How good are malware detectors at remediating infected systems?

Emanuele Passerini[†], Roberto Paleari[†], and Lorenzo Martignoni[†‡]

[†]Università degli Studi di Milano, [‡]Università degli Studi di Udine
{ema,roberto,lorenzo}@security.dico.unimi.it

Abstract Malware detectors are applications that attempt to identify and block malicious programs. Unfortunately, malware detectors might not always be able to preemptively block a malicious program from infecting the system (e.g., when the signatures database is not promptly updated). In these situations, the only way to eradicate the infection without having to reinstall the entire system is to rely on the remediation capabilities of the detectors. Therefore, it is essential to evaluate the efficacy and accuracy of anti-malware software in such situations. This paper presents a testing methodology to assess the quality (completeness) of the *remediation procedures* used by malware detectors to revert the effect of an infection from a compromised system. To evaluate the efficacy of our testing methodology, we developed a prototype and used it to test six of the top-rated commercial malware detectors currently available on the market. The results of our evaluation witness that in many situations the tested malware detectors fail to completely remove the effects of an infection.

Key words: Malware, malware detection, software testing

1 Introduction

One of the biggest problems the Internet community has to face today is the widespread diffusion of *malware*, malicious programs written with the explicit intent to damage users and to use compromised systems for various types of frauds. The second half of 2007 witnessed a drastic increase (about 135%) of the number of threats related to malware [1]. This can be ascribed to a number of different root causes, but the main reason is probably the easy financial gain malware authors obtain by selling their creations in the underground market [2]. Besides the rapid spread of malware, we are observing a parallel advance in the techniques for protecting end-users against malicious code. In order to face the growing complexity in the techniques employed by malware writers to evade detection, traditional signature-based anti-malware solutions are now being supported by behavioural, semantics-aware, approaches [3,4], that mainstream commercial products are starting to include [5,6,7].

To defend against malicious programs, users typically rely on malware detectors, which try to detect and prevent threats before the system is damaged.

Unfortunately, in many cases detection and prevention are not possible. Imagine for example a user that is not running a malware detector or a user that is running a malware detector but who gets infected before the appropriate detection signature is released. In such a situation, post-infection remediation remains the only solution to get rid of a malware and of the damages it may have caused to the system, other than reinstalling the entire system. However, the experience has taught us that sometimes automatic remediation procedures could cause more problems than they would solve [8,9].

As any kind of software application, malware detectors require thorough testing. Users do not only need a stable application, but also a product capable of detecting threats with low false-negative and false-positive rates, and capable of remediating their system from a damage caused by a malicious program that was not detected in time. For these reasons, the testing and the evaluation of a malware detector require particular attentions, to the point that the leading industries and researchers in the field have recently defined common guidelines to test this particular class of software [10]. Although these guidelines describe what should be evaluated, they do not describe any precise methodology to do that.

In this paper we address the problem of evaluating the remediation capabilities of a malware detector and we propose a fully automated testing methodology to evaluate this characteristic. The proposed methodology is dynamic. We run a malicious program in a victim system and we monitor the execution to detect what modifications are made to the environment. Subsequently, we trigger the remediation procedure of the tested malware detector to clean up the victim system. Finally, we analyse the state of the environment to verify which of the modifications previously caused by the malicious program have been successfully reverted. We have implemented the proposed methodology in a prototype and evaluated six of the most rated malware detectors on the marked. Our evaluation testifies the effectiveness of our tests and shows that the remediation procedures of the tested detectors suffers incompleteness. For example, we have empirically observed that only about 80% of the untrusted executables dropped by malicious programs on infected systems are properly removed by malware detectors.

To summarise, the paper makes the following contributions:

- a fully automated testing methodology to evaluate the completeness of remediation procedures in commercial malware detectors;
- a prototype implementation of our testing methodology;
- an empirical evaluation of six malware detectors currently available on the market, with about 100 malware samples each.

The paper is organised as follows. Section 2 motivates the importance of complete post-infection remediation. Section 3 presents the requirements of the ideal remediation procedure and sketches an overview of our testing methodology. Section 4 discusses the implementation of the infrastructure we have developed. Section 5 discusses the results of our experimental evaluation. Section 6 presents the related work. Finally, Section 7 concludes the paper.

```
WriteFile("c:\windows\poq.exe", "malicious code")
CreateProcess("c:\windows\poq.exe")
QueryKeyValue("\HKLM\...\CurrentVersion\Run", "v")  →  ""
CreateKeyValue("\HKLM\...\CurrentVersion\Run", "v", "c:\windows\poq.exe")
ReadFile("c:\...\drivers\etc\hosts")  →  "Copyri... 127.0.0.1 localhost"
WriteFile("c:\...\drivers\etc\hosts", "67.23.124.83 www.google.com\n
         67.23.124.85 www.citi.com\n")
DeleteFile("c:\malware.exe")
```

Figure 1. High-level execution trace of a sample malicious program (`malware.exe`)

2 The importance of remediation

To comprehend why remediation is a key issue in defeating malware, let us consider a sample malicious program. Figure 1 shows a fragment of an execution trace of the sample malware, reporting the most important modifications to the system performed by the application. The malicious program replicates itself into a new executable (`c:\windows\poq.exe`), creates a registry key to configure the system to start the new executable automatically at boot, and tampers the configuration of the resolver (writing into `c:\windows\system32\drivers\etc\hosts`) to hijack network traffic, directed to `www.google.com` and `www.citi.com`, to a malicious web site. Moreover, let us imagine a user whose system gets infected by this malware and that, at the time of infection, his system was not properly protected (e.g., the infection took place before a signature for detecting the malware was released). Only after a while, when the appropriate signature becomes available, the malware detector can detect the presence of the malware on the system and can remediate the damages.

