Efficient Detection of Split Personalities in Malware

Davide Balzarotti[1], Marco Cova[3], Christoph Karlberger[2]
Christopher Kruegel[3], Engin Kirda[2], and Giovanni Vigna[3]

[1]Institute Eurecom, Sophia Antipolis
[2]Secure Systems Lab, Vienna University of Technology
[3]University of California, Santa Barbara

Abstract

Malware is the root cause of many security threats on the Internet. To cope with the thousands of new malware samples that are discovered every day, security companies and analysts rely on automated tools to extract the runtime behavior of malicious programs. Of course, malware authors are aware of these tools and increasingly try to thwart their analysis techniques. To this end, malware code is often equipped with checks that look for evidence of emulated or virtualized analysis environments. When such evidence is found, the malware program behaves differently or crashes, thus showing a different "personality" than on a real system.

Recent work has introduced transparent analysis platforms (such as Ether or Cobra) that make it significantly more difficult for malware programs to detect their presence. Others have proposed techniques to identify and bypass checks introduced by malware authors. Both approaches are often successful in exposing the runtime behavior of malware even when the malicious code attempts to thwart analysis efforts. However, these techniques induce significant performance overhead, especially for fine-grained analysis. Unfortunately, this makes them unsuitable for the analysis of current high-volume malware feeds.

In this paper, we present a technique that efficiently detects when a malware program behaves differently in an emulated analysis environment and on an uninstrumented reference host. The basic idea is simple: we just compare the runtime behavior of a sample in our analysis system and on a reference machine. However, obtaining a robust and efficient comparison is very difficult. In particular, our approach consists of recording the interactions of the malware with the operating system in one run and using this information to deterministically replay the program in our analysis environment. Our experiments demonstrate that, by using our approach, one can efficiently detect malware samples that use a variety of techniques to identify emulated analysis environments.

1 Introduction

The steady growth in the number of malware samples found every day has elicited an increased effort by security vendors and analysts to develop automated malware analysis tools [1, 3–6, 8]. These analysis systems typically execute an unknown program in a restricted environment (a sandbox) and monitor the program's runtime behavior. Based on the observed behavior, analysts can then assess the severity of the threat posed by the malware and develop appropriate countermeasures. Of course, malware authors have a vested interest in creating malicious code that can evade automated screening and analysis procedures. The reason is that, by remaining invisible to automated analysis systems, malware programs can operate (and generate revenue) for a longer period of time.

To thwart automated screening, malware authors have developed a number of ways to check for the presence of malware analysis tools and popular sandbox environments [19, 32]. When the malware detects indications that a malware analysis system is present, it typically suppresses the execution of malicious functionality or simply terminates. The way in which the checks are implemented depends on the type of malware analysis system that is targeted. One class of checks leverages input from the runtime environment (the operating system) to determine whether an analysis tool is present. Often, such checks look for files, registry keys, or processes that are specific to individual analysis tools. A second class of checks exploits characteristics of the execution environment that are different between a real host and an emulated or virtualized system (which is frequently used to implement the analysis sandbox). For these checks, small variations in the semantics of CPU instructions or timing

properties are leveraged to determine whether a malware process is run in an emulator or a virtual machine (VM).

To solve the problem of "analysis-aware malware" researchers have explored two kinds of approaches. One class of approaches focuses on the development of analysis platforms that are more difficult to detect by malicious code. Cobra [38] is one of the first systems that introduced the idea of stealth (or transparent) malware analysis. To this end, the system performs dynamic translation of the malicious code under examination. That is, every code block is disassembled and inspected before it is executed. During this process, each instruction that could be used to detect Cobra is replaced with a safe version, called a stealth implant [37]. Later, researchers proposed Ether [16], a system that leverages hardware virtualization to remain invisible to malware checks.

Both Cobra and Ether have been shown to be difficult to detect by current malware. However, both systems also induce a significant performance penalty, in particular when performing fine-grained analysis. Unfortunately, this level of analysis is required for comprehensive reports such as those produced by Anubis (our own malware analysis tool [1, 8]) or similar systems [3–6]. This is because these systems need, at least, to inspect the arguments of Windows API library functions in addition to system calls, and often track additional information during runtime. The main reason for the performance impact is the fact that Cobra and Ether operate on individual instructions or single-step through the process execution. Interestingly, the authors of Ether note that their fine-grained analysis "is not meant to be used for real-time analysis," while the authors of Cobra note that the performance of their tool is "within the limits of interactive analysis." Given these limitations, these systems are not suitable for automated analysis of high-volume malware feeds. For example, Anubis receives several thousand malware samples every day, and this number is likely to be significantly larger for commercial anti-malware companies.

A second class of approaches to address the problem of analysis-aware malware is to detect the fact that a malware sample behaves differently in different environments. Recently, researchers have proposed a tool in which the execution of a malware sample in an emulated (analysis) environment is compared with the execution trace of this sample on a reference system [22]. A deviation is considered to be caused by a malware check that results in the execution of a different program path. The basic idea is appealing in theory, because it promises a very general mechanism to detect malware that behaves differently in an analysis environment than it does on a reference system. However, there are a number of problems that must be solved in practice. In particular, it is important to perform the detection of deviations efficiently, and any deviation must be the result of a malware check and not due to unrelated differences between the execution traces. Unfortunately, the previously-mentioned tool [22] fails to adequately address both challenges. First, the tool uses Ether to produce the reference trace, which causes an unacceptable performance penalty. Second, malware samples are simply executed twice, once on the analysis environment and a second time on the reference system. However, as our experiments demonstrate, executing the same malware program twice can lead to different execution runs, even when no anti-analysis checks are present. Thus, a difference between two execution traces is not a reliable indicator for the presence of any anti-analysis checks in malware samples.

In this paper, we present a tool that reliably and efficiently detects malware that changes its behavior inside an (emulated) analysis environment, that is, malware with split personality. To perform the detection, we leverage the basic insight that, given the same inputs, the execution of a program should be the same in our analysis environment and on a reference system. More precisely, we first use a kernel driver on the reference host to efficiently record a trace of the system calls (and their arguments) that are executed by the malware under analysis. This system call log contains both the output arguments (the values produced by the program and consumed by the operating system) and the input arguments (the values provided by the operating system and consumed by the program). In the next step, the malware is executed in the analysis environment. Our analysis environment is a modified version of Anubis, which is an extension of Qemu, a full-system emulator. Based on the system call log, we can supply the same input arguments that the program previously received on the reference system. That is, we can perform precise replay of the malware process. This allows us to check whether the system calls (and their output arguments) that we observe in the analysis environment correspond to the ones we expect from the reference system, given the previously-recorded information. Since the inputs to the processes are the same, we expect any deviation to be the result of a check that detected our analysis system, and hence, caused the malicious code to follow a different execution path.

