mean the malware will behave differently than it would when not under instrumentation (for example by causing network connections to time out). This means that expensive analyses such as dynamic taint analysis [25] are out of reach for malware sandbox systems.

Second, the effective "shelf life" of malware samples is limited under dynamic analysis. Modern malware is typically orchestrated via communication with remote command-and-control servers. When these servers go down (e.g., because they are taken down due to anti-malware efforts), the malware is less likely to work correctly and will display a stunted range of behaviors. Hence, dynamic analyses of malware must be done while the malware is new. This means that new analyses and features can only be employed on new samples; *retrospective* analysis is not possible, since older samples will not exhibit their original behavior when re-run. We believe this problem to be particularly relevant in the context of forensic analysis and when new, previously unknown, samples are encountered.

This latter problem also leads to a severe shortage of standard datasets for dynamic malware analysis. Such datasets cannot consist merely of the malware binaries, since these will go stale. Summaries of observed behavior, such as those provided by online services like Malwr[1], necessarily capture only a subset of the malware's activity in the sandbox. And while detailed logs, such as full instruction and memory traces, may provide sufficient fidelity to perform retrospective analyses, their cost is prohibitive, requiring many gigabytes of storage per trace and imposing a large overhead on the sandbox's runtime performance. Dynamic analysis datasets are therefore limited to the features their creators thought to include and cannot be extended, which limits their utility for future research.

A solution to this problem is dynamic analysis under deterministic replay [16, 35]. If we run a malware sample while collecting enough information to permit us to replay the whole system execution with perfect fidelity, then we are free to use expensive dynamic analysis after the fact. We can even iterate, performing

[1] https://malwr.com/.

one cheap dynamic analysis to determine if another, more expensive one is likely to be worthwhile and feeding intelligence gleaned from shallower analysis to deeper ones. There is no worry about slowing down the sandbox overly since we perform all analyses under replay. And since no code is introduced into the guest to instrument or interpose to collect dynamic features, the malware is more likely to behave normally. If the recording, which consists of an initial snapshot plus the log of nondeterministic inputs to the system, can be made small enough, we can not just collect but also store these recordings in order to apply new analyses dreamt up at later dates, as well as retrospective analyses in general. We can share the whole-system malware recordings with other researchers and, if we also provide code to collect and analyze dynamic features, enable them to reproduce our results perfectly.

In this paper, we present MALREC, a system that captures dynamic, whole-system behavior of malware. Although none of its technical features (whole system emulation, virtual machine introspection, and deterministic record and replay) are wholly novel, their combination permits deeper analyses than are currently possible. We demonstrate this by presenting a dataset of 66,301 full-system malware traces that capture all aspects of dynamic execution. Each trace is compact—the entire dataset can be represented in just 1.3 TB. We also describe three novel analyses of this dataset: an analysis of how many unique blocks of code are seen in our dataset over time, a comprehensive accounting of kernel malware and how each sample achieved kernel privileges, and a novel technique for malware classification and information retrieval based on the textual content (i.e., English words) read from and written to memory as each sample executes. Each of these analyses is currently too heavyweight to be run on a traditional malware sandbox, but we show that they can be performed at a reasonable price and a relatively short amount of time by taking advantage of the embarrassingly parallel nature of the computations.

By providing full-trace recordings, we hope to enable new research in dynamic analysis by making it easier for researchers to obtain and analyze dynamic execution traces of malware. Moreover, we believe that the dataset we provide can serve as a standard benchmark for evaluating the performance of new algorithms. And although ground truth is always elusive in malware analysis, a fixed dataset that captures all dynamic features of interest allows us to steadily improve our understanding of the dataset over time.

2 Design

2.1 Background: Record and Replay

The MALREC sandbox is built on top of PANDA [11], a whole-system dynamic analysis platform. The key feature of PANDA for our purposes is *deterministic record and replay*. Record and replay, as implemented in PANDA, captures compact, whole-system execution traces by saving a snapshot of the system state at the beginning of the recording and then recording all sources of *non-determinism*: interrupts, input from peripherals, etc. At replay time, the snapshot is loaded

and the system is run with all peripherals disabled; the stored non-deterministic events are replayed from the log file at the appropriate times. This ensures that as long as we have accounted for all sources of non-determinism, the replayed execution will follow the exact path as the original execution.

The 66,301 traces in our dataset can be compactly represented in just 1.3 TB, but are sufficient to capture every aspect of the 1.4 quadrillion instructions executed. This allows us to decouple our analyses from the execution of each sample, an idea first proposed by Chow et al. [9]. In Sects. 4 and 5 discuss the results of several analyses that would be too heavyweight to run on a live execution but are practical to run on a replay.

Although the record/replay system in PANDA is relatively robust, there are still some sources of non-determinism which are not captured (i.e., bugs in our record/replay system). As a result, some recordings cannot be correctly replayed. In our dataset, 2,329 out of 68,630 (3.4%) of our recordings cannot be replayed and hence are omitted from the analyses presented in this paper. With additional engineering it should be possible to fix these bugs and guarantee deterministic replay for all samples.