What the user expects from the detector is that it is able to remediate completely the system. That is, the malware detector has to revert *all* the modifications made to the system by the malicious program. In the case of the example, that means that the original malicious executable (`malware.exe`), the executable created (`c:\windows\poq.exe`), and the registry key (`\HKLM\Software\Microsoft\Windows\CurrentVersion\Run\v`) have to be removed from the system. Similarly the process started has to be killed, and the malicious entries added to the configuration of the resolver removed (`c:\windows\system32\drivers\etc\hosts`).

If the remediation procedure is not complete, the system can be left in an unsafe state. Imagine for example that the malware detector reverts all the actions performed by the malicious program, but that it is not able to restore the proper configuration of the resolver (i.e., to remove the malicious entries added to the file `c:\windows\system32\drivers\etc\hosts`). Even though all the malicious executables dropped by the malware are removed from the system, the security of the user is still compromised because part of the network traffic is hijacked to a malicious web site. This site can be used to steal sensitive information or to deliver new malware to the user.

System call trace ($\mathcal{S}$)	High-level behaviour ($\mathcal{T}$)

`NtCreateFile("...\poq.exe") → f`
`NtWriteFile(f, "malicious code")`
`NtWriteFile(f, "other malicious code")`
`NtClose(f)`
	`WriteFile("...\poq.exe", "malicious ...")`

`...`
`NtOpenFile("...\poq.exe") → f`
`NtCreateSection(...) → s`
`NtMapViewOfSection(h, s)`
`NtCreateProcess(h) → p`
`NtCreateThread(p) → t`
	`CreateProcess("...\poq.exe")`

`...`
`NtOpenKey("...\Run") → r`
`NtQueryValueKey(r, "v") → FAILURE`
`NtSetValueKey(r, "v", "...\poq.exe")`
`NtClose(r)`
	`CreateKeyValue("...\Run", "v", "...\poq.exe")`

`...`
`NtOpenFile("...\etc\hosts") → f`
`NtReadFile(f, 1024) → "Cop..."`
`NtWriteFile(f, "67... www.google...")`
`NtWriteFile(f, "67... www.citi...")`
`NtClose(f)`
	`WriteFile("...\hosts", "67... www.citi.com\n")`

`...` `NtDeleteFile("c:\malware.exe")`	`DeleteFile("c:\malware.exe")`

Figure 2. System call trace of our sample malicious program (`malware.exe`) and corresponding high-level execution trace.

3 Testing methodology

This section defines the ideal remediation procedure (Section 3.1) and presents the testing methodology we have developed to verify whether the remediation procedures available in a malware detector resemble the ideal one or not (Section 3.2).

3.1 The ideal remediation procedure

For the purpose of defining the ideal remediation procedure, we can think the execution of a malicious program as characterised only by interactions with the environment, where each interaction corresponds to the invocation of a particular OS routine (or system call). Let $\mathcal{S} = \langle s_0, s_1, \ldots, s_n \rangle$ be the execution trace of our malware sample. The system calls in $\mathcal{S}$ can be classified in two classes: those that modify the state of the environment and those that do not. For example, to replicate itself into a system folder, a malicious program has to create a file and to copy its code into the file. Similarly, to install itself at boot, the program has to create a particular registry key. Both activities involve a modification of the state of the environment. On the other hand, a program that reads and parses the content of a file does not alter the state of the environment. For our purpose, it is sufficient to consider only a subset of all the system calls executed by the malicious program, including only the ones that modify the state of the local environment: $\mathcal{S}' = \langle s_j \in \mathcal{S} : s_j$ contributes to modify the state of the local system$\rangle$.

To achieve a particular high-level goal, the malicious program has to execute multiple system calls. As an example, to replicate itself, the program has to create a file and then to write its payload into the file (typically in multiple passes). Nevertheless, for remediating a system from a malware infection, it is not important to know which system calls the malicious program executed to modify the system, but instead what modifications were made to the local system by the program. For this reason, we can abstract the sequence of system calls $\mathcal{S}'$ executed by the malicious program to infect the system through a set of *high-level system state transitions* $\mathcal{T}$. Each transition $t \in \mathcal{T}$ represents the effect on the local system produced by the execution of a sequence of related system calls. Let us consider again our sample malicious behaviour of Figure 1 and the corresponding system calls trace shown in Figure 2, where each high-level behaviour is associated with the sequence of system calls executed by the malware and that produces a particular state transition. In the figure, irrelevant system calls (i.e., the system calls that do not modify the state of the system) are reported in gray. As an example, to create a file on the file system (which consists in a copy of the malicious program) the following system calls are executed: `NtCreateFile`, `NtWriteFile`, and `NtClose`. The high-level state transition associated with this sequence of system calls is the creation of a new file on the system.