Our Windows process replay infrastructure, a core component of the analysis environment, is comprehensive and supports features that require special handling, such as multiple threads, memory-mapped files, and deferred system calls. This is necessary to handle the complex internals of Windows processes and make the system work on real malware programs. Our experimental results show that the proposed system can identify a wide range of different anti-analysis checks. Moreover, the system can successfully execute (replay) programs that

do not contain checks, and can detect malware in the wild that implements anti-emulator checks.

The main contributions of this paper are as follows:

- We propose a reliable and efficient approach to detect malware with split personality. Our approach works by comparing the system call trace recorded when running a malware program on a reference system with the behavior observed in the analysis environment.

- We have implemented a comprehensive replay infrastructure for Windows processes that allows us to execute the program under analysis with the same inputs on the reference system and the analysis environment.

- We demonstrate that our tool is successful in detecting a variety of checks used to identify analysis environments, including malware samples in the wild that contain checks that evade Anubis.

2 Problem Statement

The ultimate goal of an automated malware analysis tool (such as Anubis) is to obtain an understanding of the runtime behavior of malicious code that is as complete as possible. In practice, these analysis tools typically follow a dynamic approach and simply run an unknown program, monitoring its runtime behavior. The two main issues that limit the completeness of the results delivered by a dynamic analysis tool are (a) limited test coverage and (b) malware programs that detect and evade the analysis environment. To address the problem of limited test coverage, researchers have proposed extensions that explore multiple execution paths [10, 30, 39] or that scan non-executed code regions using static analysis [13].

To address the problem of malware that detects the analysis environment, researchers have proposed stealthy (transparent) analysis tools [16,38] that are more difficult to identify. As mentioned previously, these tools are effective and can gather system call traces in an efficient fashion. However, for a more fine-grained analysis that includes more than system calls, the tools have to resort to a mode in which individual instructions are inspected and logged. This is too slow for handling current malware feeds, which typically contain many thousand samples every day. Unfortunately, for a complete and comprehensive analysis, Anubis and other tools need to see more than a system call trace, and hence, require a fine-grained analysis. For example, Anubis examines Windows API library calls, and it tracks data flow dependencies.

Transparent malware analysis. One could consider directly modifying Anubis or a similar analysis tool to make it invisible to malware checks. To assess the promise of this approach, it is useful to examine the different ways in which malware can detect an analysis environment. Following the terminology introduced by the authors of Ether [16], a transparent (undetectable) malware analysis system has to fulfill five basic requirements. We will discuss these requirements in the context of Anubis, since this is the system for which we have implemented the techniques presented in this paper. However, the discussion holds in general for all approaches that execute the malware (and the operating system) in an emulator or a virtualized environment. There are alternative techniques (such as CWSandbox or Joebox) where parts of the analysis system run inside the malware process or in the same operating system. In these cases, the malware executes at a privilege level that is similar to the analyzer, and, as a result, the analysis platforms cannot be transparent [16].

The first requirement for transparency states that the analysis tool has to run in a higher privilege level than the program it analyzes. This requirement is trivially satisfied by approaches that run the target in an emulator or a virtual machine. This is also true for the second requirement, which states that all side-effects introduced by the analyzer must not be accessible through non-privileged (malware) operations.

The third requirement dictates that the semantics of the instruction set used by the analyzer has to be the same as the one used by a real host. This condition is difficult to fulfill for an emulator. An emulator basically provides a software implementation of a complex hardware device (a CPU). Hence, it is always possible that there are differences that can be detected. Indeed, previous work has pointed out *CPU semantics attacks* [16,19,22,27,32] based on differences between emulated and real CPUs, and such attacks are also used actively by malware in the wild. The checks used by these attacks exploit undocumented side-effects of x86 instructions or look for the presence of CPU-specific mode registers that are typically not present on the emulated processor.

The fourth requirement mandates that exception handling must be transparent. This also demands that system calls (including system call results) have to be the same on the analysis platform and on a real host. The authors of Ether state that this requirement can be fulfilled by Anubis and other emulator-based systems "with careful design." However, in practice, there are many ways in which the presence of an emulator can be detected by checking for artifacts in the operating system environment. This includes specific processes, drivers, or registry values in the guest operating system. For example, in a typical installation of Windows in VMware Work-

station, the registry contains over 300 occurrences of the string `VMware`. While it is in principle possible to hide these artifacts, it is difficult in practice. As a result, many *environment attacks* [11] exist that look for resources in the guest operating system to identify a malware analysis environment or the emulator/virtual machine used by the malware analysis tool.

The fifth requirement states that, for a transparent malware analysis system, there has to be an identical measurement of time. That is, instructions need to take the same time on the emulator as on the real machine. Clearly, this is not possible, since emulators and virtual machines add additional overhead that make the execution slower than on a native system. Given the large number of possible channels through which a process can time its execution, *timing attacks* [32] are very difficult to mitigate. Indeed, systems such as Ether explicitly exclude timing checks based on input from external resources (such as an NTP time server on the Internet) from their threat model.

Given the complexity of the analysis system as well as the range of CPU semantics, environment, and timing attacks that a malware author has at his/her disposal, making Anubis invisible to individual checks is clearly an uphill battle. Similarly, Garfinkel et al. [20] concluded that realizing a VMM that is indistinguishable from native hardware is difficult to the point of impracticality. Thus, we envision a more general approach to address the problem of analysis-aware malware. Our approach is a two-step process.

Detecting malware with split personalities. In the first step, we detect whether a malware program has a split personality. This is the focus of the tool presented in this paper. That is, the goal of the tool presented here is to detect malware programs whose runtime behavior in an emulator-based malware analysis system (such as Anubis) is different from the behavior that these programs exhibit on a *reference* system (a system where the malware analysis system is not present). Note that we consider an emulator (such as Qemu, which is used by Anubis) to be part of the analysis environment. Thus, our system also recognizes malware that checks for the presence of Qemu. The behavior of a program is defined as the system calls (types and arguments) that this program invokes. We believe that this is a reasonable assumption, because system calls are the means through which a program communicates with its environment and by which it can cause persistent changes to the operating system or other hosts (via the network).

It is important that a tool that identifies split-personality malware work reliably and efficiently. We strive for a system that has no false negatives or false positives. False negatives are more severe, since they imply that the analysis failed to detect a sample with a split personality, possibly missing a severe threat. False positives typically result in a performance loss, because malware samples with split personalities induce additional analysis effort. With regards to efficiency, we require that the detector do not add substantial overhead to the current Anubis analysis process. Currently, the Anubis analysis environment comprises ten Qemu workers on two physical machines. Although the machines are fully utilized, they still do not manage to process our daily sample feed. Thus, any increase in overhead immediately translates to a lower number of samples that can be analyzed.

