Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes

Alexander Küchler*, Alessandro Mantovani†, Yufei Han‡, Leyla Bilge‡ and Davide Balzarotti†
* Fraunhofer AISEC, Germany. alexander.kuechler@aisec.fraunhofer.de
† EURECOM, France. {alessandro.mantovani | davide.balzarotti}@eurecom.fr
‡ NortonLifeLock Research Group, France. {yufei.han | leyla.bilge}@nortonlifelock.com

***Abstract*—The amount of time in which a sample is executed is one of the key parameters of a malware analysis sandbox. Setting the threshold too high hinders the scalability and reduces the number of samples that can be analyzed in a day; too low and the samples may not have the time to show their malicious behavior, thus reducing the amount and quality of the collected data. Therefore, an analyst needs to find the 'sweet spot' that allows to collect only the minimum amount of information required to properly classify each sample. Anything more is wasting resources, anything less is jeopardizing the experiments.**

Despite its importance, there are no clear guidelines on how to choose this parameter, nor experiments that can help companies to assess the pros and cons of a choice over another. To fill this gap, in this paper we provide the first large-scale study of the impact that the execution time has on both the amount and the quality of the collected events. We measure the evolution of system calls and code coverage, to draw a precise picture of the fraction of runtime behavior we can expect to observe in a sandbox. Finally, we implemented a machine learning based malware detection method, and applied it to the data collected in different time windows, to also report on the relevance of the events observed at different points in time.

Our results show that most samples run for either less than two minutes or for more than ten. However, most of the behavior (and 98% of the executed basic blocks) are observed during the first two minutes of execution, which is also the time windows that result in a higher accuracy of our ML classifier. We believe this information can help future researchers and industrial sandboxes to better tune their analysis systems.

I. Introduction

Malware analysis sandboxes play a fundamental role in the analysis of suspicious software. The importance of these tools has brought to a proliferation of different platforms, resulting in a large number of both open source and commercial sandbox solutions [1]–[3], [9]. Moreover, fifteen years of research in the field has covered a wide range of technical aspects and proposed new solutions dedicated to the dynamic analysis of malicious samples [19], [34], [53], [83], [87], [101], [106].

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24475
www.ndss-symposium.org

However, despite the fact that malware analysis sandboxes are a well-studied and mature technology, little is known about the best configuration setup that is required to maximize their effectiveness. This is true even for their most simple parameter: the sample execution time. Over the years, companies running fully automated analysis infrastructures kept decreasing the analysis time to cope with the increasing number of collected samples. While this may seem an obvious response to the big data problem, there is no study that measured the impact of less analysis time on both the *amount* and the *quality* of the information collected by the sandbox; How much do we really lose by going from ten minutes per sample to three minutes? And then, from three to one?

At first, this might seem like a classic trade-off between the volume of the collected data and the amount of samples that can be analyzed in a given amount of time. However, the answer is actually much more complex than that. In fact, it boils down to two fundamental aspects. First, on how the actions performed by a malware sample are distributed over time: does the malicious behavior of a program start from the first second, or can it be observed only after a few minutes of execution? Second, it depends on how important a piece of behavior is for a specific goal. For instance, the fact that a short execution time might only expose 30% of the behavior of a sample, might not necessarily be a bad thing. It all depends on whether that 30% is sufficient to correctly classify the sample or to obtain adequate information for the analysts. Intuitively, longer execution time will result in the collection of more events, which *may* contain useful information to characterize the behavior of a sample. As a result, one would logically expect a time window with increasingly longer temporal coverage to provide better classification results. Sadly, none of these two factors (i.e., the evolution of the malicious behavior over time and the relevance of the collected information) have been studied before. This, as we will show in Section II, has left analysts and researchers without clear guidelines. As a result, everyone was free to pick his own value—mostly based on gut feelings on what they believed to be a "reasonable" execution time.

These problems, and the importance of designing a malware analysis platform based on real data, motivated us to perform a comprehensive study on the subject. By using a custom-designed solution based on the PANDA record-replay emulator [31], we collected fine-grained information on the execution of 100K samples, and use it to shed light on how time affects both the volume and the relevance of the collected data.

Our key findings, summarized in more details in Section VIII, show that most of the samples execute either for less than two minutes or for more than ten minutes. In both the cases, we could observe that while the number of system calls tends to increase linearly over time, the code responsible for them is typically explored very fast – in the first one or two minutes of execution. Our measure of the impact of stalling code in its traditional form (i.e., invocations of a `sleep` API) shows that it actually influences in a remarkable way only a low percentage of samples (close to the 3%). Furthermore, we experimented with different code-based metrics to estimate the absolute code coverage that samples reach while running in a sandbox. Thanks to these measurements, we illustrate that absolute code coverage of a sample can have a great variance, depending on its family and on the sample itself. Overall, we registered code coverage values in the range of 10% to 40%.

Finally, we implemented a state-of-the-art machine learning classifier and conducted experiments to measure how unique and how relevant the data collected in different time windows is. This helps us to answer whether it is easier to tell that a program is malicious by looking at its actions in the first minute, or by looking at those it performs after three or five minutes. Indeed, we found that the first two minutes of execution are the most representative from the perspective of a ML classifier, not just because of the amount of events registered in such a timespan, but also for their ”quality” in terms of novel information that they can provide to a classifier.

We plan to release the entire data we collected from the execution of 100K samples, to help other researchers replicate our findings and conduct further experiments on the nature and evolution of malicious behavior.

II. Motivation and Related Work

In 2007, in their seminal work that proposed the design of the original malware analysis sandbox, Willems et al. [101] noted:

> *“We found that executing the malware for two minutes yielded the most accurate results and allowed the malware binary enough time to interact with the system, thus copying itself to another location, spawning new processes, or connecting to a remote server, and so on.”*

While interesting, this was purely a qualitative statement, and the authors never mentioned what kind of experiments they conducted to support the choice of this threshold. At the other end of the spectrum, in 2011 Rossow et al. [82] executed each sample for up to one hour and noted that only 23.6% of the communication endpoints and 95% of the network protocols were observed in the first 5 minutes of analysis. This seems to suggest that if the goal is to study the network behavior, two minutes are most likely insufficient to collect the majority of the sample’s behavior.

To the best of our knowledge, the only other work that studied the interval of time required to properly analyze malware was performed in 2017 by Kilgallon et al. [51]. To be precise, the goal of the authors was to predict the time needed for a sample to show enough malicious behavior for the heuristics of the Cuckoo sandbox to assign a malicious score of five. To train their model, the authors conducted a small experiment with 3320 samples, and found that 82% of them needed to be executed for less than one minute, and 53% could already provide the required information in their first 20 seconds.

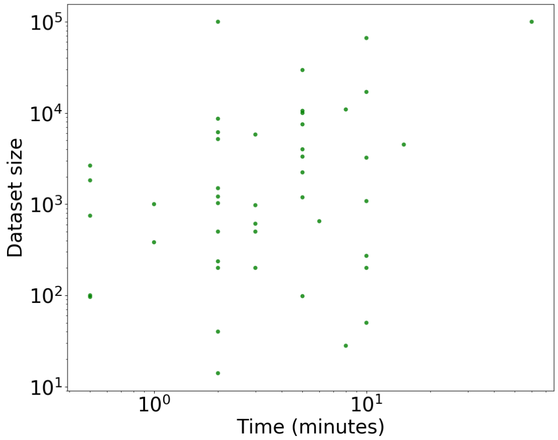

Fig. 1: Execution Time vs Dataset Size

Due to the lack of clear guidelines, over the past fifteen years researchers and security companies adopted a wide range of values for the samples execution time. As we will discuss later in this section, the rationale behind each choice is often unknown, but for sure the main goal of the analyst (for academic researchers) and the amount of samples that need to be analyzed over a limited period of time (for companies) are the two main factors that dictated the choice of one threshold over another.

Researchers tend to focus on relatively small datasets that can be analyzed over the course of many days, typically with the goal of collecting data to study a very specific behavior. Instead, security companies need to operate fully automated malware analysis pipelines to process hundreds of thousands of samples per day, thus focusing more on scalability and on the collection of generic behavioral data.

In the next two paragraphs we provide more details about the time intervals adopted in the experiments conducted by *researchers* and *security companies*.

A. Research Experiments

Researchers have been conducting a large spectrum of dynamic analysis experiments over the past twenty years. Some were explicitly designed to improve or propose new sandbox techniques, while others simply relied on sandboxes to collect data to perform other experiments — such as modeling the behavior of samples, extracting new detection signatures, train a classifier, or report on the internals of certain malware characteristics (such as packing, use of encryption, etc.).

TABLE I: Papers collected by execution time

Time (minutes)	Papers
<1	5 [44]–[46], [94], [109]
1	2 [56], [108]
2	14 [21], [26], [35], [40], [43], [52], [70], [85], [91], [95], [100], [101], [105], [110]
3	7 [15], [65]–[67], [71], [81], [93]
4	1 [58]
5	13 [12], [13], [23], [29], [38], [50], [51], [69], [78], [88], [92], [102], [106]
8	2 [20], [89]
10	7 [24], [25], [36], [41], [59], [63], [87]
15	1 [10]
>15	2 [82], [84]

We reviewed several papers by looking at the execution time threshold used by their authors, and report the results in Table I, grouped in different time ranges. The values range from a minimum of 30 seconds [44]–[46], [94], [109] to a maximum of 1h [82].

It is also interesting to note that it is very rare for a paper to discuss **why** a certain value was chosen. We only found a handful of explanations, including *"We used executions of 120 seconds, as we found that two minutes is generally enough time for most malware to execute its immediate payload, if it has one"* in 2010 [40] or *"The 5 minute window was arbitrarily defined on the assumption that the payload would execute immediately upon execution and would take no longer than 5 minutes to complete"* in 2018 [23]. However, none of the papers conducted experiments to support their choice.

Figure 1 shows the sandbox execution time versus the number of samples analyzed. There is no clear relationship among the two, confirming the fact that the sample execution time chosen by academic researchers is more the result of the personal judgment of the authors than a choice dictated by the size of the dataset.

Finally, it is also interesting to note that the execution time is rarely discussed as an important factor when comparing with previous works. It is common for some papers to extract malware behavior from a two-minute execution and compare the models with previous studies that run samples for longer periods of time. In this case it is unclear whether accuracy degradations or improvements of the proposed techniques are caused by a better solution or simply by the better data collected over a longer period of time.

B. Industrial sandboxes

While researchers often have to perform customized analysis in their experiments, we expect security companies to use more standardized architectures that have been carefully tuned over the years to maximize the trade-off between the required resources and the collected information. Unfortunately, to the best of our knowledge none of the companies disclose any details about their internal pipeline.

Therefore, to collect some data we created a simple probe binary (compiled for Windows) as already performed in the past by Yokoyama et al. in 2016 [107]. The goal of our probe was to make HTTP requests to a server under our control at regular intervals of time (with a granularity of 10 seconds). To distinguish among different executions of the same sample, we included in the request the PID of the process, a random number generated when the probe is executed, and an incremental counter. No other information about the sandbox, the environment, or the network in which the sample is executed was collected by our program.