The set of high-level system state transitions $\mathcal{T}$ can be divided in multiple classes, each of which represents a state transition involving a particular class of OS resource. For example, for a Microsoft Windows system we have $\mathcal{T} = F \cup R \cup P \cup S$ where: F represents the state transitions involving files, R the state transitions involving registry keys, P those involving processes, and S those involving system services. This separation is important because each class of state transition requires a specific mechanisms for remediation. It is worth pointing out that, in our context, we are interested only in the state transitions that modify the local system, as no remediation could be accomplished for transitions that affect remote hosts. Furthermore, we do not consider system state transitions caused by other benign processes that might be running in the test environment.

A remediation procedure $\mathcal{P}$ is *complete* if it is able to revert all the effects (i.e., the high-level state transitions) of the execution of the malware: $\forall\, t \in \mathcal{T}$, t is reverted by $\mathcal{P}$. The ideal remediation procedure is the one that is complete. Reverting a particular state-transition means to bring the state of the system back to that preceding the transition. Practically speaking, if a malicious program creates a file we expect the malware detector to remove the file; if the malicious program reconfigures the resolver, we expect the malware detector to adjust the configuration of the resolver.

3.2 Testing the completeness of a remediation procedure

Testing scenarios. The following paragraphs present two real-world scenarios that resemble the one we use to perform the testing of a malware detector. The first scenario involves a system protected by a conventional malware detector, while the second one involves a system protected by a behaviour-based detector.

Scenario 1 – Conventional malware detector. A user's system gets infected by a malicious program because the conventional (signature based) malware detector running on the system is not able to promptly detect and to prevent the infection (e.g., because the appropriate signature has not been published yet). Only later, the malware detector detects the presence of the malicious program on the system and cleans the system to get rid of the threat.

Scenario 2 – Behaviour-based malware detector. A user is running a behaviour-based malware detector on his system. The system is infected by a malicious program but the detector does not detect it until any malicious activity is observed. For example, consider malicious program that creates some files on the system and then tries to infect a running process. As the initial activity is legitimate, the malicious program is blocked only when it tries to infect other processes (or after the infection has taken place). The malware detector, after having detected the malicious behaviour, repairs the system to rollback all the potentially dangerous activities performed before the detection.

Overview of the testing methodology. Our goal is to measure remediation capabilities of the detector in any of the aforementioned scenarios. To accomplish this goal, we select a set of sample malware and we use each of these programs to infect a test system, we let to the detector to remediate the damages caused to the system by each infection, and finally we check the state of the system to see if the detector was able to revert the state to that prior to the infection. In other words, by infecting our test system with a malicious program we identify the set of system state transitions which are direct consequences of the infection and then we use these information to measure the completeness of the remediation procedure.

A generalisation of our testing methodology is outlined in Figure 3 and is summarised in the following paragraphs.

(P_1) – *Execute and trace the malicious sample.* We select a malicious program we know in advance is detected by the malware detector under testing, and we run it in the test system. To simulate the scenario involving a conventional malware detector it is sufficient to disable the detector temporarily. On the other hand, to simulate the scenario involving a behaviour-based detector the malicious program is run with the detector enabled. The execution is stopped when a timeout is reached or when the behaviour-based malware detector detects a malicious behaviour. As the execution of the malicious program is monitored by an external monitor, at the end of the execution we obtain $\mathcal{S}$, the complete trace of the system calls invoked by the program during the execution.

(P_2) – *Freeze malicious processes.* We freeze the state of the malicious program to prevent it from further altering the state of the system. Subsequent steps of the analysis will refer to that state.

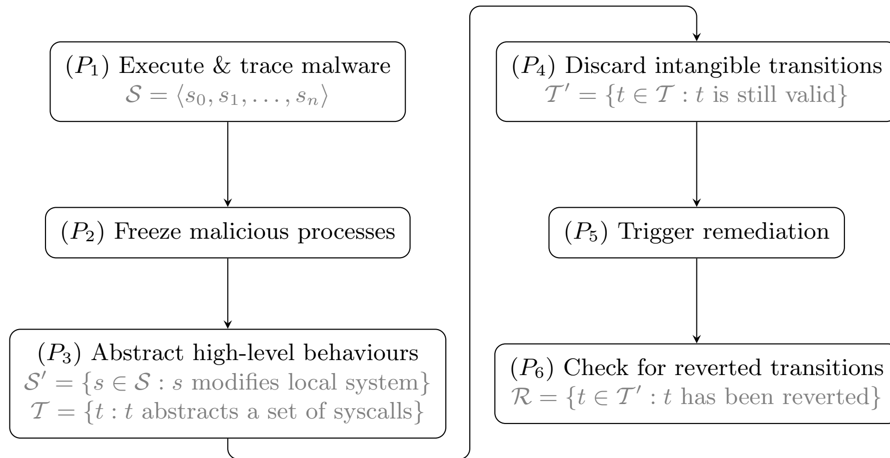

Figure 3. Overview of our testing methodology. In gray we report the outcome of each phase.