Handling malware with split personalities. The second step is to leverage the information collected during the process of detecting the malware's split personality to improve the analysis results. One possibility is to rerun malware samples with split personalities in a transparent but costly analysis framework such as Ether. In this setup, our proposed system acts as an efficient and reliable filter to detect those malware programs that require additional analysis. Currently, this approach would be practical, as the fraction of malware programs that attempt to detect the analysis environment is quite low. For example, in a previous work [7], we found that less than one percent of the samples in the Anubis database execute known checks to detect our analyzer or are packed with executable protectors (such as Armadillo and tElock) that are known to recognize or fail on Qemu. This is consistent with the findings in a related study [11], in which the authors show that about 4% of malware samples behave differently when run inside a virtual machine. However, as the fraction of split-personality malware rises, alternative solutions are required. A promising venue is to carry out additional analysis to identify the root cause for deviating behavior. Once this root cause (likely, the malware check) is found, the analysis environment can be adapted to automatically bypass such checks. An emulation technique for "emulation-resistant" programs, which follows this general approach, was described in [22].

3 Our Approach

In this section, we discuss our approach for determining whether a given malware program has split personalities. As mentioned previously, this approach is based on the basic idea of comparing the execution of the malware on a reference system with the execution on the analysis system. We will see that for our approach to be *efficient* and *reliable*, we need to extend this basic idea with a mechanism to log inputs passed to the malware on the reference system (i.e., values read from the oper-

ating system), and to replay those inputs when running the malware on the analysis system.

3.1 Efficient Detection

Claim 1: The runtime behavior of a program can be characterized by the sequence of the system calls it executes.

We justify this claim on the basis that system calls are the mechanism through which a (user-mode) process influences its environment (the operating system) as well as external hosts (through the network). Thus, to capture the kinds of actions that a malware is performing and that it may be interested in concealing, we argue that it is sufficient to inspect the sequence of system calls that it executes (of course, considering both the types of the system calls and their parameter values). This view is consistent with a large body of prior work that uses system calls to model malware behavior or the effect of exploits on legitimate processes.

On the basis of Claim 1, comparing the behavior of a program on two different systems boils down to checking that its executions produce the same sequence of system calls (same types and parameter values). Since this comparison requires only coarse-grained analysis, instead of more precise, but expensive, fine-grained analysis of individual instructions, this analysis can be done efficiently.

3.2 Reliable Detection

To discuss reliable detection of split-personality malware, we first introduce the concept of *execution-equivalence*. We say that two systems are execution-equivalent if all programs that **(a)** start from the same initial state (i.e., memory and registers are initialized with the same values) and that **(b)** receive the same inputs on both systems exhibit the same runtime behavior.

For this definition, we also assume that a program has no race condition (that is, the results of the computation are independent of the scheduling of individual threads). We believe that this is a reasonable assumption for most programs. Also, by violating the assumption, a malware author cannot bypass our detection. Instead, programs with race conditions might be incorrectly detected as split-personality programs.

Claim 2: When the behavior of a program is different on two execution-equivalent systems, this discrepancy is the result of CPU semantics or timing attacks.

The fact that the behaviors of the same program on two execution-equivalent systems are different implies that the execution of one (or more) CPU instructions on these systems yielded observable effects for the program, and the program used these effects to follow a different execution path. The effects can be caused by CPU instructions that either have a different semantics or different timing properties on the two systems. This is precisely the characterization of CPU semantics and timing attacks.

Of course, it is not true that CPU semantics or timing attacks necessarily lead to different behaviors. That is, it is possible that a malware program contains checks to detect an analysis environment, but decides to ignore the results of these checks. However, in this case, the analysis is able to observe the entire behavior of the malware binary, and hence, we do not consider such programs as split-personality.

The astute reader might notice that Claim 2 specifically focuses on CPU semantics and timing attacks, but excludes environment attacks. The reason is that environment attacks cannot be detected as behavioral differences when executing a malware binary on two execution-equivalent systems. This is because environment attacks analyze and branch on input values that are read from the operating system. Since execution-equivalence implies that the program receives identical input values on both systems, the behavior of the program will be the same. However, as we will see, the design of our system makes the analysis environment transparent to these kinds of attacks. Hence, our detection is concerned only with detecting split-personality malware that performs CPU semantics or timing attacks.

Claim 2 motivates the design of our detection approach. That is, by running a malware program on a reference system and an analysis system, any difference in the observed behavior allows us to reliably convict this malware as split-personality. Of course, this is only true when the reference system and the analysis system are execution-equivalent.

3.3 Making Systems Execution-Equivalent

Claim 3: To be able to reliably detect split-personality malware by comparing its behavior on an analysis system with its behavior on a reference system, these two systems must be execution-equivalent.

To make two systems (such as our analysis and a reference system) execution-equivalent, we have to ensure that programs start from the same initial state, and that their inputs are identical. To provide identical starting states, we use the same operating system environment for both the analysis and the reference system. This guarantees that the operating system components such as the program loader and the runtime libraries (e.g., the Windows API functions) are the same. Furthermore, we disable any randomization mechanisms that could rearrange the address space layout of a process at load time.

Ensuring that the inputs to both programs are identical is more complicated. While identical OS installations provide identical file system objects, running the same program independently can lead to different behaviors even when the program has no split personality. As an obvious example, the program could use input from a remote host that returns different results for different client connections. More subtle examples include programs that use the current time and system information such as the current processor load. Thus, it is not possible to simply execute a malware program on the reference system and on the analysis system and expect that different behaviors are reliable indicators for different personalities.

Our solution to provide identical inputs to a program consists of recording the system calls of the program that executes on the reference system, and later replaying these system calls on the analysis system. That is, we run the malware on the reference system in *log* mode. In this mode, the sequence of system calls and all of its *in* and *out* parameters are logged. Then, on the analysis system, we run the malware in *replay* mode. In this mode, whenever the program invokes a system call, we intercept the call and retrieve the corresponding call from the log trace. Then, instead of letting the OS handle the call, we simply replay the logged system call, that is, we return control to the malware, after setting the return code and all the *out* parameters to the corresponding values observed during the log phase on the reference system. For example, in our approach, when a program attempts to read a file in replay mode, it is not given access to the actual file stored on the disk of the analysis system. Instead, the program is provided with the content of the file as it was read in log mode on the reference system.

An important advantage when replaying inputs recorded on a reference system is that *environment attacks* are not effective. That is, when the malware attempts to access resources on the analysis system that could reveal its presence, the system replays the resources that were present on the reference system. Thus, during analysis, environments attacks will result in the same behavior as on the reference system, effectively making our analysis transparent to this attack vector.

Unfortunately it is not possible to simply replay all system calls that the malware program invokes on the analysis system. In fact, there are a number of system calls that are not safe to replay. For example, if we replayed the return value of a system call that allocates new memory by simply returning the address of the memory buffer that was allocated on the references system, the program would likely crash when accessing this memory. The reason is that the operating system has not created the necessary virtual memory mappings internally, while the program assumes that memory was correctly reserved. As a result, the access results in a page fault. Thus, the replay component replays values only for those system calls that read data from the environment. The input channels we consider are the file system, the registry, the network, and time computations. This ensures that system calls that obtain input values from the environment receive the proper data recorded on the reference system. Other system calls that are used for management purposes (such as system calls for allocating memory, spawning threads, etc.) are monitored but passed directly to the underlying OS.