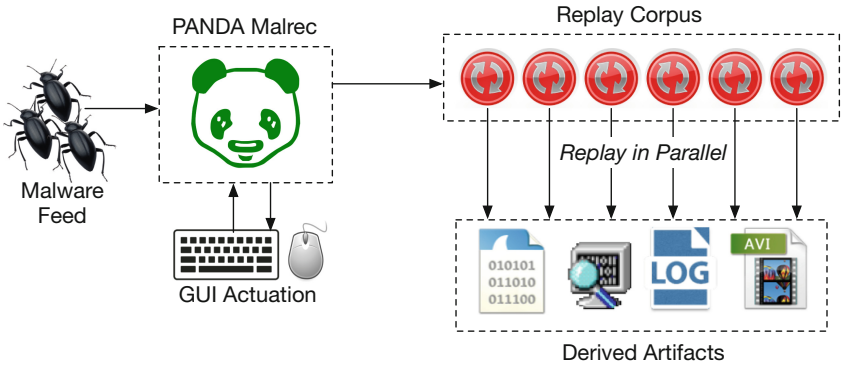


Fig. 1. The MALREC recording system. Malware samples are ingested and recorded; our actuation attempts to stimulate behavior by clicking on GUI elements. The resulting recordings can then be replayed to produce may different kinds of derived artifacts: network logs, screenshots, memory dumps, etc.

2.2 Recording Setup

Unlike many sandboxes, MALREC is *agentless*: no special software is installed inside the guest virtual machine. Instead, behavioral reports are generated later by PANDA plugins that run on the replayed execution. This increases the transparency of the emulated environment, though it is still vulnerable to sandbox evasion techniques that target the underlying emulator (QEMU). In Sect. 3.2 we provide upper and lower bounds on the number of evasive samples in our dataset.

Malware samples are loaded into MALREC via a virtual CD-ROM and then copied into the guest filesystem. Next, time synchronization is performed; this is needed both to improve transparency and because many protocols (e.g., HTTPS) depend on the client's clock being set correctly. Finally, the sample is executed with Administrator privileges. All commands are entered by simulating keypresses into the virtual machine. The recording setup is depicted in Fig. 1.

Once the malware has been started, we allow it to run for ten minutes (real time). During this time period, we periodically use virtual machine introspection (specifically, a module based on Volatility's Windows GUI support [1]) to look for buttons on screen that we should click on. This is accomplished by parsing the Windows kernel's GUI-related data structures and looking for elements that contain text such as "OK", "I agree", "Yes", "Go", etc. The goal is to get higher coverage for samples that require some user interaction.

Each PANDA replay consists of an initial VM snapshot and a log of nondeterministic events. After the recording has ended, we compress the log of nondeterministic events using `xz`. For the initial snapshot, we observe that each recording starts off with a nearly identical initial system snapshot. Thus, rather than trying to compress and store the snapshot for each recording, we instead store the differing bytes from a set of reference images. In a test on a subset of our data (24,000 recordings), we found that this provides savings of around 84% (i.e., a 6x reduction) in the storage required. This gives us an effective storage rate of around 1048 instructions per byte.

2.3 Offline Analyses

Once we have our recordings of malware, we can perform decoupled, offline analyses to extract features of interest. Because our recordings are made with the PANDA dynamic analysis system, our analyses take the form of PANDA plugins, but we anticipate that analyses from other QEMU-based dynamic analysis platforms (e.g., DECAF) should be relatively straightforward to port. An analysis plugin can extract any feature that would have been available at the time of the original recording; hence, the features available from a replay are a *superset* of those collected by traditional sandboxes—from the replay, we can re-derive network traffic logs, videos of what was on screen during the recording, system call traces, etc.

Although some of our analyses can be relatively expensive, an additional benefit of a replay-based system is that a corpus of recordings acquired over a long period can be replayed in a much shorter time by simply running each replay in parallel. There is also a practical benefit of replay here: whereas most cloud providers and HPC clusters are unwilling to run live malware samples, *replayed* malware executions cannot interact with the outside world and are hence safe to run anywhere.

For the case studies presented in this paper, we used New York University's "Prince" HPC cluster; processing the full corpus of 66,301 samples took between two days (for a bare replay with no analysis plugins) and eight days (for the unique basic block and malwords analyses). A rough calculation suggests that

at the time of this writing, Amazon EC2 `m4.large` spot instances could be used instead at a cost of around $200 (USD); this seems easily within reach of most research groups.[2]

A final benefit of offline analysis via replay is that *multi-pass* analyses can be created. In cases where some analysis is too expensive to be run on every sample, we can often find some more lightweight analysis that selects a subset of samples where the heavier analysis is likely to bear fruit. For example, in our kernel malware case study (Sect. 4) we initially build a coverage bitmap that tracks what kernel code is executed in each replay. This analysis is cheap but allows us to quickly identify any replays that executed code outside of a kernel module that was present before we executed the malware. Once we have identified this (much smaller) subset, we can then apply more sophisticated and expensive analyses, such as capturing full-system memory dumps and analyzing them with Volatility.

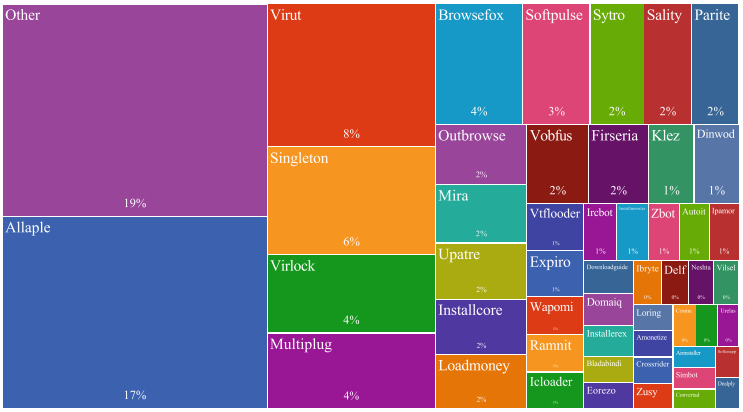


Fig. 2. The distribution of malicious samples.