(P_3) – *Abstract high-level behaviours.* We analyse the recorded execution trace $\mathcal{S}$ to extract $\mathcal{S}'$, by excluding all the system calls that do not alter the state of the system (e.g., those used to open a file in read-only mode, or to read a registry key). Then, we analyse the resulting trace to infer the high-level behaviours of the program and the corresponding set $\mathcal{T}$ of high-level system state transitions.

It is worth noting that we analyse only the behaviour of the malicious process, and its children, and we do not consider high-level state-transitions associated with other processes running concurrently on the system. Thus, some of the high-level state transitions we analyse could conflict with those associated with other processes. To mitigate this problem without increasing the complexity of the analysis, we trace the malicious program in highly passive environments, with a minimal number of potentially conflicting processes and with no user interaction at all.

(P_4) – *Discard intangible transitions.* Not all the observed high-level program behaviours lead to tangible system state transitions. As an example imagine our sample malicious programs that deletes the original executable after it has replicated. It is important to preemptively detect intangible state transitions because otherwise one might think that the transitions is reverted by the remediation procedure. For this reason, we identify such transitions and filter them out. The next phases of the testing will target only tangible transitions: $\mathcal{T}' = \{t \in \mathcal{T} : t$ is tangible on the test system$\}$.

(P_5) – *Trigger remediation.* Having collected all the information necessary to test the completeness of the remediation procedure, we can now trigger the malware detector to remediate the infection and to cleanup the system. In the case of a conventional detector we have to launch a full-system scan, which includes

the scanning of all files and running processes. In the case of a behaviour-based detector we have to authorise the detector to quarantine the malicious program; recall the behaviour-based detector has been active since the beginning of the execution of the malicious program and it has already blocked the execution of the program.

(P_6) – *Check for reverted transitions.* Once the malware detector has completed the remediation, we have to check whether each of the high-level state transitions $t \in \mathcal{T}'$ has been properly reverted. Practically speaking, that means that we have to compare the state of the system prior to the infection with the state after the infection and the remediation, to detect any mismatch that can be ascribed to the malicious program. It is worth pointing out that we cannot expect the conventional malware detector to revert state transitions that caused data loss. On the other hand, it is legitimate to expect that from the behaviour-based malware detector, as it has observed the whole execution of the malicious program since the beginning. At the end of this phase, we obtain a set $\mathcal{R} \subseteq \mathcal{T}'$ of abstract transitions that have been reverted by the malware detector. If the remediation procedure is complete, then $\mathcal{R} = \mathcal{T}'$; instead, if $\mathcal{R} \subset \mathcal{T}'$, then every transition $t \in \mathcal{T}' \setminus \mathcal{R}$ testifies the incompleteness of the remediation procedure for the malicious program used for the testing. It is worth noting that $\mathcal{R}$ could also include some state transitions that are not in $\mathcal{T}$. This happens when the malware detector incorrectly attributes a spurious action to the malicious program [8]. However, as our analysis is driven by the observed behaviours, we do not handle this situation.

4 Implementation

We have developed a prototype that implements the testing methodology discussed in the previous section, specific for testing malware detectors for Microsoft Windows. In this section, we discuss the technical details regarding the implementation of our testing infrastructure. The methodology described previously can be used to test the completeness of remediation procedures of both conventional and behaviour-based malware detectors. In the following, we describe in detail only the implementation specific for the testing of conventional detectors. Nevertheless, the implementation for behaviour-based detectors only differs in the fact the detector is active when the malicious program is executed and traced.

Figure 4 depicts our testing infrastructure. The main components of our architecture are the victim test system, where the malware sample and the detector are located, and the analysis environment, where execution traces are analysed. The malicious sample is uploaded into the test machine and its execution is monitored. Syscall traces are subsequently analysed in the analysis environment, and further abstracted into high-level state transitions that are then verified. Finally, the malware detector is allowed to scan the whole system, and then the state of the system is checked to detect the set of transitions that have been reverted.