3.4 System Call Matching

Assume that we have a system call trace that captures the behavior of a malware program on the reference system. We then execute the malware on the analysis system. For each system call that is observed, we can check whether the type and the *in* arguments of this call match the one in the log. If this is the case, we replay the return value and the *out* arguments. When the type of the observed system call or its arguments are different, we have identified a deviation in the expected behavior and, thus, can mark the malware as having a split personality.

Unfortunately, the situation is not that easy in practice. The reason is that small timing differences can cause small, temporary deviations in the behavior of a process that is replayed. For example, a little delay in the delivery of an interrupt can cause a process to issue additional system calls. More concretely, consider the `WaitForSingleObject` function, which waits until the specified object is ready to be accessed or until a timeout expires. This function is often called in a loop that spins until an operating system object is ready. Depending on the time it takes until the object is ready, it is possible that the program has executed more (or less) invocations on the reference system than in the analysis environment. Another example would be the delivery of a signal that could lead to the execution of a system call at slightly different points along the execution trace of a process.

The previously-outlined deviations result in slightly different system call traces. However, these changes do not result in any differences with regards to the actual malware behavior. That is, the persistent changes (outputs) that the program produces are still identical. To handle these cases, we have to slightly relax our definition of equal behavior. More precisely, we do not require that the sequences of system calls produced in log and replay mode are exactly the same, but we allow some flexibility to account for small differences. This *flexible matching* approach is based on the observation that small differences are usually localized in time and tend to quickly disappear as the program continues execution.

```
1  buf_skipped = []
2  buf_extra = []
3  def flexible_syscall_match(log, curr_syscall):
4    # check for deviation
5    if len(buf_skipped) == L or len(buf_extra) == L:
6      deviation_detected()
7
8    # expire old entries in the buffers
9    # (if they aren't "write" operations)
10   expire(buf_skipped)
11   expire(buf_extra)
12
13   # get the next matching syscall from the log,
14   # -1 if none
15   pos, candidate_syscall =
16     get_next_matching_syscall(log, cur_syscall)
17
18   # the next syscall in the log matches
19   if pos == 0:
20     return candidate_syscall
21
22   # if no match or extra syscalls,
23   # search a match in the skip buffer
24   if pos == -1 or pos > 0:
25     s_pos, s_candidate_syscall =
26       get_next_matching_syscall(buf_skipped,
27                                 curr_syscall)
28     if s_pos >= 0:
29       buf_skipped.remove(s_candidate_syscall)
30       return s_candidate_syscall
31
32   # if found a match in the log
33   # but not in buf_skipped, add to skip buffer
34   if pos > 0:
35     for (i = 0; i < pos; i++):
36       buf_skipped.append(log[i])
37     return candidate_syscall
38
39   # no match: add to extra bucket
40   buf_extra.append(candidate_syscall)
```

Listing 1. Flexible matching algorithm (pseudo code).

The algorithm for performing flexible system call matching in replay mode is simple (refer to Listing 1). We use two queues to keep track of short-lived differences in the executions. One queue, called `buf_extra`, records the system calls that have been invoked by the program but that were not found in the log. The other queue, called `buf_skipped`, holds those system calls that were in the log trace but were not invoked in the current execution.

At each system call invocation, our algorithm compares this system call (`curr_syscall`) with the one at the current head of the log. If the algorithm finds a match, the algorithm returns this system call and advances one step in the log (line 18).

If the first entry in the log does not match the current system call, the algorithm first searches the queue of system calls that were previously skipped. If a match is found, it is removed from the queue and is returned to the corresponding handler (lines 22–30).

If no match is found at the head of the log or in the skip buffer, it may be the case that additional system calls were executed in log mode that were not invoked in this execution. Then, we try to find a match in the log by skipping over one or more of the calls at the head of log (a sort of look-ahead operation). If this procedure yields a match, then those system calls in the log that needed to be skipped are added to `buf_skipped`, and the match is returned to our corresponding handler (lines 32–37).

If no match is found, i.e., the current system call was not invoked in log mode, it is added to `buf_extra` and passed to the operating system.

Our algorithm uses two configurable parameters, L and M. L is used to detect deviations in the behavior of two executions. More precisely, if at any point during the replay, the number of system calls that have been skipped or added with respect to the log trace reaches L, then the algorithm concludes that the current execution of the application is different than in log mode (lines 4–6). To avoid the accumulation of short-lived, localized differences, we remove system calls inserted in the `buf_skipped` and `buf_extra` queues after M executions of the matching algorithm (lines 8–11, we do not show the implementation of the `expire` function for sake of space).

4 Implementation

Our approach requires a logging infrastructure that is capable of recording and replaying Windows system calls inside and outside of virtual machines. Here, we will describe our implementation of this infrastructure and a number of interesting, technical challenges that it required to tackle.

4.1 Log and Replay Infrastructure

Our log and replay infrastructure consists of two parts: a user-space application and a kernel driver. The user-space application is responsible to load and start the driver and to control its operations by sending specific I/O control codes. It is also responsible to start the process that has to be analyzed, to communicate the process ID of the sample to the driver, and to receive and store the data generated during the logging phase.

The driver is the core part of our system. It is responsible to trap all the system calls and either log or replay all the information exchanged between the Windows kernel and the monitored application. This is achieved by hooking the *System Service Descriptor Table* (SSDT). Each entry in the SSDT contains the entry point of a Windows system call, so when an application invokes such a call, the SSDT is queried to find the address of the function that is responsible to serve it.

When the driver is loaded, every entry in the SSDT that contains a system call address is overwritten with an address that points to one of our handler functions.

However, since it is still necessary to be able to invoke the original functionalities, the addresses of the original system calls are stored in a backup table (and restored when the kernel module is removed).

It would have been extremely impractical to manually write each of the 283 function handlers that wrap all the Windows XP system calls. In addition, the function parameters often contain complex data types such as pointers to structures (that sometimes recursively contain references to other structures), making a manual approach even more complicated.

To address these problems, we implemented a tool that automatically generates the source code for the handler functions. This generator tool receives as input the system call declarations and the definition of all the system call arguments (especially, data structures). We extracted this information from the Windows Research Kernel. Then, for each system call, our tool creates the source code of two handler functions, one to be used in log mode and the other in replay mode.

In log mode, all our system call handlers have the same purpose: To dump all the data that is exchanged between the application and the kernel, working like a proxy between the user-space program and the original system call handlers. This is done by recursively logging the content of all the parameters before and after the original system call is invoked. Additionally, the return value of the system call is also logged. The kernel driver contains a buffer that is used by the handler functions to temporarily store all the parameter values of the system calls. For performance reasons, the driver does not write the contents of the buffer into a file. Instead, the user-mode program contacts the driver at given intervals, copies the content of the buffer into user space, and finally stores it in a binary log file.