3 Dataset

Our dataset consists of 66,301 full-system PANDA recordings captured at between December 7, 2014 and December 3, 2016. The samples are provided in a daily feed by the Georgia Tech Information Security Center (GTISC) and come from ISPs and antivirus vendors; from this feed we randomly select 100 executables per day.

The total number of instructions executed across the entire corpus is 1.4 quadrillion. Each recording contains, on average, 21 billion instructions, but there is significant variance among the samples: the standard deviation is 18

[2] Note that this calculation does not include the cost of storing the recordings on some EC2-accessible storage medium such as Elastic Block Store.

billion instructions, with a min of 4 billion instructions and a max of more than 330 billion instructions. The vast majority (all but 261 recordings) are of 32-bit samples. This makes sense, since on Windows, 32-bit samples can run on a 64-bit system but not vice versa.

Considering only the portions of each trace where the malware or some derived process was executing, each sample executes an average of 7 billion instructions with a standard deviation of 16 billion. Among these, 3,474 (5.2%) execute no instructions in the malware itself, indicating that they crashed on startup or were missing some necessary components.

In the remainder of this section we describe notable features of the dataset: the amount unique code, the prevalence of evasive malware, and the number and distribution of families (as determined by antivirus labels).

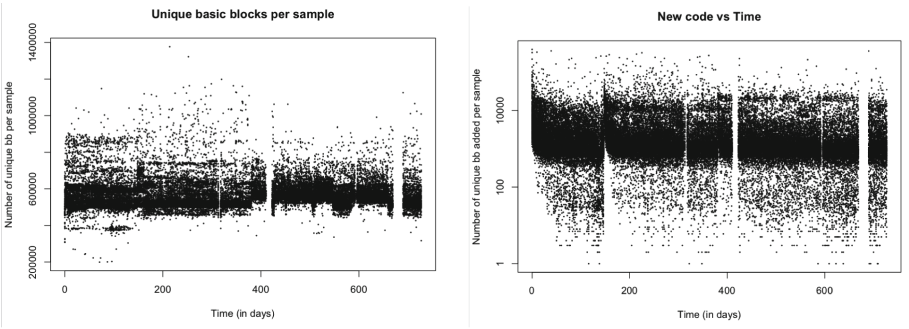
3.1 Unique Code Executed

PANDA replays of malware allow us to accurately measure exactly how much new code each malware sample actually represents. By this we mean the actual code that executes, system-wide, dynamically, when the malware sample is activated. We can use PANDA recordings to perform a detailed analysis of this aspect of the MALREC corpus.

Each recording was replayed under PANDA with the `ubb` plugin enabled, which performs an accounting of the code actually executed, at the basic block level. The plugin considers each basic block of code immediately before it is to be executed by the emulator. If the block has not been seen before in this replay, then we add it to a set, Ubb_i, the unique basic blocks for malware sample i. For every novel block encountered, we attempt to *normalize* it to undo relocations applied by the linker or loader. We use Capstone [27] to disassemble each basic block of code, considering each instruction in sequence, and then zero out any literals. This should permit comparison both within and between replays for different malware samples that are really running the same code.

In addition to the set of unique basic blocks in a replay (and thus associated with a malware sample), we also maintain the set of unique, normalized basic blocks. At the end of each replay, the set of normalized basic blocks, Nbb_i, for malware i, is marshaled. After collecting Nbb_i for each malware sample, the results were merged, in time order, to observe how much new code is added to a running total by each sample.

The number of unique (not normalized) basic blocks identified in a recording, $|Ubb_i|$, is plotted as a function of time in Fig. 3a. The average number of unique blocks in a sample is around 553,203 and it seems fairly stable across all 66,031 samples. This is a lot of code per sample, and it is in stark contrast to the number of new blocks of normalized code we see for each additional sample, as depicted in Fig. 3b. The average, here, is 3,031 new blocks per sample. In addition, we seem to see two similar regimes, the first from 0 to 150 days, and the other from 150 to 320 days. These two start with higher average per-sample contribution, perhaps close to 10000 blocks, and gradually relax to a lower value. The final average (at the far right of the plot) seems to be around 2000 blocks of new (normalized) code per additional sample.



(a) (b)

Fig. 3. A plot of the number of unique basic blocks of code per sample recording, Ubb_i, as a function of time, both before (a) and after (b) normalization.

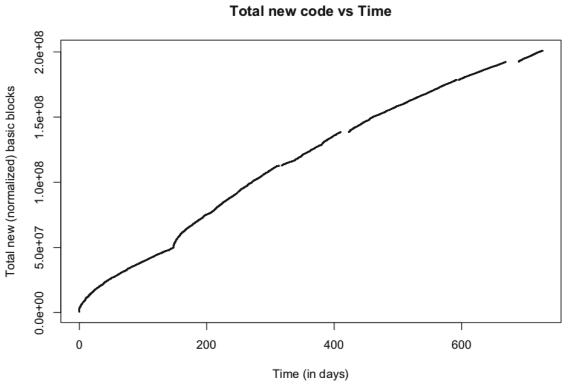


Fig. 4. A plot of the new code contribution for each malware sample, considered in time order. New code is the number of not seen before basic blocks, after normalizing to remove the effect of code relocation

The total amount of novel code seen so far in the MALREC corpus is plotted in Fig. 4. Time, in days, is on the horizontal axis, and on the vertical is the total accumulated number of unique, normalized basic blocks. Thus, e.g., after 200 days, we have about 75 million unique, normalized basic blocks of code; after 400 days we have about 130 million basic blocks, and at the end of the MALREC corpus, at about 727 days, we have collected over 200 million unique normalized blocks of code. We can see here the same effect observed in Fig. 3b: there appear to be two distinct regions, one from 0 to 150 days, and another beginning at 150 days. Both regions have similar shape, with the slope at the beginning higher than later in the region. Over time, both appear to seek a linear (but not horizontal) asymptote.