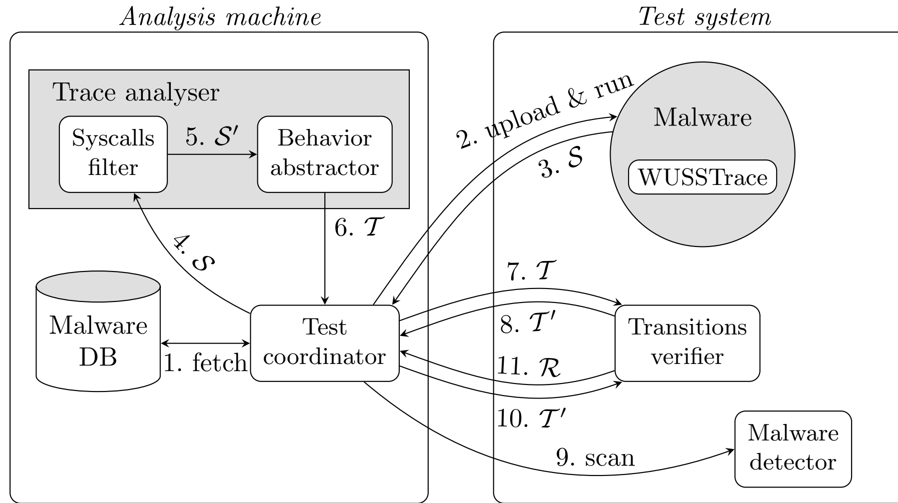

Figure 4. Architecture of the testing infrastructure

4.1 Tracing the malware sample

The malware sample is executed and traced in the test system (steps 1–3 in Figure 4). For the tracing we relay on our home made system call tracer, codenamed WUSSTrace [11], a user-space system call tracer for Windows. WUSSTrace parses the majority of the arguments of system calls, thus allowing a subsequent fine-grained analysis of the behaviour of the program. Each intercepted system call is logged into an easy-to-parse XML trace, together with its input and output arguments. If the monitored process creates other processes or threads, these are monitored recursively. We are aware that user-space tracing can be easily circumvented by a nasty malware and that safer solutions exist (e.g., hooking from kernel space or through virtual machine introspection). However, we made this decision only to ease the development of our prototype.

We set a timeout on the execution of the malicious program and the other processes it creates. If a monitored process does not terminate spontaneously before the timeout expires, we freeze the process, by suspending the execution of all its threads. By freezing the malicious process instead of terminating it, we allow the malware detector to operate in a "best-case scenario", where it can apply in-memory scanning techniques to analyse the memory image of the processes and to apply all the available heuristics.

4.2 Analysis of the system call trace

In order to analyse the system calls issued by the monitored malware sample, we developed a trace analysis tool that off-line performs the abstractions needed to infer the high-level program behaviours and the corresponding system state transitions. In our current implementation, we focus on the identification of

the files, registry keys, processes, and system services that have been created or tampered by the malicious sample or by any of its child processes. For this reason, starting from a trace $\mathcal{S}$, we obtain (steps 4 and 5 in Figure 4) the set of system calls that modify the state of the environment ($\mathcal{S}'$) by including only those syscalls that lead to the system state transitions of interest: file-system modifications (e.g., `NtCreateFile`, `NtOpenFile`, `NtWriteFile`), modifications of registry keys (e.g., `NtCreateKey`, `NtSetValueKey`), process creation or infection (e.g., `NtCreateProcess`, `NtOpenProcess`), *etc.*

To abstract $\mathcal{S}'$ into high-level behaviours and the corresponding set of state transitions $\mathcal{T}$, we need to correlate together the system calls that contribute to the same high-level behaviour (step 6). In order to identify the syscalls responsible for a particular behaviour (i.e., those that operate on the same resource) we employ standard data-flow analysis techniques [12]. The data-flow analysis is not fine-grained, as we do not log every single machine instruction executed by the monitored processes. Thus, dependency relationships between system calls are identified through handles (i.e., Windows resources identifiers): if system call s_2 uses handle h and the system call s_1 is the (dynamic) reaching definition for h, then we can assume that s_1 and s_2 operate on the same resource. As an example, when we find in the execution trace a `NtSetValueKey(r, "v", "...\poq.exe")` system call we need to determine the name of the key that is being written; for this purpose, we compute the dynamic backward slice for the key handle `r` and we analyse the arguments of the system call that originally defined it [13]. Similarly, in order to compute the name of the files that are actually modified by the malware, we calculate the dynamic reaching definition for the file handle `f` used by the system call `NtWriteFile(f, "...")`; this reaching definition will correspond to a `NtCreateFile` or `NtOpenFile`, and through the analysis of its input arguments we can infer the name of the file being written.

4.3 Filtering of intangible high-level transitions

Having built the set $\mathcal{T}$ of high-level state transitions that represent the modification of the system caused by the malicious program, it is important to ensure that each transition $t \in \mathcal{T}$ is valid (i.e., it represent an actual modification of the state of the environment, that is tangible after the malicious program has terminated or it has been frozen). Indeed, any spurious state transition must be discarded, as it could negatively affect the accuracy of the evaluation of the remediation procedure.

As an example consider again our sample malicious program, whose high-level behaviour is summarised in Figure 1. The program replicates its payload and then deletes the original executable. When the execution of the program in the test system terminates (or is frozen) the executable no longer exists on the system. If we do not test whether the file still exists on the system prior to the invocation of the malware detector we might erroneously praise the malware detector for something it has not done. On the other hand we want to be sure that system state-transition, even if not annihilated by the malicious program itself,

are effectively tangible. The assumption that each write access to a resource of the system produces a modification of the system state might be too broad. For example, several malware often overwrite registry keys with the actual content of the keys; thus, despite the keys are overwritten, the system state does not mutate (this is probably a side effect caused by the use of some high-level libraries). A similar situation might occur with memory mapped files, because these files are written without invoking system calls and thus we have to conservatively assume that a file mapped with write permission is eventually modified.