In replay mode, the driver has two main tasks: To provide the application with the same input data that was stored during the execution in the reference environment, and to analyze the application behavior looking for deviations from the expected one.

As explained in Section 3, it is not possible to blindly intercept all the system calls and replay their parameters. Some functions have important side effects in the kernel (e.g., when the program requires to allocate new memory) and, therefore, they must be forwarded to the original handler to be processed. Other system calls (e.g., when the program reads the value of a key from the registry) can instead be safely replayed by substituting all the *out* parameters and the return value with the corresponding values extracted from the log file. We say that the handler *forwards* a system call in the former case, and *replays* it in the latter.

Finally, the last task of the driver during the replay phase consists of monitoring the application behavior for possible deviations, which is done by using the flexible matching algorithm described in Section 3.

In the rest of this section, we will describe a number of practical aspects of our log and replay infrastructure.

4.2 Handle Consistency

In Windows, handles are opaque integers that are used as an abstraction to provide a uniform interface to kernel objects. Depending on the context in which they are used, handles may refer to files, registry keys, processes, timers, events, communication ports, etc. Since we replay some of the system calls and we let the operating system execute others, in replaying mode, an application will have two kind of handles: *live handles* that refer to existing objects in the kernel, and *replayed handles* that were retrieved from the log file and passed to the application by one of our replaying function handler. For example, if the application tries to open a file, we intercept the `OpenFile` system call and we replay all the outgoing parameters, including the `FileHandle`, i.e., the reference to be used by the program for any further operations on the file. However, since the `OpenFile` system call is intercepted by our driver and it is never received by the kernel, the handle we return does not reflect any actual object in kernel memory.

The problem arises because certain system call wrappers (e.g., the wrapper for `Close`) can operate on both kinds of handles. To operate correctly, these system calls need to distinguish if they are passed a live handle (which cannot be replayed) or a replayed handle (which cannot be forwarded to the OS). Therefore, our system maintains a list of all replayed handles. Then, when a system call wrapper receives a handle that is not in the list of replayed handles, it simply forwards the call to the operating system.

4.3 Networking

Windows does not have special system calls dedicated to networking operations. Instead, most of functionalities are exported using an undocumented interface through `NtDeviceIOControlFile`, a generic system call that is used by user-space applications to communicate with device drivers.

`NtDeviceIOControlFile` has two general-purpose, opaque parameters, called `InputBuffer` and `OutputBuffer`, which are used to exchange information between the application and the driver. The specific data and the format of these parameters depend on the functionality required by the application. To correctly implement our network handlers, we had to reverse engineer the parameters used by the most common network operations and understand their formats

and semantics. For instance, when the RECV function is invoked, it requires the first word of the input buffer to be a pointer to a data structure in the process memory that contains (among other data) a pointer to a byte array that will store the data received from the network socket.

In replay mode, the driver analyzes the parameters of the NtDeviceIOControlFile. If the function specified by NtDeviceIOControlFile is one of the functions that our tool supports, then the corresponding handler is invoked with the values from the log file (thus, replaying the network operation that was requested). Otherwise, the system call is simply forwarded to the operating system. This avoids disrupting services using the NtDeviceIOControlFile interface that we have not reverse engineered. We have currently implemented support for TCP sockets. Support for UDP system calls could be added in a similar way.

4.4 Deferred Results

An additional issue that arises with logging and replaying network traffic is that networking system calls often return before results are available (e.g., before the data requested from the network is ready to be sent to the user-space program). This phenomenon is referred to as *deferred results* and occurs commonly also with file system-related system calls.

More precisely, whenever a system call returns a STATUS_PENDING result, it means that the required action was successfully initiated but the results are not yet available to the application. Then, the program thread has to wait until the operation is completed by invoking NtWaitForSingleObject (or the similar NtWaitForMultipleObjects) on the event handler that was specified when the operation was initially requested.

To log deferred results, we follow a two-step strategy. First, when a system call handler has some of its output parameters deferred, it simply stores the memory location of each deferred parameter to an internal Deferred Parameters List (DPL). The second step occurs in the handler of the NtWaitForSingleObject system call. The handler forwards the call to the OS. If the result is STATUS_SUCCESS, it means that the deferred data is now available. Then, the values for the parameters that were saved in the DPL can be retrieved, and they are inserted into the correct position in the log.

Replaying deferred results is simple. In fact, when a handler needs to replay a system call with deferred results, it immediately copies all the data (including the values of deferred parameters) into its output parameters and returns it to the application. However, to prevent changing the application behavior, it also replays the STATUS_PENDING return code. This creates the impression for the application that certain parameters are indeed deferred (even though they have already been copied to the application). Therefore, the application synchronizes its execution using the NtWaitSingleObjection system call, following the same execution path that was followed in the reference environment.

4.5 Thread Management

Managing multi-threaded applications poses additional challenges. In particular, in log mode, we need to ensure that all threads of the process under test are properly identified so that we can distinguish system calls made by different threads. To do this, our handler for the NtCreateThread system call initializes a new log for every new thread. It then simply forwards the system call to the OS, and records the thread ID assigned to this new thread. We use the thread ID to uniquely associate a thread to its log. Other call handlers then get the thread ID of the currently executing thread and use it to store the execution information in the correct log. Logs also store the relative order in which threads invoked system calls.

In replay mode, the handler for the NtCreateThread system call lets the OS handle the call and associates the newly created threads (their thread IDs) to the corresponding logs according to the order of the NtCreateThread invocations. Other system calls, similarly to what done in log mode, use the thread ID to retrieve the correct log and replay inputs. Absent race conditions between the threads, this is sufficient to correctly replay multiple threads. While we did not find malware samples that relied on race conditions, it would be possible to handle these cases, for example, by forcing that multiple threads are scheduled in the same relative order, as outlined by the authors of [35].

4.6 Memory Mapped Files

Memory mapping is a technique used in most modern operating system to map all or part of the content of a file to a memory area in the process address space. When a file has been mapped into memory, the program can freely read and modify its content without invoking any additional system call. This would prevent our system to load and replay those operations. [29] proposes an interrupt-based technique to intercept the access to a memory-mapped area and dump its content whenever the process modifies this area. However, we found that in our case, the most common uses of memory mapping can be safely handled by simply forwarding the memory map to the OS.

In Windows, memory mapping is commonly used by the process loader to load dynamic libraries (DLLs) into

memory. Dynamic libraries should be considered an input of the application and therefore should be, at least in principle, logged and replayed by our system. However, it is much more efficient to have the same copies of the libraries in the reference and analysis environments and let the application free to load them from disk. For this reason, when our kernel module receives a request to open or memory map a system library, it deactivates the replay for the system call and it forwards the request to the operating system.