We investigated the cause of the jump at the 150 day mark, and found that this corresponded to the date we made a configuration change to the VM image

and took a new snapshot (specifically, we made a configuration change to disable mouse pointer acceleration so that our GUI actuation could accurately target on-screen elements). The jump in new code, then, is likely not related to any feature of the incoming malware samples but rather an artifact of the recording environment.

3.2 Evasiveness

A concern with dynamic malware analysis is that samples may detect that they are being analyzed and refuse to show their true behavior. With MALREC, this may happen only during the recording phase, since the behavior during replay is deterministically equal to the one previously registered, regardless of the analysis plugins employed during replay. Although some previous work [4] has attempted to uncover evasive behavior in malware by running it in both a sandbox and on a reference (non-emulated) system and comparing the resulting traces, here we attempt a more difficult task: measuring the prevalence of evasive malware from a single trace of each sample. In this initial work, we do not completely succeed in characterizing all evasive behavior, but we are able to provide an estimate for the lower and upper bounds on the number of evasive samples by checking for known evasion techniques and clear signs of malicious behavior, respectively.

We first checked the execution traces of our samples against a few well-known, recognizable indicators of emulation awareness. In particular, we looked for Windows Registry accesses to keys related to the configuration of virtual machines:

```
HARDWARE\\ACPI\\DSDT\\VBOX__
HARDWARE\\ACPI\\FADT\\VBOX__
HARDWARE\\ACPI\\RSDT\\VBOX__
iSOFTWARE\\Oracle\\VirtualBox Guest Additions
```

We also kept track of registry queries containing key values like `VMWARE` and `VBOX`. Finally, we checked whether the malware ever executed the `icebp` instruction. This is an undocumented instruction, originally intended for hardware debugging, which is known to be handled incorrectly by QEMU and is therefore almost exclusively used as a "red pill" to detect emulation. Using these criteria, we identified 1,370 samples (approximately 2% of our dataset) which showed strong signs of sandbox awareness.

To obtain the upper bound, we compared the reports obtained from VirusTotal[3] behavior API with the analysis logs extracted from our dynamic analysis environment. We focused on keeping track of the actions on the file-system (written files) and the processes created by each sample. We also filtered out those samples which we could not clearly identify as malicious and assign a malware family label to (Sect. 3.3). Comparing those data sources we were able to identify 7,172 samples (10.81% of the dataset) showing the exact same written files and created processes both in the VirusTotal sandbox and in our platform. This

[3] https://www.virustotal.com.

allows us to conclude that those samples are unlikely to be evasive. This kind of approach, it must be noted, does not provide information regarding those samples which are able to evade VirusTotal, and may or may not evade MALREC.

We conclude that evasive malware makes up between 2% and 87.2% of our dataset. We acknowledge that these are not particularly tight bounds, but they can be improved over time with help from the research community to approach a full, ground-truth accounting of evasive malware and its sandbox avoidance techniques.

3.3 Malware Labeling

The labeling of malicious software samples is a long-standing problem in malware categorization tasks. Since our dataset is too large to be manually analyzed, we rely on labels provided by antivirus (AV) scanners from VirusTotal Using AVs labels as a reference classification, poses some well known challenges. Different AV vendors use different naming schemes with proprietary label formats, multiple aliases for the same software classes, generic names (e.g., "Downloader"), and often conflicting identification results. The ubiquity of this problem has encouraged an abundance of research efforts [15,19,22,28]. For the analyses in this paper we employed the AVClass tool [28] to perform malware family name de-aliasing, normalization, and plurality voting amongst the retrieved labels. AVClass assigns a unique label to 58,187 samples belonging to 1,270 distinct families in our dataset. Figure 2 shows the distribution of samples over malicious families.

4 Case Study: Kernel Malware and Privilege Escalation

Kernel mode malware is a powerful threat to the security of end users' systems. Because it runs at the same privilege level as the operating system, it can bypass any OS access control mechanisms and monitoring. Existing malware sandbox systems are not well-suited to analyzing malware with kernel mode components because their instrumentation focuses on system calls or higher-level user mode APIs. By contrast, the recordings in our data set capture the whole-system execution, and so we can perform analyses on any kernel-mode code. In addition, because of the lack of instruction-level granularity, current sandboxes cannot analyze privilege escalation exploits that may allow malware to load its kernel mode components in the first place.

We performed an analysis aimed at detecting any kernel mode malware in the MALREC dataset. We created an analysis plugin, `kcov`, which used a bitmap to shadow each byte of kernel memory and track whether it was ever executed. This allows the kernel code coverage of all replays that start from the same snapshot to be compared; furthermore, we can use introspection (e.g., Volatility [33]) to map the list of loaded kernel modules on the uninfected image and check whether any code outside a known module was executed. Any code executed outside of these

kernel modules must therefore be a results of the malware's activity (though, as we will see, it does *not* necessarily mean that those modules are malicious).

Comparing kernel code executed to the list of modules loaded before the malware infection yielded 574 recordings that might contain kernel malware. We then systematically examined the replays to determine whether the kernel code was malicious and how it achieved kernel-mode execution.