We identify intangible state transitions by querying directly the test system in the exact same way we query it to detect if a transitions has been reverted by the malware detector (steps 7 and 8). Only tangible transitions $\mathcal{T}' \subseteq \mathcal{T}$ are targeted by the remaining phases of the testing. We detect registry keys or files that are effectively modified by comparing their actual content with their content preceding the infection. To do that we maintain a database of hashes of the content of all files and registry keys of the test-system before the infection. We discard all the behaviours that preserve the content of these resources. Similarly we also discard all the behaviours that involve the creation of files, registry keys, and processes that cannot be found on the test system at the end of the execution of the malicious program. Further details about how the test system is queried are given in the next paragraphs.

4.4 Execution and evaluation of the remediation procedure

At this point it is possible to trigger the malware detector to analyse the system and clean it from the infection. We invoke it to perform a full-scan of the file system, of the registry, and of the image of running processes (step 9 in Figure 4). We also enable all the heuristics supported to improve the detection and remediation rate. When the detector terminates the analysis of the system, we verify which of the state transitions associated with the execution of the malicious program have been reverted (step 10 and 11). Recall that the system state transitions $\mathcal{T}'$ can be divided in multiple classes according to the type of resource affected by a transition. That is, $\mathcal{T}' = F \cup R \cup P \cup S$, where F, R, P, and S are the classes of transitions involving respectively files, registry keys, processes, and system services. Each class of transitions requires a particular procedure to verify whether the transition has been reverted or not. A transition $t \in \mathcal{T}'$ is considered to be reverted by the malware detector when one of the following conditions is satisfied:

- if $t \in F$, the file subject of the transition is deleted or modified by the malware detector;
- if $t \in R$, the registry key subject of the transition is removed or modified by the malware detector;
- if $t \in P$, the process spawned by the malicious program is terminated;
- if $t \in S$, the system service created by the malicious program is disabled.

Note that we optimistically assume that any modification made by the remediation procedure to a resource manipulated by the malicious program successfully restores the initial state of the resource.

To test the aforementioned conditions, we leverage a small helper program we run in the test system, that allows us to query the state of a particular resource. For example, if we have observed the malicious program to create a registry key, we query the helper to check whether the key still exists on the system and, if so, to retrieve its contents and perform the appropriate comparisons.

5 Experimental results

This section presents the results of the testing of six of the top-rated commercial malware detectors. The goal of our experimental evaluation was to prove the effectiveness of the proposed testing methodology and not to compare the tested malware detectors to tell which was the best and which was the worst. The experiments witnessed the effectiveness of our testing methodology. Indeed, they highlighted that none of the tested malware detector has complete remediation procedures. Furthermore, the experiments showed that the type and percentage of system state transitions reverted varies substantially among detectors.

5.1 Experimental setup

We tested the following malware detectors: Avast Professional 4.8, Kaspersky Anti-virus 2009, McAfee VirusScan Enterprise 8.5.0, Nod32 Anti-virus 3.0, Panda Anti-virus 9.0.5 and Sophos Anti-virus 7.6. We selected the malware detectors that facilitated the most the batch analysis, that is, those invokable directly from the command line and with the ability to cleanup the system automatically. We assumed that the detection capabilities of the command line version (with the proper arguments) and the GUI version corresponded. The virus definitions of each product were last updated on 15 January 2009. To discourage any direct comparison among the malware detectors, they were tested using different sets of about 100 malware samples, chosen randomly from a corpus composed by several thousand samples collected in the last quarter of 2008. All the samples tested were detected by the six detectors.

We performed the evaluation of our testing methodology using as test systems multiple VirtualBox virtual machines, each one running a different malware detector. To prevent other processes to alter the state of the system resources affected by the malicious programs used for the testing, we stripped down the virtual environments used for the analysis: we stopped all unnecessary services and processes and we did not interact at all with the environments. We traced the execution of the selected malicious program for five minutes and we performed all the steps of the analysis without restarting the test system. After each test, we restored the original clean state of the virtual machine.

5.2 Evaluation of state-of-the-art malware detectors

Figure 5 presents the overall results of our experiments. The names of the malware detectors have been anonymised to discourage comparisons. For each malware detector, we report the average percentage of system state transitions that

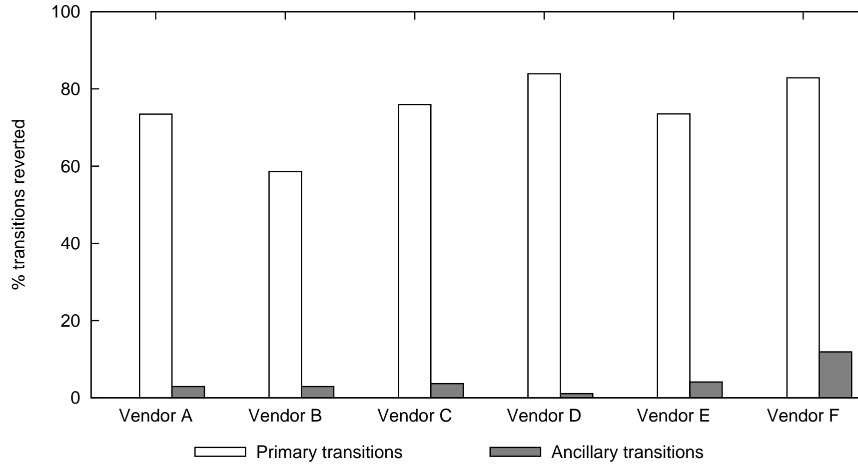

Figure 5. Average percentage of system state transitions reverted by each malware detector.