A second very common use of memory mapping that we observed in malicious samples is to create and later execute an executable file. To address this problem, we parse the log file that has been generated in the reference environment. Every time we find a file that is created by the process and then memory-mapped, we remove from the log the system calls responsible to open and create the file. Then, during the replay phase, the matching algorithm will be unable to find a match for the system call that was removed from the log file. Therefore, the system call is added to queue of extra system calls, and it is forwarded to the operating system. As a consequence, the handle for the memory-mapped file is generated by the operating system (i.e., it is a live handle), and other system calls that operate on this handle will not be replayed by our system. The practical effect is that the application is going to create the file and operate on it also during the replay mode.

4.7 Current Limitations

Implementing a system to correctly log and replay system calls under Microsoft Windows is a very complex task. In many cases, we had to rely on reverse engineering internal, undocumented Windows data structures and behaviors.

The current prototype supports a large set of functionalities and can be used to analyze real programs and malware samples. However, as any prototype, it still has some limitations that we can summarize in the following areas:

- *Multiple processes*: The current prototype is not able to log and replay the input if the program is composed of multiple processes. In practice, if the different processes do not communicate with each other, we can still instruct our system to analyze one process at a time.

- *Random numbers*: Correctly replaying applications that rely on random numbers to take non-deterministic decisions would require to replay the random number generator of the reference system on the analysis system. While the random number generator of Windows is implemented in user mode (and, thus, is not readily accessible from our kernel driver), its entropy sources are located in the kernel [17]. In the current implementation, we already log and replay most of the sources, with the exception of the KSecDD device driver. By also logging and replaying the data generated by this driver, we would be able to replay random numbers.

- *Inter-process communication and asynchronous calls*: Our prototype does not replay any local procedure call (LPC) communication. All the related system calls are forwarded to the operating system, with the risk of causing a deviation in the application execution.

- *Complex memory map scenario*: As we explained in the previous section, our support for memory mapping is not complete and does not support, for example, the use of mapped area as a shared space between different processes.

Some of the previous limitations may seem quite severe. In fact, it is possible for an attacker to exploit our limitations to prevent our log/replay infrastructure to work properly. However, these attacks would only cause our system to detect a deviation and raise an alert. As a consequence, we would simply incur the performance penalty of re-running the sample with a more fine-grained and expensive analyzer, such as Ether.

It is also possible to evade our system when a malware program delays the checks that aim to detect the presence of the analysis environment. This is similar to postponing malicious behavior for some time (e.g., some minutes) so that the analysis system will stop monitoring the process before the sample starts any malicious activity. Unfortunately, there is not a simple solution for this problem, with the exception of running the analysis for a longer time.

In addition, it might be possible to evade our flexible matching algorithm by dividing the malicious activities into a number of very short (and far apart) sequences of system calls, so that the localized differences would fly under the radar of our detection mechanism. Even though this is possible in theory, we did not investigate how difficult it would be to implement in practice a (malware) program in this way.

Finally, another limitation of the current implementation is that malware that gains access to kernel structures (i.e., rootkits) could detect our driver component and take some countermeasures to avoid detection. However, we do not believe that this is a significant problem. First, a previous study of current malware trends showed that only 3.34% of samples install kernel drivers [7]. Second, it is likely that such malware would perform environment checks (and, thus, would be detected by our tool) before attempting to gain control of the kernel, otherwise, the

Sample	Syscall Log Size	Replayed (VMware & Qemu)
SystemTime	123	100%
Registry	195	100%
Network	512	100%
FileSize	128	100%
FileRead	114	100%
Four Threads	252	100%

Table 1. Simple Log and Replay test.

malware author would expose some of the program's malicious functionality to the analysis system. Finally, we could resort to transparent solutions to implement our log and replay system. In fact, we believe we could have used Ether, running in the fast *coarse-grained* mode, for this. However, we have not explored this possibility because Ether was not available when we started our project.

5 Evaluation

We evaluated our system by conducting four different experiments involving a number of real-world programs. All tests have been run on Microsoft Windows XP Service Pack 3, installed in a VMware virtual machine and on the Anubis system (i.e., on a Qemu image). Both systems were installed from the same CD-ROM and were updated to the same patch level.

In our experiments, we use interchangeably Anubis or VMware as the reference system (and the other, as the analysis system). Since our tool is designed to detect differences in the behavior of an application when it is run in two different environments (any two environments), it is not necessary that one is always selected to be the reference system, nor that the reference system be a physical machine. Of course, if a malware attacks both systems (i.e., it is capable of detecting both Qemu and VMware), we may fail to detect it. To avoid this problem, it is possible to use a real machine as the reference system, and reset the computer state using hardware devices, e.g., a hard disk write-cache card such as coreRestore [2]. However, for our experiments, using a virtual machine or emulator as a reference system made it easier to rollback to a clean state after each test. Finally, both Anubis (Qemu) and VMware are actively targeted by real-world malware with split personality.

Experiment I. For the first experiment, we evaluated the ability of our system to correctly log and replay all the input channels we identified in Section 3.

For these experiments, we use VMware as the reference system and Anubis as the analysis system. The test suite included six simple programs we developed to read and write data from different channels, i.e., the file system, the registry, the system time, and the network. We also combined different input/output operations together in a multi-thread application. Table 1 shows the number of system calls that were logged and the fraction of those that were correctly replayed (or forwarded to the operating system if no replay was required for that particular call) during the replay phase. A value of 100% means that the system was able to reach the end of the log without detecting any deviation from the logged behavior, thus demonstrating the ability of our system to properly log and replay that input channel.

Experiment II. In our second experiment, we want to show that our tool can detect a variety of different, real-world detection attacks (which we manually inserted into a real-world malware).

We obtained the source code of SDBot, a well-known botnet application. We first compiled and tested in our system a vanilla version of this bot, which did not include any check to detect if it is running in a virtual machine. In Anubis (which is used as the reference system in this experiment), the application generated a log file containing 1,905 system calls. Upon inspecting this trace, we found that the 1,524th system call is responsible for spawning a new process. Since the creation of a new process is not currently supported by our tool, we consider two executions of SDBot equivalent if their behavior is the same up to the `NtCreateProcess` call. For this experiment, we designated VMware as the analysis system.

We then created 6 different versions of the SDbot program, each time adding to the code a different test to detect VMware. We extracted these techniques from several publicly available tools such as Red Pill, Scoopy, VMDetect, and SourPill. Whenever one of these tools combined together different approaches, we separated them in different programs to test them separately.

We ran each sample in log mode under Anubis and replayed it under VMware. As shown in Table 2, all the samples containing a VMware detection attack were correctly detected by our tool. It can be seen that under VMware, each sample executed only 460 system calls. This is because the application terminated immediately after detecting the presence of the virtual machine. The behavior under Anubis and VMware is clearly different, and, therefore, it was flagged by our tool.

Experiment III. In this experiment, we show that our tool can correctly log and replay real-world malware that does not try to detect if it is running in an analysis system.