Of our 574 candidates, 71 did not contain malicious kernel code. The unknown kernel code could be attributed, in these cases, to side effects of normal operation, such as starting a legitimate user-land service with a kernel-mode component, loading a signed driver, installing a new font, etc. In some cases it is possible that a benign driver is being loaded for malicious purposes. For example, one sample installed the `tap0901.sys` tap driver bundled with OpenVPN; while this is not malicious by itself, some older versions contain security vulnerabilities that could then be used to load further unsigned code into the kernel.

Malicious kernel code is loaded in a variety of different ways. Because our replays allow us to reproduce instruction-level behavior, we can potentially identify new techniques used by kernel malware. The load mechanisms we found are listed in Table 1. The most common technique by far is to simply call `NtLoadDriver`; because our samples are run with administrator privilege and the 32-bit version of Windows 7 does not enforce driver code signing, this is a straightforward and successful way to load a kernel module. More exotic techniques were also used, however: four samples modified the Windows Registry keys that determine the driver used for the built-in high definition audio codec, causing the rootkit to be loaded.

Table 1. Load techniques of kernel-mode malware

Technique	Count
`NtLoadDriver`	497
Replace legitimate driver	4
Kernel exploit	2

The final and most interesting technique for loading malicious code was to exploit a vulnerability in the Windows kernel. We found just two samples in our dataset that used this technique; both exploited MS15-061, a vulnerability in `win32k.sys`, which implements the kernel portion of the Windows GUI subsystem. It is surprising that this technique is used at all: all samples are executed with administrator privileges, so there is no reason for malware to attempt kernel exploits. This indicates that the malware samples in question simply do not bother checking for more straightforward means of achieving kernel code execution before launching an exploit.

Overall, we find that rootkits are very uncommon in our dataset, and most achieve kernel privileges using standard APIs. We note, however, that these analyses would be difficult to perform with traditional sandbox systems, which

typically monitor at the system call boundary and may miss kernel code. In particular, we know of no existing sandbox analysis that can capture kernel mode exploits, but our system can as a byproduct of reproducing full system executions. In the future, we hope to run more samples *without* Administrator privileges and then apply analyses such as checking for violations of system-wide Control Flow Integrity [3,26] to detect exploits automatically.

5 Case Study: Classification with Natural Language Features

In this section, we introduce a novel approach to the task of malware behavioral modeling for the identification of similar samples. Our guiding intuition is to model malware behaviors as textual documents by isolating textual features (natural language words) from the byte content of memory accessed during the sample execution. Our goal is to effectively transform the problem of classifying malware samples into a task which could be tackled with techniques derived from the well studied domains of text mining and document classification. The overwhelmingly high percentage of code reuse in malicious software discovered in the wild [30,31,36] will lead to a high occurrence of the same words in malware samples belonging to the same families.

5.1 Acquiring the Features

We first identified the subsets of each whole-system trace where the malware was actually executing. To do so, we monitored system calls and tracked:

- the original malware process;
- any process directly created by a tracked process;
- any process whose memory was written by another tracked process (in order to catch instances of code injection attacks).

For each of these processes, we record the portion of the replay (indexed by instruction count) where the process was running.

Next, we replayed our corpus with a plugin that monitored memory reads and writes and checked for byte sequences that matched an entry in our wordlist. We initially used the technique reported in [12] but found that it was unable to scale to large wordlists. Instead, we built an optimized string search plugin that first creates an index of valid four-byte prefixes from the wordlist and stores them in a hash set; because the vast majority of memory writes will not match one of our prefixes, this allows us to quickly ignore memory writes that cannot possibly match. If the prefix does match, then we proceed to look up the string in a crit-bit tree (a variant of a Patricia Tree) and update a count for any matching word. We also perform normalization by converting each character to its uppercase equivalent, ignoring punctuation characters, and skipping NULL bytes (this last normalization step allows us to recognize the ASCII subset of UTF-16 encoded strings, which are common on Windows). The optimized version

of our plugin imposes a relatively small (40%) overhead on the replay and easily handles wordlists with millions of entries.

Our wordlist consists of words between 4 and 20 characters long appearing in the English version of Wikipedia, a total of 4.4 million terms. Wikipedia was chosen because it encompasses both generic English words and proper nouns such as names of companies, software, etc.

5.2 Preprocessing

The *bag-of-words* representation, thus obtained, uses a vectors of dimension $N_{w,d}$ for each sample. Across the whole dataset, the total number of unique words is around 1.4 million. These high-dimensional feature vectors are computationally expensive to manage and often provide a noisy representation of the data. We therefore performed several preprocessing steps to reduce the dimensionality of our data. We first removed all words seen in a baseline recording of a clean, operating system only, run of the analysis platform, which allowed us to avoid words not correlated with the actual malicious behaviors. Next, we eliminated words found in more than 50% or less than 0.1% of the corpus (D). In the former case, a word that is present in a huge number of samples cannot be characteristic of a single family (or of a small group of families) and hence does not provide any valuable information. At the same time, very rare words, found only in a insignificant fraction of samples, would not provide a clear indication of affinity with a specific class. Finally, we removed strings which were purely numerical or contained non-ASCII characters, since these features are less likely to be interpretable. This procedure lowered the dimensionality of our feature vectors to ≈460,000.