were reverted. The average is computed on the total number of malware used to test each detector. The transitions are separated in two groups, according to their security impact on the system: primary and ancillary. Primary transitions are those that have a high impact on the system, while ancillary transitions have a minor impact. A user should expect all primary transitions to be reverted by the malware detector, while he could tolerate if some ancillary transitions were not reverted. The partitioning of transitions in the two groups has a certain degree of subjectivity. We divided each of the transitions classes F, R, P, and S in primary and ancillary as follows:

- F (files): we consider as primary transitions all those that involve executable files (e.g., `.exe`, `.dll`, `.bat`, `.pif`, `.scr`), while as ancillary those involving the remaining types of files.
- R (registry keys): we consider primary transitions those that involve registry keys that can be used to start programs automatically and ancillary all the remaining ones.
- P (processes): we consider primary the transitions that create processes where the executed files match any of the files dropped on the system by the malicious program; the remaining processes started by the malware but executing programs already present on the system are instead considered ancillary.
- S (services): for simplicity we treat system services as normal processes.

The graph in Figure 5 clearly shows the effectiveness of our testing methodology at evaluating the completeness of remediation procedures. None of the tested malware detectors turned out to be complete, even if only primary transitions are taken into account: about 75% of the total primary transitions and only 4% of the total ancillary transitions were reverted.

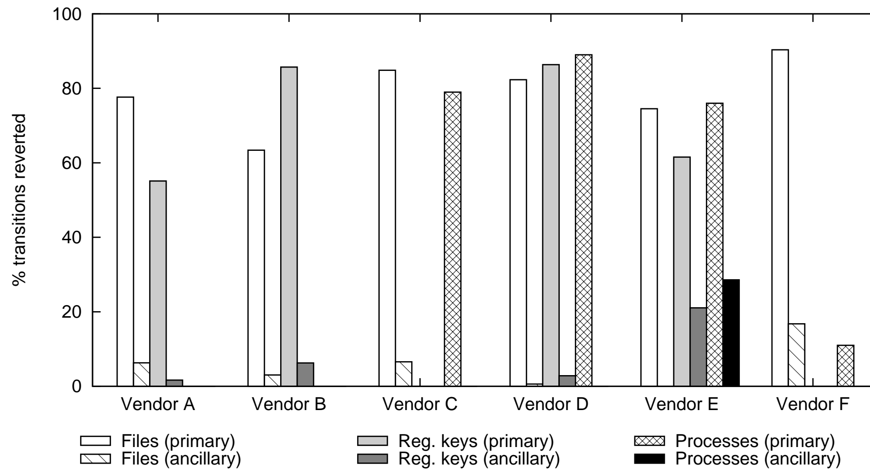

Figure 6. Average percentage of primary and ancillary state transitions, partitioned by the system resources involved, reverted by each malware detector.

A more detailed overview of the average distribution of primary and ancillary system state transitions reverted, for each transition class, is reported in Figure 6 (product names have been anonymised). While all malware detectors reverted the majority of primary transitions involving files, some of them (e.g., Vendor C and Vendor F) did not revert transitions involving registry keys at all. Other detectors instead (e.g., Vendor A and Vendor B) did not seem to terminate malicious processes, although we did not check the state of the system after a reboot.

We did not test interactively whether the system continued to work properly after infection and remediation. Indeed, there could exist situations in which an incomplete or improper remediation might render the system unusable. For example, imagine a malicious program that creates a registry key pointing to an executable, and that the existence of the key mandates the existence of the file (e.g., in Windows XP, the `Image File Execution Options` registry key). If the executable were removed, but the key were not, the system would stop working. We plan to address this problem in the future.

6 Related work

In this section we briefly review the work done by the research community on malware detection and analysis. We also present some recent results that focus on execution of untrusted applications without any risk for the system and on the problem of testing malware detectors.

6.1 Malware detection and analysis

The traditional approach for the detection of malicious code is based on signature matching of various complexity [14]. A signature can be a sequence of bytes that identifies pieces of data or code of the malicious program, but even very complex algorithms that test whether a particular program satisfies certain properties. The advantage of using sophisticated detection methods is that signatures become more generic and thus a single signature can be used to detect multiple variants derived from the same family. On the other hand, from the remediation point of view, excessively generic signatures do not allow to distinguish variants. If single variants cannot be told apart, the remediation procedure cannot take variant-specific behaviours into account and cannot perform a complete cleanup.

Purely signature-based approaches have demonstrated their weaknesses when packed, polymorphic and metamorphic malware appeared. The research community started to move toward behaviour-based solutions. Behaviour-based detection [3,4] and analysis [15,16,17,18] approaches do not focus on the syntactic structure of the analysed program, but try to consider its semantics. Because these solutions work by observing a concrete execution of the malicious sample, they could provide much more accurate remediation procedures.

6.2 Execution of untrusted applications

In [19], Hsu *et al.* present a framework to automatically remove a malicious program from the system and also to repair any damage it could have done. The safe state of the system is restored by using the logs of the execution and by reverting each logged operation. An alternative approach is proposed by Liang *et al.* [20]. Untrusted programs are executed in a sandbox and the changes made to the "virtual" system are committed to the real one at the end of the execution, only if the program can be considered innocuous.