The test suite used in the third experiment consisted of 10 malicious samples randomly chosen from the Anubis database among the malware known not to contain any virtual machine detection techniques. We first ran

Sample	Syscall Log Size	Result
SDBot + CPUID Time Check	460	Detection (deviation)
SDBot + GDT Check	460	Detection (deviation)
SDBot + LDTR Check	460	Detection (deviation)
SDBot + RedPill Check	460	Detection (deviation)
SDBot + VmWare Port Check	460	Detection (deviation)
SDBot + STR Check	460	Detection (deviation)

Table 2. Experimenting with SDBot and different VM detection techniques.

the samples in VMware (reference environment), logging their system calls. Then, we executed the samples in Anubis, once without replaying any inputs, and once in replay mode. The log collected during the execution of the samples contained between 1,136 and 30,066 system calls.

Table 3 reports the results of the experiments. The table contains three columns. The first column reports the name that the Kaspersky anti-virus scanner associates with the malicious sample. The second column shows the result of the experiments with the system call replay disabled, i.e., with the system configured to pass all the system calls to the operating system. In this case, the kernel module was not replaying any input data to the application. The third column shows the results when the system call replay was enabled. In the table, a result of OK indicates that the flexible matching algorithm found the behaviors to be identical. This means that the behavior on the analysis system matched the behavior on the reference system until the system call log was exhausted (or an unsupported functionality was encountered).

It can be seen that, even with system call replay disabled, the behaviors were the same in five cases. The main reason for this is that the two Windows installations are identical. However, in the remaining five cases, with system call replay disabled, our tool detected a deviation in the program behavior. When the replay was enabled, as shown in the last column of the table, the matching succeeded for all ten samples. This result demonstrates the importance of replaying the input in our approach: Without the replay, even on two identical systems, the behavior of an application can in fact be different between the two executions.

Experiment IV. Finally, in this experiment, we show that our tool is capable of detecting real-world malware that behaves differently when running inside an analysis system.

To populate the test suite, we extracted random samples from the Anubis database by querying for executables packed with one of the packers that are known to either detect or not to work properly under Qemu (e.g., Armadillo and tElock). Notice that these are the packers that the authors of Ether found not to be analyzable under Anubis [16]. In all cases, our technique was able to report the attempts of the samples to detect the presence of the analysis system.

During our test, we also found a sample (Heur.Trojan.Generic in Table 4) that worked properly under Anubis, but immediately terminated in VMware, and a sample (Backdoor.Win32.SdBot) that raised an Internal Error in VMware, but had a normal execution in Anubis.

Performance. The overhead introduced by our system largely depends on the type of operation performed by the program under analysis. For instance, writing a large amount of information to a file requires our logging module to dump and store the full data in the log. However, in our experiments, we noticed that our tool introduced on average 1% overhead when logging and replaying malware samples. When used in combination with an emulator-based malware analysis system such as Anubis, the delay this adds to the overall analysis time is negligible.

We also compared the time required to analyze a malware with Anubis and Ether. To perform a fair comparison, we run Anubis with all its analysis enabled, and Ether in one of its fine-grained analysis modes. We report here the result of running one test case that is representative of the different performance of these two systems. In particular, we executed the command `7za.exe a test.zip 1KB_rand_file`, which compresses a 1KB-long random file. Its execution under Anubis took 4.267 seconds. In our VMWare reference machine with our kernel module it took 1.640 seconds. The same execution in Ether running in memwrite mode took 77.325 seconds (approximately 20x slower). Therefore, even considering that in our experiment we have to run each sample twice, we can still analyze split-personality malware one order of magnitude faster than under Ether.

6 Related Work

Since malware is a significant security threat, a large body of work exists that presents techniques to analyze and detect malicious code. In this section, we first high-

Sample	Syscall Replay Disabled	Syscall Replay Enabled
Email-Worm.Win32.Bagle.fk	OK	OK
Backdoor.Win32.Rbot.bng	FAIL	OK
Backdoor.Win32.Agent.eny	OK	OK
Email-Worm.Win32.Zhelatin.cl	FAIL	OK
Trojan-Downloader.Win32.Agent.alnx	OK	OK
Backdoor.Win32.Rbot.ccb	FAIL	OK
Backdoor.Win32.SdBot.gen	FAIL	OK
Virus.Win32.Parite.a	OK	OK
Trojan-Downloader.Win32.Dluca.gen	OK	OK
Hoax.Win32.Renos.wu	FAIL	OK

Table 3. Real Malware with no VM-checks.

Sample	Packer	Deviation Detected?
Trojan-Proxy.Win32.Bypass.a	tElock	YES
Heur.Trojan.Generic	PE_Patch.UPX	YES
Backdoor.Win32.Agobot.aow	Armadillo	YES
Trojan-Spy.Win32.Banker.pcu	tElock	YES
Worm.Win32.AutoRun.pga	Armadillo	YES
Trojan-Spy.Win32.Bancos.zm	tElock	YES
Trojan-Downloader.Win32.Agent.acrm	tElock	YES
Backdoor.Win32.SdBot.fme	Armadillo	YES
Trojan.Win32.KillAV.or	Armadillo	YES
Net-Worm.Win32.Kolab.ckp	Armadillo	YES

Table 4. Real Malware with VM checks.

light a number of systems that use dynamic and static approaches for combating malware. Then, we focus on the different ways to detect malware analysis environments, emulators, and virtual machines. Finally, since we propose a component to replay malware, we survey related work in the area of process and system replay.

Malware analysis and detection. The traditional approach to detect malware, as implemented in anti-virus scanners, is based on (string) signatures that match specific malware binaries [36]. Because code obfuscation and runtime packing can be used to easily evade this type of detection, researchers have proposed more sophisticated techniques, for example, detection based on model checking [12], recognition of structural similarities between malware samples [24], and semantics-aware analysis of code templates that implement specific functionality [13].

Dynamic detection techniques are complementary to static analysis and typically aim to detect the execution of malicious code based on system call patterns [28]. Tools exists to intercept Win32 function calls [21], or to perform taint analysis and track data dependencies between system calls and library functions [9, 40]. This allows one to capture the behavior of malware in a more precise fashion and identify operations that are related.

Currently, the most popular approach for malware analysis relies on sandboxes [1, 3–6, 8]. A sandbox is an instrumented execution environment that runs an unknown program, recording its interactions with the operating system (via system calls) or other hosts (via the network). Often, this execution environment is realized as a system emulator or a virtual machine.

Stealth and transparent analysis. Since emulators and virtual machines are popular choices for implementing dynamic analysis systems, there have been a number of attempts to develop checks (pieces of code) to detect them.

Red pill [33] is arguably the most well-known check to determine whether code is executed under VMware. More checks have later been developed for VMware [19, 23, 32], but also for system emulators such as Bochs and Qemu [19, 31, 32]. To this end, researchers have looked for instructions that behave differently on an emulator than on a real host, using both manual [19, 32] and automated fuzz testing techniques [31].