Rather than dealing with raw word counts, information retrieval typically assigns each word a Term Frequency Inverse Document Frequency (TF-IDF) score, which evaluates the importance of a word in relation to a document collection. Essentially, a higher score implies that the word appears many times in a small fraction of documents, thus conveying a high discriminative power. Conversely, a low score indicates that the word appears either in very few or very many documents. TF-IDF is computed for each word w and document d as:

$$tf(w,d) = a + (1-a)\frac{freq(w,d)}{max_{w\prime \in d}(freq(w\prime,d))},\ a = 0.4 \tag{1}$$

$$idf(w,D) = \log \frac{|D|}{|d \in D : w \in d|} \tag{2}$$

$$tfidf(w,d) = tf(w,d) \cdot idf(w,D) \tag{3}$$

Because the vast majority of memory accesses consist of non-textual data, some shorter words will match by coincidence. To adjust for these "random" matches, we measured the frequency of each 3-byte sequence in the memory accesses in our corpus. We then compute a likelihood score for each word w as the sum of the logarithms of the frequencies of each 3-gram g composing the word:

$$randlike(w) = \sum_{g \in w} \log(freq(g)) \tag{4}$$

And finally combine the two scores to obtain a single word score:

$$wordscore(w, d) = tfidf(w, d) \cdot (-1) \cdot randlike(w) \tag{5}$$

5.3 Classification Model

Before attempting any classification task, the dataset was divided into 3 subsets: a training set, a test set, and a validation set (which was held out and not used during training), by randomly sampling 70%, 15%, 15% of the elements, respectively. Because machine learning algorithms deal poorly with class imbalance (which is present in our dataset, as seen in Sect. 3), we rebalanced it by undersampling the over-represented classes to a maximum of 1000 samples per class and imposing a hard lower threshold on under-represented families, considering only those categories represented by at least 100 samples. The balanced dataset consists of 28,582 malware samples belonging to 65 families.

Despite the preprocessing described previously, the feature vectors still posed a serious problem due to their high dimensionality. We additionally used Incremental Principal Component Analysis (PCA) to further reduce dimensionality while preserving the greatest amount of variance in the new dataset. Based on empirical experiments on a small subset of the dataset, we chose to reduce each vector to 2048 components for classification.

Our classification model is based loosely on the Deep Averaging Network used by Iyyer et al. [14] for sentiment analysis. The model is a 5-layer feed-forward artificial neural network, with 2048, 1024, 512, 256, and 65 nodes in each layer. A final *Softmax* layer was used for multi-class classification. We implemented the network in Tensorflow [2] and trained it on our dataset. Given the different shape of our input vectors from the ones used in the sentiment analysis context, we performed a classical feature scaling normalization instead of vector averaging. In addition to the regular application of dropout to the hidden layers, in our model the dropout is applied also to the input layer, by randomly setting to zero some of the features, similar the technique suggested by Iyyer et al. [14].

5.4 Results

The Deep Network classifier achieves an overall F1 score of 94.26% on the validation set, and its median F1 score is 97.2%. Figure 5 shows the confusion matrix for the classifier. This is a remarkably positive result considering that malware family classification was performed on such a high number of different classes (65). The average F-Score is lowered by a few isolated families: only 6 are below 0.75 and just 2 below 0.5. The reported results seem to confirm the ability of the model to exploit the peculiarities of the dataset, obtaining a good predictive capacity.

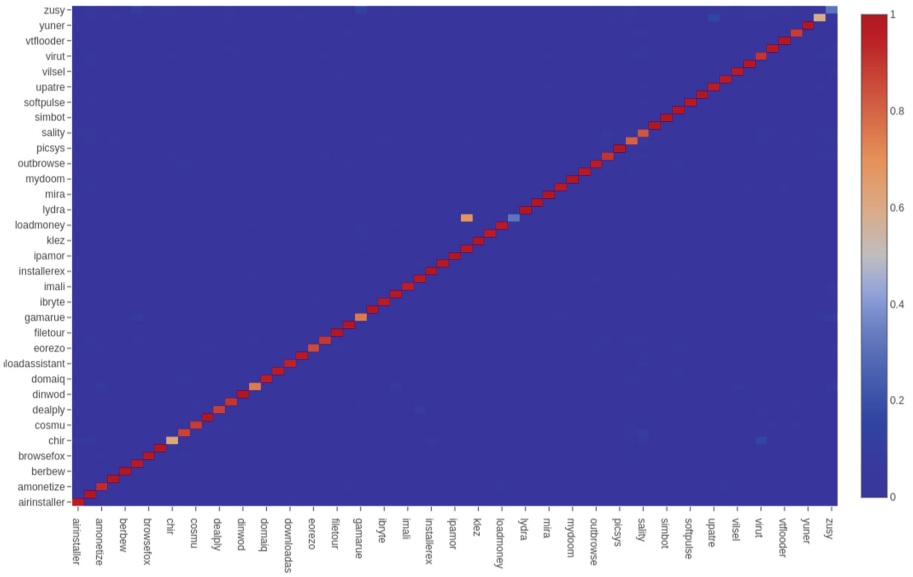


Fig. 5. Confusion matrix for deep network classification of the validation set.

5.5 Textual Search in Malware Samples

In order to further show the usefulness of this approach to the characterization of malicious software behavior, we experimented with employing a full-text search engine to explore the natural language feature data. Using Elasticsearch,[4] we indexed the words recovered during execution. This enabled us to efficiently retrieve the details of malicious programs by specifying particular strings.

We performed two kind of queries. First, we looked for relevant strings regarding the 20 most represented malware families, found in reports and white-papers by antivirus vendors. The ransomware Virlock is clearly identifiable by the strings included in the reports, with the word "bitcoin" appearing in 1,420 samples, of which 74% are Virlock. Other classes like Virut, Sality and Upatre showed words which appeared in samples of those families around 15% of the times. On the other hand, some words which were found in a large number of samples of different families were also found inside almost all the samples of a specific family. For instance, the word "mira", found in 24,000 samples, appeared also in each of the 1,109 samples of the class Mira.