We first submitted the probe to VirusTotal [8] because it serves as entry point for a large number of security tools and companies. After submitting a file, VT shares it with different Antivirus engines that analyze the file in their own sandboxes. Several companies also receive feeds from new submissions, thus increasing the number of times our probe was analyzed. Even though VT could impose a time limitation to perform dynamic analysis, we observed multiple re-executions of the sample after the initial submission and our results confirm that the majority of the executions happened well after VT displayed its result. In addition to VirusTotal, we also submitted our probe to the list of online sandbox services collected by Yokoyama et al. [107].

A month after the initial submission our probe had been analyzed by 32 different sandboxes. We can summarize the results in three main categories:

- 23 sandboxes ran the sample exactly once for a fixed interval of time, ranging from 30 seconds to 4 minutes.
- 4 sandboxes ran the sample but manipulated the `sleep` invocation to lower the sleeping time. In this case we received all HTTP requests in a burst. The sample was analyzed for a maximum period of 2-to-3 minutes (including the manipulated `sleep` invocations).
- Finally, 5 sandboxes repeated the analysis of our probe multiple times during the month following our initial submission, but always for 2 to 3 minutes. None of these sandboxes altered the `sleep` invocations.

The finding of this experiment is that companies seem to execute samples for an average time that ranges from 30 seconds to 4 minutes. Considering the number of new malware samples that need to be analyzed every day, this is an understandable choice to achieve scalability.

C. Related Work

A huge body of research exists in the field of dynamic malware analysis. In the following, we summarize work aiming at studying common malware techniques and thus at improving the quality of malware sandboxes.

A well-known technique to identify and consequently evade malware analysis sandboxes is the analysis of the execution environment to identify inconsistencies in the behavior of the CPU, missing artifacts of user interaction, additional artifacts in memory or fingerprinting well known malware sandboxes [37], [64], [72], [74], [80], [104], [107]. As a counter-technique, researchers proposed to execute malware samples in different environments and observe differences in the execution traces to identify environment-aware samples [17], [48], [49], [57], [61]. While a successful sandbox detection can alter a sample execution time, these work focused on the techniques – or on the way those techniques can be detected during the analysis.

Another branch of research focuses on increasing the execution coverage by identifying trigger-based behaviors and on enforcing the execution of multiple paths [22], [30], [68], [75], [97]. While these techniques can help to collect a more complete picture of the behavior of a sample, they introduce a very large overhead that make them unsuitable for malware analysis services. Among the solutions to increase coverage, the most relevant to our study are those that focus on mitigating the effect of stalling code [28], [54], [99]. As we mentioned above, some basic form of stalling-code mitigation is deployed by some of the sandboxes we tested and also incorporated in our experiments.

Other work studied the impact of time for successfully launching specific types of attacks [11] which is orthogonal to the goal of our work.

III. Experiment Design

As our goal is to measure the evolution of a sample's behavior over time, it is first necessary to identify how these two dimensions, *behavior* and *time*, can be properly measured. In this section we discuss the different available options and the final choices that guided the design of our experiments.

A. Measuring Runtime Behavior

A common approach to model the behavior of a program is to observe its APIs and/or system calls invocations, as they provide a detailed view of the interaction of the sample with its surrounding environment. However, syscalls are executed throughout the entire life of a program, and just observing new syscalls is not necessarily a sign of observing "new" behavior. For instance, a ransomware sample can open a large number of files, reading and writing their encrypted content over and over again. This translates to, among others, many invocations of `NtOpenFile` with different arguments (as the file name varies). However, all of them are somehow associated with the same high-level behavior, and therefore an analyst does not gain any additional information by observing the creation of yet another encrypted file.

In contrast, a different malware sample may first open the executable of another program (e.g., to infect its code), and then open a number of documents to exfiltrate some private data. In this scenario, the same `NtOpenFile` syscall occurs in completely different contexts and therefore exhibits new behavior that is relevant for the analysis.

To distinguish such cases, in our experiments we will use three orthogonal ways to measure the behavior of a running sample. First, we will look into the list of syscalls and the appearance of new classes of high-level actions (e.g., network traffic, filesystem activity, or registry operations). Second, we will collect the actual code executed in the sandbox, to capture the *context* in which each syscall was executed. To continue with the previous examples; while the `NtOpenFile` syscalls in the ransomware sample likely originated from the same snippet of binary code, the syscalls for infection and exfiltration in the second sample were certainly triggered by different pieces of code. Therefore, a natural way to observe the appearance of new behavior is to examine the execution traces — where new code equals new runtime behavior.

Syscalls and binary code give us a way to capture the *amount* of behavior, but not necessarily its *quality*. In other words, knowing that 80% of the syscalls happen in the first two minutes does not necessarily mean that those system calls are the ones that really capture the core behavior of a sample. The actual parameters of those syscalls can be very important and maybe, while the amount of new behavior decreases with time, its quality increases.

Therefore, we decided to add a third dimension by using a machine learning classifier as a way to capture how important the new behavior is to flag a sample as malicious. The intuition is that, if a classifier achieves higher accuracy by using the information collected after a few minutes of execution, this means that those actions were more distinctive of the behavior of the sample. After all, one of the main goals of malware analysis is to collect enough information to accurately classify an unknown sample.

B. Behavioral Metrics

So far we discussed *which* information we can use to capture the evolution of a sample behavior. Now we want to investigate *how* this information can be measured and presented.

For this, we propose three classes of metrics to use in our experiments. The first, is the **Relative coverage**. The relative coverage of a sample execution at time t is defined as the fraction of behavior (expressed either as the number of observed system calls or as the amount of executed code) with respect to the maximum number that is observed for that sample over the entire execution interval. In other words, the relative coverage captures which *fraction* of behavior we collect at different points in time. For instance, if a sample executes 10K basic blocks in ten minutes, how many of them can we observe if we only execute the program for five minutes? By answering this question, this metric can provide us with an efficient way to understand how much an analyst would lose, in terms of *amount* of collected data, by reducing the execution time in her sandbox. In our experiments we will use two relative coverage metrics: the Relative Syscall Coverage (R_s) and the Relative Code Coverage (R_c).

The second class of metrics we use to measure the behavior evolution is the **Absolute Code Coverage** (A_c)—which tries to estimate how much code was executed wrt the total sample code. The challenge with this metric is that while it is simple to extract the instructions that are executed (as the emulator can easily collect this information), it is difficult to precisely estimate the part that was *not* executed. As we explain in Section V-C, we can do that either by measuring the size of the executable memory regions, or by disassembling those regions to recover individual basic blocks. The first approach may overestimate the code by also including data (if the two are interleaved), while the second suffers from the limitations of the disassembler algorithm (e.g., to recursively traverse the CFG and to deal with possible obfuscation and anti-disassembly tricks). In Section V-C we will show that in our experiments the Absolute Code Coverage computed by looking at raw bytes or by looking at basic blocks are almost identical. Therefore, unless explicitly mentioned, through the paper we will use basic blocks as the basic unit of code.

The third and final metric we use in our analysis relies on the **accuracy of a machine learning classifier** that uses the sequence of the system call records observed during the sandbox execution. By gradually extending the length of the time window and measuring the resulting classification accuracy, we can assess the quality of the data collected over that specific amount of execution time.

C. Measuring Time

As the goal of our work is to analyze the evolution of a sample behavior over time, a crucial step is to devise a way to precisely measure at which time each action was performed.

First of all, we need to define what *time* means in this context. To this end, we first discuss the differences between *measured time* and the *sandbox time*. Our system (explained in details in Section IV) requires an emulator to capture each basic block executed by the malware samples. Therefore, our environment introduces an extra overhead which slows down the execution compared to a more traditional malware analysis sandbox, resulting in alterations on the original execution time. To identify the latency that our system introduced to the measurement, we compared the executions of the same binaries in a Cuckoo [2] sandbox deployed on top of the KVM virtualisation software (which acts as our reference sandbox) and inside our infrastructure. During the executions of the binaries, we fetch the timestamps corresponding to some specific syscalls from both our sandbox and Cuckoo, and compare them to estimate the slowdown.

The limitation of this technique is that the executables under observation have to follow a deterministic pattern when invoking the syscalls. Thus, we crafted two custom binaries in which we control their deterministic syscall invocations (the two binaries differ in the set of syscalls that are invoked). We executed each binary 15 times in our system and 15 times in the vanilla Cuckoo sandbox and obtained an overall slowdown of 3.1x. However, for a real binary this value can be much lower if the program spends a considerable amount of time waiting for external events or inputs (e.g., when receiving network packets). We therefore repeated the test for several samples from our test set that show deterministic behavior but which interact with the filesystem or receive network packets. In these cases, we could observe a minimum slowdown of $1.4x$ and a maximum slowdown of $3.7x$. This is well below the worst-case slowdown of $10x$ of recording with PANDA compared to executing a binary on a native system as reported by the authors [31], [32], [90].

Another key problem while taking measurements about the time, is that several samples invoke the *Sleep* and *SleepEx* functions while running. Very often, malware authors use this to stall execution as an anti-analysis trick. However, because of its popularity, sandboxes can be designed to reduce (or entirely skip) any sleep time — an action which is often called time-warping. As we discuss in Section II-B, 4 of the 32 industrial sandboxes we tested adopted this approach. This raises an important question: if a sample executes one syscall at time t, then waits for one minute and executes a second one, what is the time in which the second event occurs? Is it *real time* (i.e., $t + 60$ seconds), or the *warped time* (i.e., t)?

Our solution, as explained in Section IV-A, is to use time-warp to ignore any sleep operation, to maximize the amount of data we can collect. However, our system is designed to also collect the expected sleep duration, allowing us to virtually re-introduce the proper delays during the analysis. This way, we can precisely assess how important it is to skip sleep operations on the amount of data that can be collected by the sandbox.

D. Sample Selection

Setting up a representative malware dataset is a challenging task that can be performed in different ways depending on what the data needs to closely represent and what the goal of the analysis is. For instance, one may want to balance malicious and benign samples while another may focus on obtaining a broad number of different families or even on balancing the presence of individual attributes in the dataset (such packed vs non-packed). Our goal is to analyze the impact of execution time on a large-scale malware analysis pipeline which might receive a large number of fresh samples to analyze every day. As such, we wanted our dataset to mimic those that are regularly analyzed by security companies. To satisfy this requirement, for our malicious dataset we downloaded fresh samples submitted to VirusTotal, i.e., samples observed for the first time on the same day in which we downloaded and analyzed them (this is what Ugarte-Pedrero et al. [98] call the *catch of the day*). This was the only criteria we used for our selection. We did not try to impose any balance to the downloaded data, because when previously unknown samples are identified, they need to be analyzed without any a priori knowledge about them. However, to avoid biases due to the collection day, we only retrieved our data in small chunks of 2K samples, and repeated the procedure every few days. This way we could guarantee that samples were always analyzed as soon as they were collected, thus maximizing the probability of being still active and therefore of running correctly at the time of the execution. Because we were limited by the scalability of our solution, which required up to one hour to complete the analysis of each sample and to produce elaborated reports about its execution, we stopped our data collection when we reached 100K samples. This took over 6 months at total. Overall, the final instance of the dataset included 86K malicious samples and 14K benign ones. We included all types of Windows PE malware and we did not filter any specific malware families or types.