The increased efforts to detect emulators and virtual machines have prompted researchers and practitioners to look for ways to hide the presence of such execution environments. Initial work [26] focused on removing specific artifacts in VMware that are targeted by well-known checks. Later, researchers proposed more complete sys-

tems that use virtualization [16] or dynamic translation in combination with stealth implants [38] to remain transparent to a wider range of malware checks. While these systems are successful in hiding their presence, they incur a performance penalty that is prohibitive when deploying them in large-scale automated malware analysis setups.

In addition to systems that attempt to remain transparent to malicious code, researchers have looked at ways to detect that a malware sample contains such detection checks. To perform this detection, the systems presented in [11] and [22] compare the behavior of a sample on a reference (real) host with the behavior of this sample on an analysis (or virtual) host. However, both systems simply execute the malware under analysis in two different environments. Unfortunately, as our experiments have demonstrated, re-running the same sample twice can lead to different behaviors that are not the result of any malware checks. Hence, this analysis approach is not reliable. In addition, the system presented in [22] also uses a very costly technique (Ether in fine-grained analysis mode) to produce a reference system call trace.

Process and system replay. A number of systems exist that aim at providing deterministic replay of an application or of an entire system [14]. For example, ReVirt [18] uses the virtual machine UMLinux to monitor a process and create logs of its interaction with the guest operating system. The logs are created on the host OS and can then be used to replay the entire virtual machine. However, ReVirt modifies the host and guest operating system and requires the analyzed programs to be run inside their virtual environment UMLinux. It is a powerful approach that allows comprehensive replaying, but it cannot analyze a program that contains virtual machine detection checks, since it uses a virtual machine itself. Flashback [35] is a debugging tool for Linux that allows process replaying. It creates shadow processes at various checkpoints that mirror the state of a process at a given time. In addition, the system traps system calls to log their parameter values. In contrast to our solution, Flashback provides its own system calls to enable user programs to programmatically create snapshots at certain checkpoints. For this reason, it needs to modify the operating system. Another replaying tool is Jockey [34], which inserts trampoline functions into system call code to direct the program flow to its own code where system call parameter values are recorded. These logs are then used in replay mode to reproduce the behavior of a process. Finally, Tornado [29] enables to replay the application input by intercepting and replaying the system calls. Unfortunately, a common drawback of all the four aforementioned approaches is that they run on Linux. Thus, they cannot be used to analyze Windows binaries.

ExecRecorder [15] is a virtual-machine-based log and replay framework for post attack analysis and recovery. It can replay the execution of an entire system by checkpointing the complete system state (virtual memory and CPU registers, virtual hard disk and memory of all virtual external devices) and logging all architectural nondeterministic events. ExecRecorder is based on the system emulator Bochs [25]. An advantage of ExecRecorder is that it can also run Windows in its virtual environment. However, it also suffers from the drawback that virtual machine detecting malware cannot be analyzed.

7 Conclusions

Malicious code is one of the most significant security threats on the Internet. To assess the malicious potential of the thousands of new malware binaries that are discovered every day, dynamic malware analysis systems (sandboxes) have proven to be valuable tools. As a reaction, malware authors have started to add checks to their code that detect the presence of such sandboxes. When a check determines that the malware program is analyzed, it typically hides malicious functionality or simply crashes. As a result, security analysts might mistakenly classify a binary as benign or underestimate its threat.

In this paper, we present a technique to reliably and efficiently identify malware programs that attempt to detect the presence of Anubis (which is our emulator-based sandbox) and similar tools. Our technique works by recording the system call trace of a program when it is executed on an uninstrumented reference system. Then, the binary is run on the analysis system, replaying the inputs that have been previously seen. Whenever the program shows a different behavior, we conclude that the malware has a split personality; that is, it has used CPU semantics or timing attacks to identify the presence of our sandbox. In this case, the binary can be forwarded to a more costly, but fully transparent, analysis system for further examination. Our experiments demonstrate that our system effectively and efficiently detects binaries with split personalities, while it can successfully replay programs that do not contain any checks for Anubis or the emulator (Qemu).

Acknowledgements

This work has been supported by the Austrian Science Foundation, (FWF) under grant P18764, Secure Business Austria (SBA), and the WOMBAT and FORWARD projects funded by the European Commission in the 7th Framework. Marco Cova was partially supported by a Symantec Research Labs Graduate Fellowship.

References

[1] Anubis: Analyzing Unknown Binaries. http://anubis.seclab.tuwien.ac.at, 2009.

[2] Computer Forensic Solutions. http://cfs-llc.net/index.htm, 2009.

[3] CWSandbox. http://www.cwsandbox.org/, 2009.

[4] Joebox: A Secure Sandbox Application for Windows. http://www.joebox.org/, 2009.

[5] Norman Sandbox. http://www.norman.com/technology/norman_sandbox/, 2009.

[6] ThreatExpert. http://www.threatexpert.com/, 2009.

[7] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights Into Current Malware Behavior. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the European Institute for Computer Antivirus Research Annual Conference (EICAR)*, 2006.

[9] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2009.

[10] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically Identifying Trigger-based Behavior in Malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Detection: Countering the Largest Security Threat*. Springer, 2007.

[11] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[12] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the USENIX Security Symposium*, 2003.

[13] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware Malware Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[14] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. D. Bosschere. A Taxonomy of Execution Replay Systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[15] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 66–71, New York, NY, USA, 2006. ACM.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[17] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the Windows Random Number Generator. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[19] P. Ferrie. Attacks on Virtual Machines. In *Proceedings of the Association of Anti-Virus Asia Researchers Conference*, 2007.

[20] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[21] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the USENIX Windows NT Symposium*, pages 135–144, Berkeley, CA, USA, 1999. USENIX Association.

[22] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating Emulation-Resistant Malware. In *Proceedings of the Workshop on Virtual Machine Security (VMSec)*, 2009.

[23] T. Klein. ScoopyNG – The VMware detection tool. http://www.trapkit.de/research/vmm/scoopyng/index.html.

[24] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[25] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, (29), 1996.

[26] T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, 2006.

[27] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[28] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[29] F. C. Michiel, F. Cornelis, M. Ronsse, and K. D. Bosschere. TORNADO: A Novel Input Replay Tool. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1598–1604, 2003.

[30] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[31] 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 USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[32] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proceedings of the Information Security Conference*, 2007.

[33] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://www.invisiblethings.org/papers/redpill.html`, 2004.

[34] Y. Saito. Jockey: A User-space Library for Record-replay Debugging. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging (AADEBUG)*, pages 69–76, 2005.

[35] S. M. Srinivasan, S. Kandula, S. K, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX Annual Technical Conference*, pages 29–44, 2004.

[36] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[37] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[38] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized Executions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[39] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[40] 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 ACM Conference on Computer and Communications Security (CCS)*, 2007.