We also searched for a list of 10 well-known antivirus vendors and U.S. bank names. In this case we noted that the Virut[5] family showed a very high propensity to contain names of AV products. Words like "McAfee", "Sophos", "Norton", and "Symantec" were found in multiple samples belonging to that family (from 15% to 37% of the retrieved samples). Bank names, instead, were often found in

[4] https://www.elastic.co/products/elasticsearch.

[5] Virut is a malware botnet that spreads through html and executable files infection.

different families, with "Wells Fargo" appearing prevalently in Firseria samples, "Paribas" in Lydra, and "Barclays" in Lamer.

We plan to make this search engine available as a public interface to our dataset in order to help researchers quickly locate samples of interest.

6 Related Work

Given the relevance of the threat posed today by malicious software distributed over the Internet, a wide number of different attempts have been made at studying it. Two main approaches exists for the analysis of software samples: static analysis, where the binary file is thoroughly inspected but not executed, and dynamic analysis where the unknown program is run in an instrumented environment in order to acquire information regarding its behavior.

Static analysis, in theory, allows a thorough, and complete, exploration of the behavior of the analyzed sample, and is often more cost-efficient than dynamic approaches. There are, however, two definitely relevant obstacles to static malware analysis: obfuscation and packing [23]. Packing is the practice of hiding the real code of a program through possibly multiple levels of compression and/or encryption. Obfuscation is the process of taking an arbitrary program and converting it to a new program, with the exact same functionality, that is unintelligible, by some definition of that characteristic [5]. Due to the still unsolved nature of those issues, in this work we focused our attention on dynamic analysis platforms.

6.1 Dynamic Analysis Platforms

Over the years a noticeable wealth of research work has been produced in the field of dynamic malware analysis systems. Instrumentation of user level programs at runtime was firstly conceived as a technique for profiling, performance evaluation and debugging. It has been deeply studied with systems like Pin [20], Valgrind [24] and DynamoRIO [7]. Those systems, while achieving good performance and enabling heavyweight analyses, were limited to single user level processes. Successively, in the late 2000s, dynamic analysis platforms aimed at the identification of malicious software started to become common.

A remarkable example of one of such platforms, which allowed the collection of whole system execution trace, was Anubis [21]. Anubis was based on the QEMU emulator and is now discontinued. The platform was employed in [6] which represented a milestone in research regarding automated malware analysis. The proposed approach abstracted system calls traces with the definition of a set of operating systems objects, useful to avoid the high variability of traces describing similar behaviors, and performed the clustering operation using Locality Sensitive Hashing, which allowed the system to scale easily with large datasets.

A different technique was proposed with Ether [10], which leveraged hardware virtualization extensions to eliminate in-guest software components, in

order to appear as transparent as possible to the analyzed malware sample. This approach was specifically targeted at thwarting sandbox evasion attempts from virtualization-aware malicious programs. Another, distinct, method is the one found in BitBlaze [29], by Song et al. which fuses together elements of static and dynamic analysis. The structure of this complex system was composed by three main elements: Vine, a static analysis tool providing an intermediate language for assembly; TEMU, built on the QEMU whole system emulator, which provided the dynamic analysis capabilities; Rudder for on-line dynamic symbolic execution. The maintenance of all these platforms seems to have ceased.

Other relevant dynamic analysis systems are still currently actively used in academia and industry contexts. CWSandbox [34] was developed in 2007 to track malicious software behavior through API hooking. Contrary to Anubis, CWSandbox was designed to use virtualization, instead of CPU emulation. Cuckoo Sandbox[6] is another widely used automated malware analysis environment, which powers the on-line analysis service Malwr.[7] Similar to its predecessors, Cuckoo produces comprehensive reports containing information regarding: system call traces, network and file-system activity, screenshots, and memory dumps.

All the platforms mentioned above, however, lack the flexibility provided by the incremental and retrospective analysis capabilities of MALREC. These advanced functionalities are granted by the use of PANDA [11] (Platform for Architecture-Neutral Dynamic Analysis) which implements the concept of deterministic replay to allow the recording, compressed storage, and unmodified replay of full-system execution traces. In the next subsection we will look with particular attention at those sandbox systems which allow the recording and instrumented replay of execution traces.

6.2 Deterministic Replay Systems

Usage of the record and replay methodology was firstly introduced for debugging purposes by LeBlanc and Mellor-Crummey [18] to tackle non-determinism in execution flows. Different solutions to the problem of providing deterministic executions of computer programs in presence of nondeterministic factors were developed over time. A comprehensive overview of those methodologies is provided by [8]. A particularly interesting examples of these techniques, is the use of time-traveling virtual machines (TTVM) for operating systems debugging [17], which allowed whole system recording. VMWare also used to support the Record and Replay feature in its Workstation products [32]. This capability, however, was removed in the Workstation 8 release.

Pioneering work in the field of recording and replaying whole system traces for intrusion detection was provided by ReVirt [13] by Dunlap et al. The Aftersight project [9], in 2008, was the first to apply the approach of decoupling heavyweight trace analyses from the actual sand-boxed execution. The architecture if this

[6] https://cuckoosandbox.org/.
[7] https://malwr.com/.

system was based on two main components: a recording step executed inside a VMWare virtualized environment, and a replay phase on a system emulated through QEMU to enable deep instrumentation.

The concept of decoupled dynamic analysis was further expanded in V2E [35] by Yan et al. V2E exploits hardware virtualization to achieve good recording performance and software emulation to support heavily instrumented analyses. The presented implementation, which conceptually resembles that of Aftersight, is based on the use of Linux KVM (Kernel Virtual Machines) during the recording phase and the TEMU emulator to enable heavyweight analyses during replay.