As mentioned earlier, our study also aims at assessing the impact of the analysis time on machine learning-based malware detection techniques. Clearly, such techniques also require a fully supervised training set composed by well tagged benign and malicious samples, so as to build and train the classifiers to categorize samples as malicious or benign. For this reason, we only selected samples that were identified to be malicious by at least 5 AV detection on VirusTotal–a rather conservative solution compared with the threshold used by other works [60]. Moreover, in contrast to many studies that selected benign samples by picking popular Windows applications or installation files, which in general are very well-known files and therefore easy to spot and whitelist by the security companies, we assembled our benign dataset from VirusTotal submissions. The selection criteria was to choose samples that were never identified as malicious by any of the AV companies and that had been submitted to VirusTotal at least 6 months prior to our analysis. This would provide enough time to the AV to adapt their signatures to cover new families and thus minimize the probability of selecting unknown malicious files as benign.

IV. Design & Implementation

As we explained in the previous section, our study requires us to collect very fine-grained information about the operations performed in the system during the execution of a sample. In particular, we want to collect all system calls, but also to monitor the executed instructions, in the form of individual Basic Blocks (BBs). Moreover, to estimate the absolute code coverage at any given point in time, we also need to extract all the executable regions that are part of the memory of the running processes.

While malware analysis sandboxes are typically executed either on a bare metal machine by using a virtualization technology, for the level of details required by our experiments we need to resort to an emulator (QEMU in our case). Moreover, since our analysis would introduce an unreasonable overhead, we decided to use PANDA [31] to first record a sample execution, and then replay it while using our instrumentation.

This solution hinders the scalability of our system but it allows us to obtain a better visibility on the execution of a program, which is a necessary step to uncover the impact of the sample execution time on the amount of information that could be obtained through the malware analysis sandbox.

A. System Overview

Our analysis starts by loading the sample into PANDA and by recording its execution for a specified amount of time. Based on the data presented in Section II, we decided to use 15 minutes as execution threshold–as this was the highest value we observed in research papers or industrial solutions, with the exception of two longer experiments that only focused on network activity.

However, to allow for a more fair analysis of downloaders and droppers, our system detects samples that spawns new processes based on files they previously wrote to the disk, and in this case we restart our timer when a dropped file is executed. In a real world scenario dropped files could be in principle collected and analyzed separately, which is equivalent to our solution of restarting the clock for dropped files. Overall, our recording system is similar to the one used by Malrec [87] but we adapted the solution to the new PANDA version 2 while the current Malrec dataset is available for PANDA 1 only.

We then replayed the recorded execution while running several plugins dedicated to collect the data required by our analysis. The plugins collect all system calls that occur in the context of the monitored processes, all BBs that are translated and executed by the emulator, as well as a complete memory dump whenever one of the monitored processes is about to terminate. From the memory dumps we then extract the content of the process' memory and finally all of its executable basic blocks.

System Calls Collection – One of our goals is to collect all system calls invoked by the malware sample under analysis. PANDA comes with the `syscalls2` plugin [4] which provides callbacks triggered when a system call is called or when it returns. Based on these callbacks, we implemented our own plugin that hooks every system call that occurs during the execution of the malware sample. The plugin logs all system calls together with the timestamp of their occurrence, the PID of the process in whose context the call occurs, as well as all the syscall parameters. However, PANDA only allows us to monitor the system calls from outside OS and does not make any high-level information available. Therefore, each argument or return value is represented by a generic `uint32_t` value. While this is fine for integer values, in practice, many arguments to system calls are pointers to structures. Therefore, if higher-level information is required for further analysis, we need to parse the memory to retrieve additional information. Unfortunately, large parts of the Windows OS internals are not well documented.

We partially solved this problem by extracting information from Volatility's offset-tables for Windows 7 Service Pack 1 [39][1]. We also used the description provided by Petritsch [76] on how to retrieve network packets from system calls. Finally, PANDA's win7proc plugin [5] infers high-level information from several system calls related to processes, registry, file system, and shared memory. By combining the information provided by all these sources, our system is able to lift data for all system calls related to the interaction with the file system, the registry, as well as for processes, memory management, and network-related operations.

Basic Blocks – The `PANDA_CB_AFTER_BLOCK_TRANSLATE` callback provides a way to instrument the BB translation, an operation that the emulator performs right before the first execution of each BB. With this methodology, we collect all BBs when they are executed for the first time in the context of a process we observe.

To estimate the code coverage achieved by a given execution, we also need to extract all executable BBs belonging to the malware sample. Since, due to packing and obfuscation techniques, it is often impossible to retrieve the program code statically, we resort to extracting the BBs from the process' memory image. It is reasonable to assume that the maximal amount of code is present in memory right before the process terminates. While this might not be the case for packers of type VI (according to the classification of Ugarte-Pedrero et al. [96]), these are extreme, and very inefficient, forms of packing that are used in only 0.2% – 1.8% of the samples. Moreover, it is important to note that we only report the absolute coverage for completeness, while all our measurements rely on relative metrics, which are not impacted by packing as they only measure those basic blocks that are actually executed by the program.

To dump the memory at the end of the process' execution, hooking the `NtTerminateProcess` call was insufficient. Instead, we found that for all possible ways a process has to terminate, the `ProcessDelete` bit (the 3rd bit of Byte 0x270 of the `EPROCESS` structure) is set to 1 right before the memory of the process is freed by the Windows kernel. In contrast, all other bits and timestamps are set after the memory space has already been freed. Hence, our system checks if this

[1] Volatility is a popular memory forensics framework and its operations rely on OS internal details that are retrieved by manually reverse engineering the target systems.

bit is set when the process context changes, thus dumping the memory before it is freed by the operating system.

We then use the Rekall [6] memory forensics framework to extract the memory image of the process and analyze the virtual address descriptor tree of the process' memory content.

By parsing its output, we can extract all regions which are marked as executable but do not belong to any named module (e.g., a shared library). Each one of these regions is then processed by SMDA [7], an open source recursive disassembler optimized for recovering code from memory dumps. SMDA is based on Nucleus [14], which can detect more functions in a binary than traditional disassemblers. We also extended SMDA to better guide it through the code regions within the address boundaries which have been executed in PANDA, thus increasing the disassembler's ability to explore the code.

Recovering Time Information – A problem we encountered in our analysis is that, during replay, we needed a way to retrieve the timing information according to the malware recording. In other words, we wanted to know exactly when a given system call or basic block was executed during the sample recording and not during the, much slower, replay.

To achieve this goal, we exploited the `KUSER_SHARED_DATA` structure and its `SystemTime` field at offset `0x14`. This field stores the current system time (expressed as an offset since January 1st, 1601 00:00:00 in 100 nanosecond ticks) and its value in memory is regularly updated. As PANDA records all memory-writes, any update to `SystemTime` is also recorded. This allows us to recover the exact timestamp when each event was performed during the recording – thus removing the overhead of our system from the results of our analysis.

Windows API Hooks – Several malware samples intentionally delay their malicious activity by sleeping for a certain amount of time, thus bypassing traditional dynamic analysis sandboxes. Therefore, we decided to hook the `Sleep` and `SleepEx` functions in `Kernel32.dll` to detect such cases. Whenever one of the two functions is called, we add the time of the sleep to the system time but return immediately. This allows us to bypass sandbox evasion tricks based on the system time.

Furthermore, we hook the functions `CreateProcessInternalW`, `CreateProcessWithLogonW` and `CreateProcessWithTokenW` to inject our dll and install the hooks also into child processes. In case a file is dropped by the malware and later executed, we want to extend the time of our analysis. We therefore collect all written files by hooking `NtCreateFile`.

All events recorded through the API hooking are immediately transmitted to our analysis framework while the malware is still running. We collect the time the malware sample sleeps and the child processes spawned by the sample. We decided to implement a simple client/server mechanism to share this information. The client is included in the injected DLL and sends messages containing the timestamp, the PID, and the event information to the server running outside PANDA, which logs the information for the subsequent analysis.

TABLE II: Top 10 families in the dataset

Family name	**Total**
sytro	4162
stihat	3936
blackmoon	3911
agen	3574
dinwood	2821
sillyp2p	2806
high	2682
upatre	2613
mira	2505
ulise	2473

V. Results

According to AVClass [86], our dataset contains 806 different malware families and $6,989$ samples classified as singleton (i.e., for which it was not possible to recognize a common label). Even though some families are present with higher frequency than others (Table II reports a ranking of the top 10 families present in the dataset), no family was predominant and even the largest accounted for only 4K samples – thus resulting in a well balanced dataset.

Our experiments were conducted in a sandbox running the 32-bit version of Windows 7 Service Pack 1. We further configured the sandbox to have an internet connection and simulate basic user interaction to enable a realistic execution of the samples.

According to what we discussed in Section II, each sample was executed for up to 15 minutes. Then, the execution was replayed, this time by using our PANDA plugins to collect the required information. In average, the complete analysis of one sample took around one hour, resulting in over 4100 days of CPU time – which in our case were distributed over 80 parallel virtual machines.

Overall, in our experiments we performed $5.9M$ minutes of malware execution, over which we collected $205M$ system calls and $84M$ unique basic blocks.

A. Filtering

As expected, some samples implement tricks to detect the presence of the analysis environment or that they were executed inside an emulator. While currently, the actual amount of samples that adopt anti-analysis techniques is not known, one of the last experiments to measure its adoption was performed by Symantec in 2014. The authors found that anti-VM techniques were in decline, with only 18% of the samples that refused to run in a virtual environment [103]. In addition, some programs also failed to run because of lack of parameters or missing dependencies.

From one point of view, since our goal is to put ourselves in the position of a security company that needs to analyze unknown samples collected by their infrastructure, one may see these samples that failed to properly executed as "part of the game". Since these executables cannot be easily removed without first executing them, the tuning of the sandbox needs

TABLE III: Summary of the dataset

	Discard	**<2m**	**>2m & <15m**	**15m**	**Tot**
Malicious	8,402	55,669	10,658	11,290	86,019
Benign	1,456	10,385	798	1,375	14,014

to take them into account. On the other hand, we wanted to avoid polluting our findings with these cases, since their actual number may vary depending on the sophistication of the sandbox and on how realistic the analysis environment is. Clearly, to obtain a more accurate picture about the malware analysis, we need to discuss only the *running* samples, without including broken binaries, programs that exit because of wrong parameters or missing dependencies, or malware that detected the presence of our analysis environment. Therefore, we decided to conservatively filter out samples that did not show a sufficient amount of activity during their execution. This has an important implication for our results, as when we conclude that a certain percentage of malware shows a certain distribution in its runtime behavior, we always mean "*a percentage of the malware that successfully executes*". This needs to be kept in mind when interpreting the results of our analysis.