In the operating systems and self-healing communities, a number of different works investigate the problem of automatically reverting the modifications made by an unwanted program. As an example, in [21] the authors present Speculator, a modified Linux kernel that allows speculative execution of user-space processes. Speculator avoids blocking user processes during slow I/O operations (such as remote I/O operations): the system predicts the operation's result, checkpoints the process and allows it to continue; later, if the prediction is found to be incorrect, the process is reverted to the checkpointed state.

6.3 Evaluation of state-of-the-art malware detectors

The need for automatic testing methodologies targeting anti-malware products has been clearly stated by the Anti-Malware Testing Standards Organisation (AMTSO) [10]. However, little research work focuses on the evaluation of malware detection and remediation solutions. One of the few examples is represented by [22]; in this paper, Christodorescu *et al.* present a technique for generating test-cases to stress malware detectors. They use program obfuscation techniques

to evaluate the resilience of malware detectors to various transformations of the malicious code. The goal of our paper instead is to estimate the completeness of remediation procedures. For this reason, the testing infrastructure described in our paper could complement their work, in order to produce more comprehensive testing methodologies.

Other researchers highlighted the importance cleaning infected systems and the importance of testing such functionality [23,24]. Motivated by the same convictions, this paper contributes to address this problem by proposing a fully automated testing methodology and an extensive evaluation of several state-of-the-art commercial products.

7 Conclusions

Malware detectors are essential components for preserving the security of computer systems. They allow to detect and prevent malicious software and, when malware cannot be stopped from infecting a system, they allow to recover from the infection. In this paper we presented an automated testing methodology to assess the completeness of remediation procedures used by malware detector to clean up compromised systems. We used this methodology to test six of the most rated malware detectors on the market and found out that the dangerous effects of an infection are seldom completely removed. In the future we plan to investigate automatic techniques for generating more complete remediation procedures.

References

1. Symantec Inc.: Symantec internet security threat report: Volume XIII. Technical report, Symantec Inc. (April 2008)
2. Franklin, J., Perrig, A., Paxson, V., Savage, S.: An inquiry into the nature and causes of the wealth of internet miscreants. In: Proceedings of the 14th ACM conference on Computer and communications security (CCS'07), New York, NY, USA, ACM (2007) 375–388
3. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005), Oakland, CA, USA, ACM Press (May 2005) 32–46
4. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection, RAID, Cambridge, Massachusetts, U.S.A. Lecture Notes in Computer Science, Springer (September 2008)
5. NovaShield Inc.: NovaShield Anti-Malware `http://www.novashield.com/`.
6. Sana Security: Primary Response SafeConnect `http://www.sanasecurity.com/`.
7. PC Tools: ThreatFire AntiVirus – Behavioral Virus and Spyware Protection `http://www.threatfire.com/`.
8. Slashdot: AVG virus scanner removes critical Windows file `http://tech.slashdot.org/article.pl?sid=08/11/10/2319209`.

9. Heise Media: Bitdefender and GData delete winlogon system file http://www.h-online.com/security/Bitdefender-and-GData-delete-winlogon-system-file--/news/112652.
10. AMTSO: Fundamental principles of testing http://www.amtso.org/documents/doc_download/6-amtso-fundamental-principles-of-testing.html.
11. Martignoni, L., Paleari, R.: WUSSTrace – a user-space syscall tracer for Microsoft Windows http://security.dico.unimi.it/projects.shtml.
12. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer-Verlag (1999)
13. Agrawal, H., Horgan, J.R.: Dynamic Program Slicing. In: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, USA (June 1990) 246–256
14. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional (February 2005)
15. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR). (2006)
16. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security & Privacy **5**(2) (March/April 2007) 32–39
17. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: Proceeding of the 2007 IEEE Symposium on Security and Privacy, Oakland, California, USA, IEEE Computer Society (2007) 231–245
18. Yin, H., Song, D., Egele, M., Kirda, E., Kruegel, C.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS, Alexandria, VA, USA, ACM (2007)
19. Hsu, F., Chen, H., Ristenpart, T., Li, J., Su, Z.: Back to the Future: A Framework for Automatic Malware Removal and System Repair. In: Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06), Washington, DC, USA, IEEE Computer Society (2006) 257–268
20. Liang, Z., Venkatakrishnan, V.N., Sekar, R.: Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In: Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC '03), IEEE Computer Society (2003) 182–191
21. Nightingale, E.B., Chen, P.M., Flinn, J.: Speculative execution in a distributed file system. In: Proceedings of the twentieth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2005) 191–205
22. Christodorescu, M., Jha, S.: Testing malware detectors. SIGSOFT Software Engineering Notes **29**(4) (2004) 34–44
23. Bruce, J.: The Challenge of Detecting and Removing Installed Threats. In: Virus Bulletin Conference. (October 2006)
24. Morgenstern, M., Marx, A.: System Cleaning: Getting Rid of Malware from Infected PCs. In: Virus Bulletin Conference. (June 2008)