7 Conclusion

Automated malware analysis systems have become increasingly crucial in dealing with the triage of computer attacks. In this paper we introduced a novel sandbox platform, MALREC that overcomes several shortcomings of traditional malware analysis systems by leveraging high-fidelity, whole-system record and replay. MALREC enables the development of complex, iterative analyses that would be infeasible in standard dynamic analysis platforms. We also introduced a new dataset of 66,301 full-system recordings of malicious software. This dataset, along with accompanying documentation, can be found at:

http://giantpanda.gtisc.gatech.edu/malrec/dataset

We presented two case studies based on this dataset which highlight the usefulness of whole-system record and replay in malware analysis. In the first, we comprehensively catalog the kernel-mode malware present in our dataset. We discovered 503 samples which loaded malicious kernel modules using 3 different techniques, including exploiting a vulnerability in the Windows kernel for privilege escalation. The second analysis takes advantage of the ability to monitor every memory access without disrupting the execution of the malware by extracting fine-grained features based on the natural language words read from or written to memory during execution. We then showed that we could employ those features to train a Deep Neural Network classifier that achieved a global F1-Score of 94.2%.

It is our hope that this system and dataset will enable future research in dynamic malware analysis by providing a standard benchmark and a large supply of test data. This will allow researchers to directly compare the results of competing dynamic analyses, reproduce each others' work, and develop deep, retrospective analyses of malware.

Acknowledgments. We would like to thank our anonymous reviewers for their helpful feedback, as well as Paul Royal and the Georgia Tech Institute for Information Security and Privacy for their help in obtaining malware samples for MALREC. Funding for this research was provided under NSF Award #1657199.

References

1. Volatility command reference - GUI. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Gui
2. Abadi, M.N., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv: 1603.04467 [cs], March 2016
3. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communications Security (2005)
4. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G.: Efficient detection of split personalities in malware. In: NDSS (2010)
5. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_1
6. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: NDSS, vol. 9, pp. 8–11. Citeseer (2009)
7. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: International Symposium on Code Generation and Optimization, CGO 2003, pp. 265–275. IEEE (2003)
8. Chen, Y., Zhang, S., Guo, Q., Li, L., Wu, R., Chen, T.: Deterministic replay: a survey. ACM Comput. Surv. **48**(2), 1–47 (2015)
9. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 1–14 (2008)
10. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 51–62. ACM (2008)
11. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with PANDA. In: Program Protection and Reverse Engineering Workshop (PPREW), pp. 1–11. ACM Press (2015)
12. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: mining memory accesses for introspection. In: ACM Conference on Computer and Communications Security (CCS), pp. 839–850. ACM Press (2013)
13. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. **36**(SI), 211–224 (2002)
14. Iyyer, M., Manjunatha, V., Boyd-Graber, J., Daumé III, H.: Deep unordered composition rivals syntactic methods for text classification. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (vol. 1: Long Papers), pp. 1681–1691 (2015)
15. Kantchelian, A., Tschantz, M.C., Afroz, S., Miller, B., Shankar, V., Bachwani, R., Joseph, A.D., Tygar, J.D.: Better malware ground truth: techniques for weighting anti-virus vendor labels. In: ACM Workshop on Artificial Intelligence and Security (AISEC), pp. 45–56. ACM Press (2015)

16. King, S.T., Chen, P.M.: Backtracking intrusions. ACM Trans. Comput. Syst. (TOCS) **23**(1), 51–76 (2005)
17. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, p. 1 (2005)
18. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. **36**(4), 471–482 (1987)
19. Li, P., Liu, L., Gao, D., Reiter, M.K.: On challenges in evaluating malware clustering. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 238–255. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15512-3_13
20. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Notices, vol. 40, pp. 190–200. ACM (2005)
21. Mandl, T., Bayer, U., Nentwich, F.: ANUBIS ANalyzing unknown BInarieS the automatic way. In: Virus Bulletin Conference, vol. 1, p. 2 (2009)
22. Mohaisen, A., Alrawi, O.: AV-meter: an evaluation of antivirus scans and labels. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 112–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08509-8_7
23. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 421–430. IEEE (2007)
24. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN Notices, vol. 42, pp. 89–100. ACM (2007)
25. Newsome, J.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed System Security Symposium (NDSS) (2005)
26. Prakash, A., Yin, H., Liang, Z.: Enforcing system-wide control flow integrity for exploit detection and diagnosis. In: ACM SIGSAC Symposium on Information, Computer and Communications Security (2013)
27. Quynh, N.A.: Capstone: Next-Gen Disassembly Framework. Black Hat USA (2014)
28. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
29. Song, D., et al.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7_1
30. Tian, K., Yao, D., Ryder, B.G., Tan, G.: Analysis of code heterogeneity for high-precision classification of repackaged malware. In: 2016 IEEE Security and Privacy Workshops (SPW), pp. 262–271, May 2016
31. Upchurch, J., Zhou, X.: Malware provenance: code reuse detection in malicious software at scale. In: 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), pp. 1–9, October 2016
32. VMWare: Enhanced Execution Record/Replay in Workstation 6.5, April 2008
33. Walters, A.: The Volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility

34. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. **5**(2), 32–39 (2007)
35. Yan, L.K., Jayachandra, M., Zhang, M., Yin, H.: V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis. ACM SIGPLAN Not. **47**(7), 227–238 (2012)
36. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109, May 2012