From a practical standpoint, we applied two conservative thresholds to the collected data: the first over the number of invoked syscalls, and the second over the number of executed basic blocks. More specifically, for the syscalls we defined a set of functions which capture signs of meaningful activity (e.g., disk operation or network-related activity) and we then required that at least 50 of these functions were invoked. With regards to the basic blocks, we adopted $4,000$ basic blocks as a minimum threshold for a successful execution. While these values are somehow arbitrary, our sole goal here is to be conservative enough to remove samples that did not execute correctly. In our dataset, these thresholds discarded $8,402$ (10.5%) of the malicious samples and $1,456$ (10.9%) of the benign ones. Among the remaining samples, the minimum execution time before a malicious sample terminated was 28 seconds—which is certainly sufficient for a program to show some of its runtime behavior.

B. *Execution Time*

The first interesting result we observed in our experiments is that most malware samples terminate their execution before reaching the sandbox threshold. This is very important because it means that even if a sandbox is configured to execute malware for ten minutes, 81% of the samples that successfully executed will not reach this threshold, and over half of them will terminate within the first minute. The full cumulative distribution of the execution time is reported in Figure 2 for both the benign and the malicious files. It is interesting to note that the two curves are remarkably similar, and that both show that either a program terminates in the first three minutes, or it continues to run for more than thirteen. A complete breakdown of the entire dataset, divided according to the samples termination time, is presented in Table III.

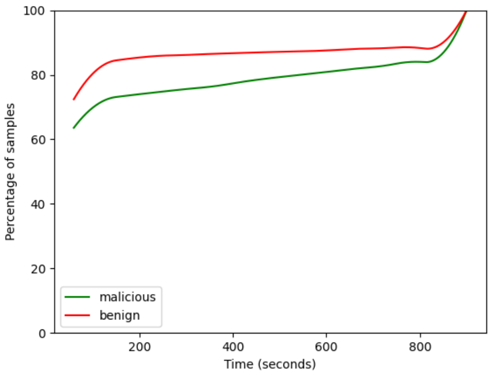

Fig. 2: Malware execution times

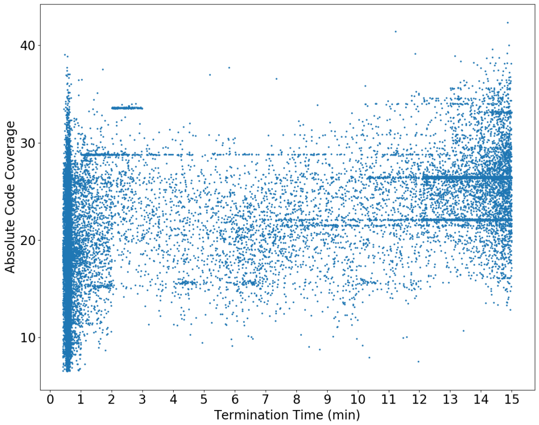

Fig. 3: Absolute Code Coverage vs Termination Time

This result could be explained by the fact that either a program performs some actions (e.g., it downloads a second-stage binary, or it manipulates some data) and then terminates, or it remains active potentially indefinitely (as in the case of a botnet or an application that requires user interaction). By looking closely at the samples that terminated in the first minute, we fetched the system calls related to network operations (e.g. `NtDeviceIOControlFile/DeviceControl`) and we analyzed their input parameters to understand if the samples were opening a new connection, downloading data, or sending data. We noted that respectively 51% of the malicious and 67% of the benign samples exhibited signs of network activity, thus suggesting that the samples did not simply detect the presence of the VM. Moreover, the top families reported in Table II were more prevalent among these short-lived samples (43% of the samples belonged to the top nine families, which covered instead only 29% of the entire dataset). Similarly, 67% of the benign samples that executed for less than one minute showed signs of network operations. This is likely due to droppers and installers.

Figure 3 shows the Absolute Code Coverage (on the Y axis) reached by each sample at the end of its execution (on the X axis). The plot confirms once more how most of the samples are either exiting in the first two minutes, or executing until the end of the 15 minutes window. It also shows that the absolute code coverage does not change much among these two

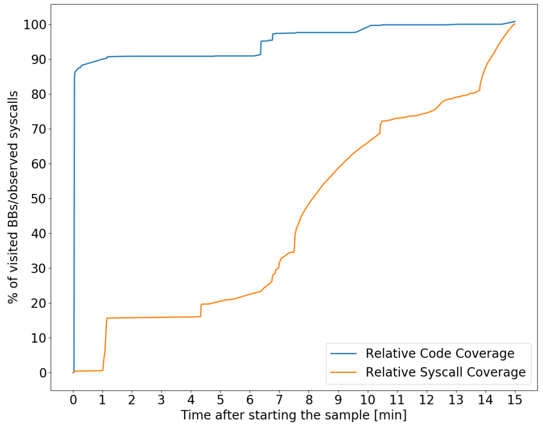

Fig. 4: Relative Syscall and Code Coverage of a Malware Sample

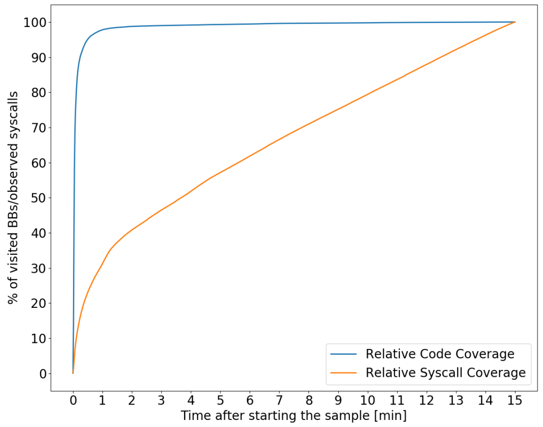

Fig. 5: Global evolution of syscalls and unique basic blocks over time

extremes—ranging from 18.8% of the samples terminating in the first minute to 26.4% for those on the right side of the graph. Similarly, benign samples reached an average Absolute Code Coverage of 24% at the end of their execution.

C. Behavior evolution

Figure 4 synthesizes the main results of our study for one random malware sample (our system generated one graph like this for each sample in our dataset). It is interesting to note the distance between the two lines, one showing the Relative Syscall Coverage (i.e., the percentage of the system calls observed at time t with respect to the total number observed over the 15 minutes of our experiment) and one the Relative Code Coverage (i.e., the percentage of unique basic blocks observed over time). It is surprising that most of the basic blocks were observed during the first minute, while the number of system calls grows more slowly for the entire duration of the experiment. For instance, after three minutes R_c is over 90% while R_s is only 16%.

To look at the samples in a cumulative way, we plotted Figure 5, over all basic blocks and syscalls that have been executed by all samples, to provide a view on the global evolution of R_c and R_s. Once more, the orange line shows the cumulative evolution of the Relative Syscall Coverage, but this time for all samples together. The curve increases rapidly for the first two minutes, even when there are still more samples running. After that, it continues with an almost perfect linear growth that seems to suggest that, as far as samples are running, the longer we execute them, the more behavioral data we can collect.

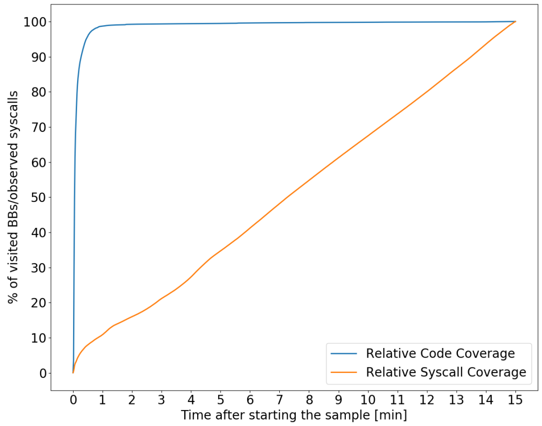

Fig. 6: Global evolution of syscalls and unique basic blocks over time, only for samples that executed until the end of the 15 minutes time window

However, the blue line shows a completely different picture. In fact, the Relative Code Coverage for all samples flattens very fast, already reaching 97% in the first minute and 99% after three minutes. While 19% of the samples run for more than ten minutes, between 10 and 15 minutes we only observe a negligible 0.2% increase in the overall code coverage – suggesting that a negligible amount of *new* behavior is observed in this time window. In fact, by combining the two lines, Figure 5 suggests that the fact that samples continue to execute new system calls does not necessarily mean that these system calls also capture new functionalities (this is also confirmed by the machine learning classifier discussed in Section VI). Instead, these calls are likely repetitions that are executed from the same code (and therefore the same behavior) that was already observed before.

One may argue that this aggregated result is skewed by the fact that half of the samples terminate during the first minute. In other words, most of the basic blocks are observed at the beginning because that is when most of the samples are running. Even though this would not explain the linear system call growth, further evidence against this hypothesis is provided by Figure 6. This is the same graph we discussed above, but this time plotted only for the samples that executed until the end of the 15 minutes period. The system call line is now perfectly diagonal, confirming that its initial increase was in fact due to more samples running during the first minute of experiment. However, the Relative Code Coverage line remains almost unchanged. Therefore, also for those samples that kept running until we stopped the sandbox, over 90% of the basic blocks were already visited during the first minute.

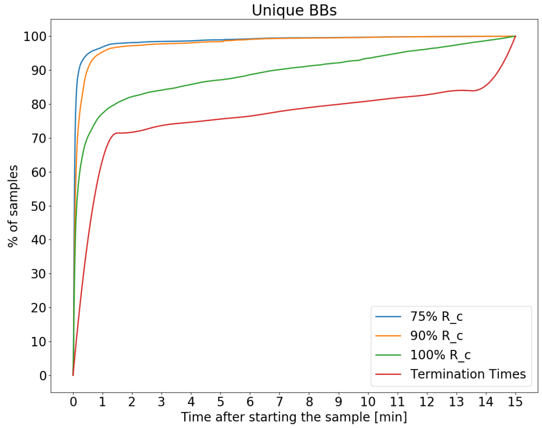

Fig. 7: Global evolution of stable behaviors over time

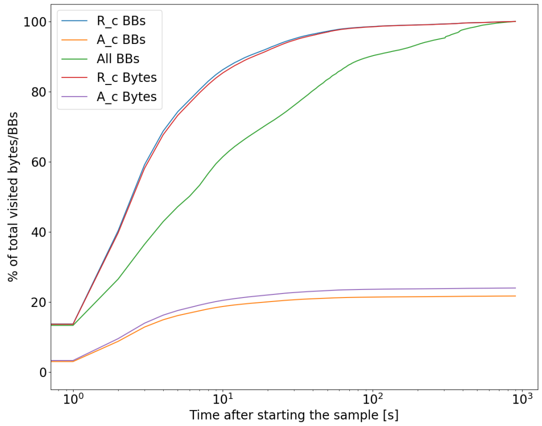

Fig. 8: Impact of counting bytes vs basic blocks, and removing or not library code

A different way to further investigate this is to look at the number of samples that reach a stable behavior over time. Figure 7 shows the percentage of samples for which we observed respectively 100%, 90%, and 75% of the basic blocks at a certain time within our 15 minutes timespan. Here, it is interesting to observe the difference between the three curves with respect to the termination line (in red). For instance, after two minutes the 71% of the samples have terminated their execution, but for more than 80% of them we have already observed 100% of their relative code coverage—and the two curves kept this 10% difference until the last minute of experiments. This indicates that R_c does not reach 100% only when the sample terminates. In some cases (that 10% in the graph) samples still run for several minutes without triggering any new line of code.

Comparison Among Different Code Coverage Metrics

As we explained in Section III-B, we can measure absolute code coverage by counting either basic blocks or by measuring the size of the executable memory—and both techniques are subject to different errors. Moreover, as we explained in Section IV, our system is able to distinguish between the code of the sample and the code belonging to shared libraries or to other processes the malware is infecting at runtime. However, since this process is also potentially subject to errors, one more conservative way would be to count all code, independently from to whom it belongs to.

So far, we have presented results based on counting only those basic blocks that are part of the malicious sample code. In our opinion, this is the most appropriate way to look at malware behavior. However, to account for the possibility of measurement errors, it is important to show that our findings would be the same also if using any of the other measurement approaches. Figure 8 compares the different techniques. Note that we use a logarithmic time scale to further emphasize the differences, that would otherwise be compressed to a small region of the graph.

The plot shows two important things. First of all, there is practically no difference in the Relative Code Coverage metric if we compute it by counting basic blocks or raw memory regions (red and blue curves). The effect on the Absolute Code Coverage (curves orange and purple) is more visible, and leads to a total code coverage of 21.6% if we count disassembled basic blocks or 23.8% if we look at memory regions.

Finally, the green line shows the Relative Code Coverage if we include **all** code, including external libraries. Furthermore, even in presence of errors in the process of dumping the memory and determining the executable basic blocks or code regions (e.g. due to packers that re-encrypt memory regions), this line still contains the executed basic blocks and thus serves as a lower bound for R_C. At first, this curve grows slower than the ones computed before. For instance, after two minutes the code coverage computed by including all code is 91%, while by looking only at the code belonging to the malware sample itself the value is 98.7%. This difference can be explained by the fact that a piece of malware code can invoke the same library function twice but with different parameters, thus resulting in different execution paths inside the library code itself. In any case, at ten minutes the difference between the two curves is already below 0.5%.

Impact of Stalling Code

As we already mentioned in Section III, when measuring time we need to decide whether we want to fast forward through `sleep` invocations or preserve the program pauses. In our experiments, 14% of the malicious and 6% of the benign samples called `Sleep` (or `SleepEx`) at least once—accounting for a cumulative sleeping time of respectively $569,310,375$ and $6,097$ minutes.

To assess the impact of stalling code on the amount of time required to execute a sample, we compare in Figure 9 the evolution of the Relative Syscall and Relative Code Coverage metrics when the sleep time is preserved. Quite surprisingly, sleep time seems to have an almost negligible effect on code coverage and a 10% reduction to the cumulative number of system calls collected over the 15-minute period.

In fact, at a closer inspection, 75% of the 11,874 malware samples that call `sleep` only stalled their execution for less than one minute. Overall, 3.1% of our malicious samples slept for more than three minutes and only 2.3% for more than ten. This result provides an important figure for companies

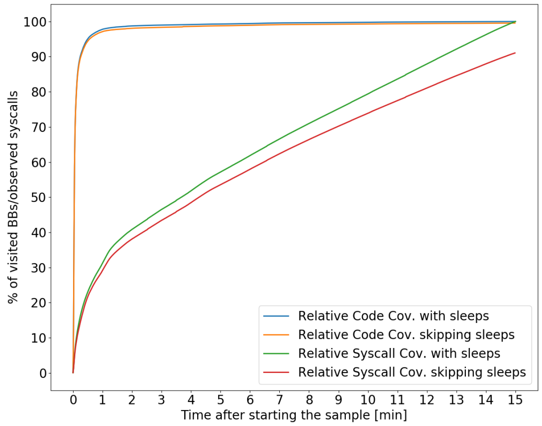

Fig. 9: Impact of sleep time over the global evolution of syscalls and code coverage.

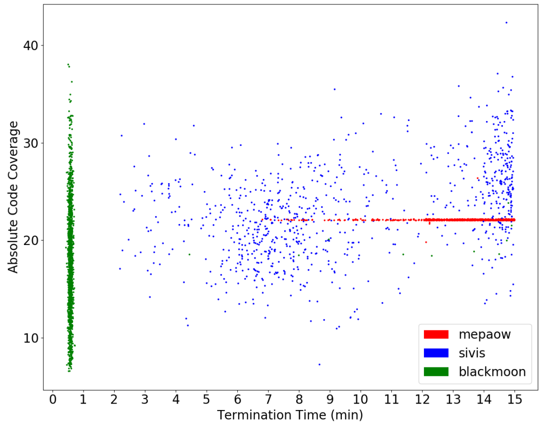

Fig. 10: Absolute Coverage comparison between 3 families

that use sandboxes that do not skip sleep time (the majority, according to the experiment we discussed in Section II). Based on our experiments, these companies can expect between two and three percent of the samples they analyze to complete evade their sandbox environment.

Impact of Code Injection

Code injection is a well-known technique applied by malware [18], [55]. Our sandbox only tracks processes that belong to the same process tree of the malware. Therefore, code injection could bypass our analysis if code is injected into a process outside the sample's process tree. To measure its impact on our dataset, we analyzed the collected behavior for signs of code injection techniques by identifying write-access to memory regions of other processes. Our experiments show that while in total 19% of all samples write to memory of other processes, only 2% of all samples inject code into processes outside their own process tree and are therefore able to partially evade our data collection mechanism.

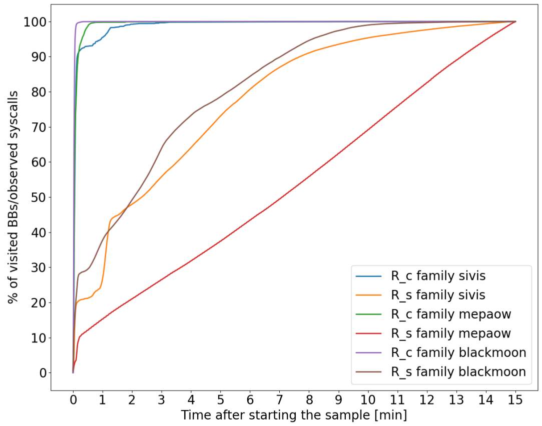

Fig. 11: Code and Syscall coverage comparison between 3 families

D. Intra-Family Variability

So far, we always looked at aggregated values computed over the entire dataset. However, it is interesting to see whether our metrics tend to remain constant within a given malware family. If so, the distribution of samples among families in the dataset could have a huge impact on our results.

To answer this question, we re-plotted the two figures that best summarized the results of our experiment (the scatterplot of the absolute code coverage and the R_c and R_s curves), but only with the samples of three families: sivis, mepaow, and blackmoon. Figure 10 shows the three families by using different colors. The distribution of the points are very different: all mepaow samples have the same code coverage, but differ in their execution time; blackmoon samples are the opposite, as they all run for a very short amount of time but achieve different absolute code coverage; finally sivis' points are scattered everywhere, showing no clear pattern.

To explain these different behaviors we had to manually look at the details of each family. *Blackmoon* is a banker that tries to inject code into other programs by calling `NtCreateSection` and `NtMapViewOfSection`. Different samples had different targets, and often more than one. If none of the targets was found running in the machine, which is our case, the malware terminates. This is a common scenario, that shows what happens when a malware family is analyzed in a sandbox that does not properly mimic the expected environment. *Mepaow* is instead a trojan keylogger. All samples in our dataset had exactly the same behavior, which explains why we observe the same coverage. But the malware calls many times `NtWaitForSingleObject`. Sometimes the wait fails, and in that case the call blocks until a timeout is reached. This explains why the execution time of most samples varies between 12 and 15 minutes. Finally, *sivis* is a file infector, and therefore the executables we analyzed in our sandbox are in fact other programs infected by the malware, thus explaining why both the execution time and the A_c metric were so different from one sample to the other.

Figure 11 shows the Relative code and syscall coverage lines. Different families have different trends but overall we still observe a very steep exploration of basic blocks (a little less pronounced for the file infector as its sample execute code from the infected applications) and a more smooth progress of the system calls.

To conclude, with these three examples, we illustrated how different families can result in different behavioral profiles, but in particular that it is often difficult to predict how long a sample would run and how much code it will trigger, even within a single known family.

E. Analysis of long-time running samples

Since a large portion of malware in our dataset executed for the whole duration of the 15 minutes time window, we decided to manually investigate the nature of these samples. In the following, all percentages are relative to the samples that ran for the entire 15 minutes period.

The first aspect we studied is the amount of network related syscalls and the contacted IP addresses. More specifically, we noticed that 22.45% of the samples executing for 15 minutes were contacting remote servers for the entire time frame. We further analyzed the invoked syscalls and the contacted IP addresses and we dissected these samples into three minor groups. The first one, accounting for 10.45% of the long-running samples, is represented by C&C samples that receive a command from the remote server and show some patterns which always repeat, such as opening and writing a file, or setting a new entry in some registry. In contrast, the second group (9.71%) are botnet samples that contact many more IP addresses (from a minimum of 10 to a maximum of 820 distinct IPs), some of them belonging to infected hosts and some others to machines under the attacker's control. The last group (2.29%) is composed of C&C samples as well, but this time receiving only a small number of commands from the remote server. Therefore, samples in this last group spent most of their time in invocation of the `NtWaitForSingleObject` / `NtWaitForMultipleObject` system calls, i.e., sending a request to the server and waiting for a response from it.

A second behavior that we could observe looking at the last minutes of execution is related to filesystem accesses. In particular, we retrieved the number of filesystem activities from the system calls log, and we considered all the samples that were still showing such behavior when they were interrupted (i.e., when we reached 15 minutes). In total, 40.17% of the samples kept repeating file operations (from a minimum of 10,000 operations) for their entire execution time. Because of the shown behavior and after a manual analysis of some of these samples, we categorized them as file infectors (which was also confirmed by the AVClass-labels of the samples).

Intuitively, we would have expected that a non-negligible part of the long-time running samples consisted of malware that spawn a GUI and spend 15 minutes waiting for a user interaction. Detecting these samples at syscall level is not trivial, because no direct mapping between the graphical API and the internally invoked syscalls exists. Moreover, most of the graphical operations are performed by means of IO control messages sent to and from the dedicated device driver (by means of `NtDeviceIOControl`) that do not allow to easily recover the real meaning of the syscall. Hence, we tried to estimate the number of malicious samples that implement a GUI at API level, i.e., by fetching when a sample imports and invokes methods such as `MessageBox`, `MessageBoxEx`, etc. This way we were able to determine that 19.75% of long-running malicious samples indeed used a graphical interface to communicate with the user and to receive an input. This percentage must be considered as a lower bound, because we use the import address table (IAT) to resolve the calls and it is well-known that the IAT can be obfuscated by packers.

Finally, we found that 11.19% of samples justify their 15 minutes duration because they continuously invoke `NtWaitForSingleObject` / `NtWaitForMultipleObject` on alertable objects (Mutexes, Events). This set contains samples that use these syscalls invocation to implement some form of stalling code and other cases where the malicious program really needs to wait for some event (e.g., wait for tasks performed by other threads).

The categories described above cover 93.57% of the samples that reached the 15 minutes threshold. For the remaining 6.43% we were unable to identify a common pattern. However, we can hypothesize that this group includes some undetected GUI samples or malware that implementing complex and long-lasting computations (for instance crypto loops or crypto mining functions).

VI. Impact on Malware Classification

In the previous section we focused on the volume of system calls and basic blocks that can be collected by running malware samples for a given amount of time. However, the volume alone tells only part of the story. For instance, it is possible that while most of the data is collected over the first minutes of execution, these early events are mostly associated to generic actions (e.g., loading shared libraries and testing internet connectivity), and that the "quality" of data improves instead over time. In this section we propose one possible way to assess the quality of data, by measuring its impact on the accuracy of a machine learning classifier with different execution time of the samples. The choice of the classifier can slightly change the contribution of individual features in the classification task. However, the overall accuracy of the applied classifier reflects the strength of the statistical association between the involved features and the malware classification output. A higher classification accuracy denotes that the input features are statistically more informative with respect to the classification use. In our study, by analyzing the classification power of the features within different time windows, we can therefore unveil the association between the length of the time window and the amount of information useful for classification in each window.

A. Classifier

The literature is full of ML-based approaches applied to dynamic analysis traces. For instance, classic machine learning models [16], [77], such as Markov chain and Support Vector Machines, were applied on sequences of system calls derived from dynamic analysis to capture sequential patterns of successively executed system calls. Pascanu et al. [73], inspired by text classification research, proposed the use of recurrent neural networks, such as Long Short-Term Memory (LSTM) [42] and Gated Recurrent Units (GRU) [27] for modeling system call sequences. Similarly Jindal et al. [47] considered the dynamic analysis reports containing system calls, network activities, changes to the registry and file actions as documents to which they applied recurrent neural networks to automatically extract language features from the reports. The success of all these approaches follows the same spirit: the sequential patterns of the behavioral features, such as the executed system calls, play an important role in characterizing malware.

Since our goal is not to design a new ML-based classification scheme, we decided to mimic the state-of-the art sequential-mining-centric solutions in our study. At each time step, we transformed all the system calls (including their parameters) available in the collected malware samples to categorical features. Then a sequential inference model was applied for malware classification. Given a sequence of n system calls $\mathbf{x} = \{x_0, x_1, x_2, ..., x_n\}$, we used the categorical index x_i of each system call (the string with both system call names and parameters) as the corresponding categorical feature. The derived categorical feature vector $\mathbf{x}$ is used as input to the classifier. Let $\mathbf{x}(i)$ denote the system call feature vector extracted from the i-th minute of the run-time of a malware sample. Each $\mathbf{x}(i)$ is composed by a sequence of system calls $\{x_i^j\}$ (j=1,2,3,...,t). The sequential inference model is organized similarly as Recurrent Neural Network [42]. It is defined with a cascaded structure as follows:

$$\begin{aligned} &h_i = \delta(w_1 * f(\mathbf{x}(i)) + w_0 * h_{i-1}) \ \ (i = 2, 3..., m) \\ &h_1 = f(\mathbf{x}(1)) \end{aligned} \tag{1}$$

where h_i denotes the latent embedding of the system call sequences from the beginning to the i-th time step. In our study, we simply set h_i as a scalar variable and δ as the sigmoid activation functions. $f(x_i)$ denotes the embedding function mapping the categorical feature vector $\mathbf{x}(i)$ to a scalar embedding variable. w_1 and w_0 are the combining coefficients concatenating the current feature embedding $f(x_i)$ and that of the previous time window h_{i-1}. f is chosen as Gradient Boosted regression trees to generate the embedding of the categorical features, as proposed by [111]. Combining the tree-based classifier and sequential inference structure, the design of the classifier is robust against the potential imbalance of malware samples of different families. In the following experiments, f is empirically fixed to have 100 trees. Empirical results verify the setting of the trees. The training process was conducted recurrently to optimize a binomial classification objective as:

$$\begin{aligned} f^*, w_0^*, w_1^* &= \min_{f,w,h} \sum_{i=1}^{s} \ell(h_m^s, y_s) \\ \ell(h_m^s, y_s) &= -y_s log(h_m^s) - (1 - y_s) log(h_m^s) \end{aligned} \tag{2}$$

where h_m^s denotes the embedding of the system call sequence sample $\mathbf{x}_s$ derived at the final time step of $\mathbf{x}_s$. It integrates the information from all m time steps and uses it as the embedding feature vector of $\mathbf{x}_s$. y_s denotes the class label of $\mathbf{x}_s$. In our study, $y_s = 1$ indicates that $\mathbf{x}_s$ is malicious and vice versa. According to Eq.1. the designed temporal model encodes the sequential pattern of the system call sequences via recurrently updating the embedding variable h. At each time step, the embedding variable h_i integrates the embedding feature of the system calls observed at both the current and the last time steps. In our study, we used Gradient Boosted trees as an economic embedding function to avoid the highly intense computation of training LSTM [42] and pre-training word2vec embeddings of the categorical features as the input to LSTM [62]. Gradient Boosted trees are designed intrinsically to handle categorical input without any pre-trained embeddings. Moreover, cascaded Gradient Boosted trees can be equally powerful in capturing non-linear relations between input features and the predicted malware classes [112]. Compared to decision trees, the gradient tree model is known for its tolerance to the class imbalance issue. This model is embedded with class re-weighting in its design of loss function, which balances the impact of misclassification penalty between positive and negative classes. Furthermore, we deploy in the experimental study a cascaded boosted tree model, in which each layer corresponds to a time window of 1 min. We extend the cascaded model to cover continuously longer run-time periods, as reported by Table IV and Table V.

Note that our goal is to understand which minute during the dynamic analysis provides the best information. Therefore, we do not claim either to outperform the existing techniques proposed to date, or that our machine learning technique is the most appropriate for this problem.

B. *Experiments*

As presented earlier, a large number of both benign and malicious samples run for less than 5 minutes. To prevent the designed classifier from overfitting due to early stop of the samples, we decided to work only with the samples that run for at least 5 minutes, and used the first 5 minutes to understand whether with longer analysis time more informative behaviours are observed.

By converting the sequence of system calls into the categorical features, we obtained a richer (i.e., a bigger number of unique system calls with parameters) system call collections. However, we would like to stress that richer collections of system calls do not necessarily indicate more useful information for the classification. Instead, encoding irrelevant system calls into the feature set can dilute the discriminating power of the features.

We organized two experimental tests. At first, we divided the whole run-time period into successive and non-overlapped chunks of one minute. In this way, we evaluated the malware classification accuracy using the system-calls observed within the increasingly larger time windows of the first 1, 2, 3, 4 and 5 minutes separately, as shown in Table IV. After that, in Table V, we further applied the classifier with 100 trees on the system call collections observed with the first, the second, the third, the forth and the fifth minute. The variation of the classification performances obtained in each of the windows is used to justify which time period provides the most useful information for classifying malware samples.

Our dataset is composed by 1500 benign samples that meet our criteria, to which we added 6K malware samples chosen randomly among those that run for more than five minutes. In both of the experiments (in Table IV and Table V), for each length setting of the time window, we conducted 10-fold cross validation and computed averaged ROC-AUC scores (Area-Under-Curve score of Receiver Operating Curve) and PR-AUC scores (Area-Under-Curve scores of Precision-Recall Curve).

TABLE IV: Classification accuracy from the first K-minutes

Time window length	ROC-AUC score	PR-AUC score
First minute	0.968	0.950
First two minutes	0.967	0.955
First three minutes	0.965	0.953
First four minutes	0.965	0.953
First five minutes	0.965	0.953
First ten minutes	0.966	0.953

TABLE V: Classification accuracy for each minute

Minute	ROC-AUC score	PR-AUC score
First minute	0.968	0.950
Second minute	0.910	0.900
Third minute	0.907	0.889
Fourth minute	0.914	0.894
Fifth minute	0.910	0.901

The ROC Curve summarises the trade-off between the true positive rate and false positive rate for a model using different thresholds over the probabilistic output of the classifier. The Precision-Recall curve is used as an alternative yet popular accuracy measurement in information retrieval [79]. The PR curve is usually used to complement the ROC, especially in the case where testing samples are imbalanced. In each round of the 10-fold cross validation, we randomly split the whole data set. 80% of the data instances are used to train the classifier. The remaining 20% of the data instances are used to evaluate the classification accuracy of the built classifier. The sampling strategy was designed to ensure that the samples selected as part of both the training and the testing set covered the same set of malware families. The purpose is to guarantee a fair evaluation of classification performances. By computing the average accuracy obtained by across the validation tests we remove the impact of potential bias / artifacts introduced by the training-testing data split.

The classification performances obtained by using different time windows are provided in Table IV. In contrast, Table V provides the classification performance measurement obtained by using only the system calls observed within the first, second, third, fourth and fifth minute separately. As shown in Table IV, the best classification accuracy is observed by using the data collected during the first one and two minutes. By further extending the length of the time window to five minutes, the classification accuracy decreases slightly and then stabilizes quickly.

In addition to these tests on the first five minutes, we also extended our experiments to compute the AUC scores of the time window covering the first *ten minutes*, by following the given cross-validation test protocols. As observed in the last row of Table IV, the ROC-AUC and PR-AUC scores of the first ten minutes are 0.966 and 0.953 respectively (thus slightly worse than by using only the first two minutes). This confirms our intuition and our previous observations: longer time periods do not result in better classification performances. In fact, a time window with longer temporal coverage enables us to collect more system call observations in the dynamic analysis report. Nevertheless, many of those system calls recorded over the later windows already appear in the first two minutes, as we unveil by Figure 5 and Figure 6 in Section V. These recurrent system calls don't bring significantly more useful information for the classification stage, and even cause a small accuracy deterioration of the ML model. In fact, it is a well-known issue of machine learning-based classifiers that irrelevant features can dilute the descriptive power of the feature representation and decrease the resulting classification accuracy [33]. The results given in Table V confirm our conclusion. The system calls collected during the first minute provide the most useful information about the behavioural profiles of the samples. In contrast, The AUC scores derived using only the system calls collected in the second, the third, the forth and the fifth minute are consistently lower than that of the first minute. This denotes that extending the execution time window does not consistently introduce as useful features as those collected during the first minute. Table IV shows that combining the actions observed in the second minute can still help to characterize the behaviours of the samples with the ML models. Longer time windows don't introduce significant improvements of the ML model's performance.

VII. Discussion

In the previous sections we showed the results we obtained from our measurement study. We now want to put these results into context and discuss how they influence previous dynamic malware analysis experiments conducted by other researchers. We also provide recommendations for future dynamic malware analysis experiments and discuss possible limitations of our study.

A. *Threat to Validity and Limitations*

Just as every other malware analysis experiment, also our work suffers from common limitations related to the nature of analyzing adversarial software. In particular, malware authors can try to evade analysis environments by using several techniques (see Section II-C). While this is true also for other sandboxes (and thus affects real-world malware analysis pipelines) in Section V-A we describe how we tried to conservatively filter out samples that did not run correctly in our environment. However, we cannot rule out the possibility that some malware detected our environment but then continued to execute its code without showing its actual malicious behavior.

As our main contribution is to measure to what extent malicious code can be analyzed within a certain execution time, the generalizability of our results to other sandboxes is very important. Therefore, both our sandbox as well as our machine learning classifier are designed to mimic the current state of research as closely as possible. Our solution based on an emulator is inherently more precise that most VM-based sandboxes, but the precise set of collected events can differ from one sandbox to another. Another thing that might differ is the distribution of malware samples under observation. The dataset we collected for our experiments is the largest ever used for such a fine-grained analysis and because of our sampling strategy we are confident that it closely resembles the one that might have been observed by other commercial sandboxes in the same period of time.

B. Impact on Prior Work

In Section II-A, we showed that the execution time of malware analysis experiments in academia greatly differs from one paper to another, ranging from less than 30 seconds to one hour. While none of these studies provided an explanation to support the chosen threshold, every research paper is motivated by different goals which can justify a longer or a shorter analysis time.

Our results suggest that in most cases it is sufficient to execute a malware sample for two minutes to observe the majority of its behavior. The data collected during this window also led to the best accuracy for our machine learning classifier. Therefore, we now want to see how these results could have impacted those studies that either executed samples for a very short time (i.e., for less than two minutes according to Table I), or for an unusually long amount of time (i.e., more than five minutes according to our survey). We chose to focus on these intervals because on the one hand execution times shorter than two minutes might have compromised the results of an experiment, while on the other hand the use of long time thresholds was probably a sign of a sub-optimal use of computational resources. This is also important because the number of analyzed samples is often determined by the amount of time at the disposal of the researchers. Thus, running samples for longer than it is needed likely results in the choice of smaller datasets, which could affect the statistical significance of the results.

However, it is very important to understand that the exact time threshold depends on the overhead introduced by the analysis system (in our case between 1.4x and 3.7x). Moreover, the fact that previous experiments were conducted often a decade ago, therefore on much slower server infrastructures, make an exact comparison very difficult.

By looking at the papers in the first category (short execution time), we can find mainly two different motivations for the experiments performed by the authors. The first is to showcase a prototype of the proposed technique. In this case, the focus of the work is on the design of the sandbox itself. Examples in this category are papers that describe new network-oriented sandboxes [44], [45], [108], [109] or those describing the use of hypervisor techniques [56]. Overall, we believe that for these papers the short execution time was not critical for the overall contribution. However, the results obtained by the experiments have to be handled with care. The second motivation for papers in this category is to identify the most suitable classifier based on either system calls or API calls (i.e., [46], [94]). In this case, the 30 seconds timeout used by the authors cast some doubts about the findings of these studies and it is unclear whether the same results would hold for a longer execution time.

Among papers that executed samples for longer than five minutes, several work aimed at classifying or detecting malware [10], [20], [25], [59], [89]. For such cases, our results suggest that the long execution times adopted by the authors were not required. The small case study by Canzanese et al. [25] confirms our results. Other papers in this category aim at classifying Android malware by extracting behavioral features [24], [36], [63]. It is particularly noteworthy that Cai et al. [24] only include such apps that explore at least 50% of their code during the experiments (an amount we rarely observed in any of the samples in our dataset). Overall, we cannot draw conclusions about their execution times as the specifics of Android and Windows malware may differ. Finally, the goal of Severi et al. [87] was to provide a collection of malware traces in PANDA to enable further research. The authors presented a use-case to estimate the global unique code that is ever executed in a sandbox and one about a classification based on features extracted from the analysis. While our results suggest that the execution time of 10 minutes is not required in these scenarios, it can still be helpful to collect longer traces for future experiments. Finally, a last aspect is the collection of network packets from running malware [41], [82], [84], typically to observe DNS and HTTP requests. Russow et al. [82] found that only 23.6% of the endpoints which can be observed within one hour have been discovered in the first five minutes. This suggests that it can be useful to execute samples for a longer time. However, such a long analysis time is out-of-scope in most scenarios.

C. Recommendations

Rather than revealing inadequate execution times in previous dynamic malware analysis experiments and sandboxes, the main goal of our work is to provide insights to guide *future* dynamic analysis experiments.

We suggest to run samples for two minutes (adjusted for the actual overhead of the system) if the goal of the dynamic analysis is to perform a classification of the samples or simply to build a behavioral report for manual inspection. This is the case for most industrial analysis environments, whose applications are mostly classification tasks of unknown programs into different malicious or benign categories. Furthermore, we suggest all industrial sandboxes to implement countermeasures for basic timing attacks, usually deployed by malware authors as a defence strategy to hide a part of the malicious behaviors. Only few commercial sandboxes already handle the `sleep` functions, but they should also be aware of the presence of malware that implements alternative techniques to delay the time (e.g., `NtWaitForSingleObject`).

Experiments conducted by researchers in academia might need to tune the execution threshold to reflect the goal of their study. However, our recommendation to the authors is to include in the papers a justification of the adopted threshold, by using our experiments and results to guide their decision.

In any case, our measurements suggest that long execution times (five minutes or more) are often unjustified and would only increase the overall analysis time–which is often a costly resource in academic studies. Therefore we believe that it is better to analyze more samples for a shorter time than to reduce the dataset size to accommodate longer executions.

VIII. Key Takeaways

In this paper we argue for a data-driven approach to malware analysis. As a first step in this direction, we conducted an extensive set of experiments to evaluate the impact of the analysis time on the results collected by a malware analysis sandbox. We can summarize our main findings along the following five points:

1) Samples, both benign and malicious, tend to run either for a short amount of time (less than two minutes in our emulator) or for a long time (over ten). This is based on samples analyzed on the first day in which they were first collected. The analysis of old samples would probably move the curve even further towards short executions, as external components and needed infrastructure may not be available anymore for the sample to continue its execution.
2) While new system calls can be collected on a regular basis for the entire duration of a program, the code coverage tends to plateau very fast, typically in the first minute or two of execution.
3) Stalling code is a very well known anti-analysis technique. However, its most common form (which relies on invoking one of the `sleep` functions) only affected 2-3% of our samples. Nevertheless, countermeasures are easy to implement (and are in fact adopted by some of the commercial sandboxes), allowing to properly analyze even those samples.
4) A single execution of a binary inside a sandbox typically exposes between 10% and 40% of its code. The actual value, and the amount of time that each sample executes for, depends on the family but it also varies greatly within the same family.
5) Our experiments with a machine learning classifier show that not just the volume, but also the "quality" of the collected data (in terms of how useful it is to classify the sample), is mostly concentrated in the first two minutes of execution.

By considering all factors mentioned above, and based on the data analyzed in our experiment, we therefore confirm the initial intuition of Willems et al. [101] that a threshold of two minutes is sufficient in the vast majority of the cases for the analysis of freshly collected malware samples.

Of course, the actual value needs to take into consideration the amount of available resources. Moreover, this value is based on new samples submitted over a period of six months, between November 2019 and May 2020. Therefore, the threshold should be regularly updated to reflect changes in the malware ecosystem or in the type of collected data. Thus, we believe security companies should adopt a self-tuning analysis infrastructure, where the parameters of the sandbox are regularly re-tuned based on the recently-collected data.

Acknowledgment

This research was supported by the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 771844 – BitCrumbs).

References

[1] Anubis sandbox. [Online]. Available: https://anubis.iseclab.org

[2] Cuckoo sandbox. [Online]. Available: https://github.com/cuckoosandbox/cuckoo

[3] Hybrid-analysis sandbox. [Online]. Available: https://www.hybrid-analysis.com

[4] Panda – syscalls2 plugin. [Online]. Available: https://github.com/panda-re/panda/tree/master/panda/plugins/syscalls2

[5] Panda plugin win7proc. [Online]. Available: https://github.com/panda-re/panda/tree/panda1/qemu/panda_plugins/win7proc

[6] Rekall forensics. [Online]. Available: http://www.rekall-forensic.com

[7] Smda. [Online]. Available: https://github.com/danielplohmann/smda

[8] Virus total. [Online]. Available: https://www.virustotal.com/

[9] Vmray sandbox. [Online]. Available: https://www.vmray.com/

[10] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 162–169.

[11] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. D. Yao, "Methodologies for quantifying (re-)randomization security and timing under JIT-ROP," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[12] B. Anderson and D. McGrew, "Machine learning for encrypted malware traffic classification: accounting for noisy labels and non-stationarity," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[13] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[14] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (Euro S&P)*, April 2017.

[15] K. Aoki, T. Yagi, M. Iwamura, and M. Itoh, "Controlling malware http communications in dynamic analysis system using search engine," in *2011 Third International Workshop on Cyberspace Safety and Security (CSS)*. IEEE, 2011, pp. 1–6.

[16] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden markov models and metamorphic virus detection," *Journal in computer virology*, vol. 5, pp. 151–169, 2009.

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

[18] T. Barabosch and E. Gerhards-Padilla, "Host-based code injection attacks: A popular technique used by malware," in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*. IEEE, 2014.

[19] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, 2009.

[20] U. Bayer, E. Kirda, and C. Kruegel, "Improving the efficiency of dynamic malware analysis," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[21] M. Brengel and C. Rossow, "Memscrimper: Time-and space-efficient storage of malware sandbox memory dumps," in *DIMVA*. Springer, 2018, pp. 24–45.

[22] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, *Automatically Identifying Trigger-based Behavior in Malware*. Boston, MA: Springer US, 2008, pp. 65–88.

[23] P. Burnap, R. French, F. Turner, and K. Jones, "Malware classification using self organising feature maps and machine activity data," *computers & security*, vol. 73, pp. 399–410, 2018.

[24] H. Cai, N. Meng, B. Ryder, and D. D. Yao, "Droidcat: Unified dynamic detection of Android malware," Department of Computer Science, Virginia Polytechnic Institute & State University, Tech. Rep., 2016.

[25] R. Canzanese, M. Kam, and S. Mancoridis, "Multi-channel change-point malware detection," in *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 2013, pp. 70–79.

[26] Y. Cao, J. Liu, Q. Miao, and W. Li, "Osiris: A malware behavior capturing system implemented at virtual machine monitor layer," in *2012 Eighth International Conference on Computational Intelligence and Security*, 2012.

[27] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[28] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *2010 IEEE Symposium on Security and Privacy*. IEEE.

[29] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *2018 IEEE S&P*. IEEE, 2018, pp. 161–175.

[30] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, "Temporal search: Detecting hidden malware timebombs with virtual machines," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. ACM, 2006, p. 2536.

[31] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.

[32] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC CCS*, ser. CCS '13. ACM, 2013.

[33] B. Efron, T. Hastie, L. Johnstone, and R. Tibshirani, "Least angle regression," *Annals of Statistics*, vol. 32, pp. 407–499, 2004.

[34] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, pp. 1–42, 2008.

[35] C.-I. Fan, H.-W. Hsiao, C.-H. Chou, and Y.-F. Tseng, "Malware detection systems based on api log data mining," in *2015 IEEE 39th annual computer software and applications conference*, vol. 3, 2015.

[36] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio, "Spotting the malicious moment: Characterizing malware behavior using dynamic features," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016.

[37] P. Ferrie, "Attacks on virtual machine emulators," Tech. Rep., 2007.

[38] D. Fleck, A. Tokhtabayev, A. Alarif, A. Stavrou, and T. Nykodym, "Pytrigger: A system to trigger & extract user-activated malware behavior," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 92–101.

[39] V. Foundation. volatility/win7_sp1_x86_vtypes.py. [Online]. Available: https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/overlays/windows/win7_sp1_x86_vtypes.py

[40] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *2010 IEEE Symposium on Security and Privacy*, 2010.

[41] M. Hatada, M. Akiyama, T. Matsuki, and T. Kasama, "Empowering anti-malware research in japan by sharing the mws datasets," *Journal of Information Processing*, vol. 23, no. 5, pp. 579–588, 2015.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] S.-W. Hsiao, Y.-N. Chen, Y. S. Sun, and M. C. Chen, "Combining dynamic passive analysis and active fingerprinting for effective bot malware detection in virtualized environments," in *International Conference on Network and System Security*. Springer, 2013.

[44] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao, "Malware behavior analysis in isolated miniature network for revealing malware's network activity," in *2008 IEEE International Conference on Communications*. IEEE, 2008, pp. 1715–1721.

[45] ——, "Automated malware analysis system and its sandbox for revealing malware's internal and external activities," *IEICE transactions on information and systems*, vol. 92, no. 5, pp. 945–954, 2009.

[46] J.-w. Jang, J. Woo, J. Yun, and H. K. Kim, "Mal-netminer: malware classification based on social network analysis of call graph," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 731–734.

[47] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: Dynamic malware analysis without feature engineering," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 444–455.

[48] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 347–362.

[49] 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*, 2009, pp. 11–22.

[50] Y. Kawakoya, E. Shioji, M. Iwamura, and J. Miyoshi, "Api chaser: Taint-assisted sandbox for evasive malware analysis," *Journal of Information Processing*, vol. 27, pp. 297–314, 2019.

[51] S. Kilgallon, L. De La Rosa, and J. Cavazos, "Improving the effectiveness and efficiency of dynamic malware analysis with machine learning," in *2017 Resilience Week (RWS)*. IEEE, 2017, pp. 30–36.

[52] D. Kim, D. Mirsky, A. Majlesi-Kupaei, and R. Barua, "A hybrid static tool to increase the usability and scalability of dynamic detection of malware," in *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2018, pp. 115–123.

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

[54] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM CCS*, 2011, pp. 285–296.

[55] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer & Communications Security*, 2017.

[56] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *ACSAC*, ser. ACSAC 14. ACM, 2014.

[57] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2011, pp. 338–357.

[58] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current Android malware behaviors," in *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE, 2014, pp. 3–17.

[59] S.-T. Liu, H.-c. Huang, and Y.-M. Chen, "A system call analysis method with mapreduce for malware detection," in *2011 IEEE 17th international conference on parallel and distributed systems*, 2011.

[60] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti, "Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem," in *NDSS Symposium*, ser. NDSS 20, February 2020.

[61] L. Martignoni, R. Paleari, and D. Bruschi, "A framework for behavior-based malware analysis in the cloud," in *International Conference on Information Systems Security*. Springer, 2009.

[62] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedings of International Conference on Learning Representations (ICLR) Workshop*, 2013.

[63] J. Milosevic, A. Ferrante, and M. Malek, "What does the memory say? towards the most indicative features for efficient malware detection," in *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2016, pp. 759–764.

[64] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1009–1024.

[65] J. Morales, S. Xu, and R. Sandhu, "Analyzing malware detection efficiency with multiple anti-malware programs," *ASE Science Journal*, vol. 1, no. 2, pp. 56–66, 2012.

[66] J. A. Morales, M. Main, W. Luo, S. Xu, and R. Sandhu, "Building malware infection trees," in *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 2011, pp. 50–57.

[67] J. A. Morales, R. Sandhu, and S. Xu, "Evaluating detection and treatment effectiveness of commercial anti-malware programs," in *2010 5th International Conference on Malicious and Unwanted Software*. IEEE, 2010, pp. 31–38.

[68] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007, pp. 231–245.

[69] V. T. Nguyen, A. S. Namin, and T. Dang, "Malviz: An interactive visualization tool for tracing malware," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 376–379.

[70] M. Nunes, P. Burnap, O. Rana, P. Reinecke, and K. Lloyd, "Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis," *Journal of Information Security and Applications*, vol. 48, pp. 102–365, 2019.

[71] P. O'Kane, S. Sezer, K. McLaughlin, and E. G. Im, "Svm training phase reduction using dataset feature filtering for malware detection," *IEEE transactions on information forensics and security*, vol. 8, no. 3, pp. 500–509, 2013.

[72] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09. USENIX Association, 2009.

[73] R. Pascanu, J. W. S. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[74] G. Pék, B. Bencsáth, and L. Buttyán, "nEther: In-guest detection of out-of-the-guest malware analyzers," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, pp. 3:1–3:6.

[75] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC '14. USENIX Association, 2014, pp. 829–844.

[76] H. Petritsch, "Understanding and replaying network traffic in Windows XP for dynamic malware analysis," Master's thesis, TU Wien, 2007.

[77] J. Pfoh, C. Schneider, and C. Eckert, "Leveraging string kernels for malware detection," in *Proceedings of International Conference on Network and System Security*, 2013, pp. 206–219.

[78] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 73–96.

[79] D. Powers, "Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation," *Journal of Mach Learn Technologies*, vol. 2, pp. 37–63, 2011.

[80] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proceedings of the 10th International Conference on Information Security*. Springer Berlin Heidelberg, 2007.

[81] C. Rathnayaka and A. Jamdagni, "An efficient approach for advanced malware analysis using memory forensic technique," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 1145–1150.

[82] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. Van Steen, F. C. Freiling, and N. Pohlmann, "Sandnet: Network traffic analysis of malicious software," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM, 2011.

[83] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 289–300.

[84] L. Rudman and B. Irwin, "Dridex: Analysis of the traffic and automatic generation of iocs," in *2016 Information Security for South Africa (ISSA)*. IEEE, 2016, pp. 77–84.

[85] Z. Salehi, A. Sami, and M. Ghiasi, "Using feature generation from api calls for malware detection," *Computer Fraud & Security*, vol. 2014, no. 9, 2014.

[86] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *RAID*. Springer, 2016.

[87] G. Severi, T. Leek, and B. Dolan-Gavitt, "Malrec: compact full-trace malware recording for retrospective deep analysis," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 3–23.

[88] T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and T. Yada, "Efficient dynamic malware analysis based on network behavior using deep learning," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–7.

[89] J. Shirley and D. Evans, "The user is not the enemy: Fighting malware by tracking user intentions," in *Proceedings of the 2008 New Security Paradigms Workshop*, 2008, pp. 33–45.

[90] M. Stamatogiannakis, P. Groth, and H. Bos, "Decoupling provenance capture and analysis from execution," in *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP'15)*, 2015.

[91] B. Sun, A. Fujino, T. Mori, T. Ban, T. Takahashi, and D. Inoue, "Automatically generating malware analysis reports using sandbox logs," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 11, pp. 2622–2632, 2018.

[92] R. Sun, X. Yuan, P. He, Q. Zhu, A. Chen, A. Gregio, D. Oliveira, and X. Li, "Learning fast and slow: Propedeutica for real-time malware detection," 2017.

[93] T. Teller and A. Hayon, "Enhancing automated malware analysis machines with memory analysis," *Black Hat USA*, 2014.

[94] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," in *2010 5th international conference on malicious and unwanted software*, 2010, pp. 23–30.

[95] P. Trinius, T. Holz, J. Göbel, and F. C. Freiling, "Visual analysis of malware behavior using treemaps and thread graphs," in *2009 6th International Workshop on Visualization for Cyber Security*, 2009.

[96] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "[SoK] Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[97] ——, "RAMBO: Run-time packer Analysis with Multiple Branch Observation," July 2016.

[98] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti, "A close look at a daily dataset of malware samples," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 1, pp. 6:1–6:30, January 2019.

[99] C. S. Veerappan, P. L. K. Keong, Z. Tang, and F. Tan, "Taxonomy on malware evasion countermeasures techniques," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018.

[100] A. Wichmann and E. Gerhards-Padilla, "Using infection markers as a vaccine against malware attacks," in *2012 IEEE International Conference on Green Computing and Communications*. IEEE, 2012.

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

[102] T. Wüchner, M. Ochoa, and A. Pretschner, "Robust and effective malware detection through quantitative data flow graph metrics," in *DIMVA*. Springer, 2015.

[103] C. Wueest. Does malware still detect virtual machines? [Online]. Available: https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=3cb6bda8-bcb6-49c6-b9e7-8247466e8441

[104] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[105] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," *SIGPLAN Not.*, vol. 47, no. 7, Mar. 2012.

[106] 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*, 2007, pp. 116–127.

[107] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes *et al.*, "Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *RAID*. Springer, 2016, pp. 165–187.

[108] K. Yoshioka, D. Inoue, M. Eto, Y. Hoshizawa, H. Nogawa, and K. Nakao, "Malware sandbox analysis for secure observation of vulnerability exploitation," *IEICE transactions on information and systems*, vol. 92, no. 5, pp. 955–966, 2009.

[109] K. Yoshioka, T. Kasama, and T. Matsumoto, "Sandbox analysis with controlled internet connection for observing temporal changes of malware behavior," in *2009 JWIS*, 2009.

[110] H. Zhang, Y. Cole, L. Ge, S. Wei, W. Yu, C. Lu, G. Chen, D. Shen, E. Blasch, and K. D. Pham, "Scanme mobile: a cloud-based Android malware analysis service," *ACM SIGAPP Applied Computing Review*, vol. 16, no. 1, pp. 36–49, 2016.

[111] Q. Zhao, Y. Shi, and L. Hong, "Gb-cent: Gradient boosted categorical embedding and numerical trees," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 1311–1319.

[112] Z.-H. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," in *IJCAI*, 2017.