Automatic Generation of Remediation Procedures for Malware Infections

Roberto Paleari[1], *Lorenzo Martignoni*[2], *Emanuele Passerini*[1],
Drew Davidson[3], *Matt Fredrikson*[3], *Jon Giffin*[4], *Somesh Jha*[3]

[1]*Università degli Studi di Milano*
{`roberto, ema`}`@security.dico.unimi.it`

[2]*Università degli Studi di Udine*
`lorenzo.martignoni@uniud.it`

[3]*University of Wisconsin*
{`davidson, mfredrik, jha`}`@cs.wisc.edu`

[4]*Georgia Institute of Technology*
`giffin@cc.gatech.edu`

Abstract

Despite the widespread deployment of malware-detection software, in many situations it is difficult to preemptively block a malicious program from infecting a system. Rather, signatures for detection are usually available only after malware have started to infect a large group of systems. Ideally, infected systems should be reinstalled from scratch. However, due to the high cost of reinstallation, users may prefer to rely on the remediation capabilities of malware detectors to revert the effects of an infection. Unfortunately, current malware detectors perform this task poorly, leaving users' systems in an unsafe or unstable state. This paper presents an architecture to automatically generate *remediation procedures* from malicious programs—procedures that can be used to remediate all and only the effects of the malware's execution in any infected system. We have implemented a prototype of this architecture and used it to generate remediation procedures for a corpus of more than 200 malware binaries. Our evaluation demonstrates that the algorithm outperforms the remediation capabilities of top-rated commercial malware detectors.

1 Introduction

One of the most pressing problems faced by the Internet community today is the widespread diffusion of malware. To defend against malware, users rely on signature- or behavior-based anti-malware software that attempts to detect and prevent malware from damaging an end-host. Unfortunately, in many cases detection and prevention are not possible. Malware authors have perfected the practice of automatically creating a large number of *variants*, or malware that appears new to detectors but exhibits the same behavior when executed. For new malware and variants, signatures for detection are rarely available by the time malware reaches a network, leaving a time window in which systems are susceptible to infection. In these situations, the ability to detect and remove the malware after infection is not enough—it is also imperative that any harmful changes to the system made by the malware are *remediated* (or reverted).

The safest way to remediate a system is to format the permanent storage and re-install the operating system from scratch. While effective, this approach is also costly and usually results in a loss of valuable personal data, particularly when data backups are incomplete or non-existent. Rather, end-users and administrators may prefer to remove only those resources left behind by the malware, leaving the rest of the system intact. Unfortunately, current anti-malware products perform poorly at this task. A recent study demonstrated that even top-rated commercial anti-malware software fails to revert the effects of all the actions performed by malware during infections [15]. Needless to say, partially-remediated systems are unstable and prone to error.

In this paper, we present a system that automatically generates *remediation procedures* from malware binaries. These remediation procedures can be executed on infected systems to restore the state to a clean configuration, and are capable of remediating the effects of a malware sample *a posteriori*, without observing the infection take place. The fact that our remediation procedures are generated to cover a particular malware binary, rather than a specific sequence of system events resulting in an infected state [7, 19], amounts to a substantial break from previous technologies. Using our system, one can generate a single general-purpose executable that is capable of reversing the effects of a malware sample on an arbitrary number of hosts *after the fact*. In other words, one does not need to be aware of our system, or make use of it, until *after* the infection takes place. To achieve this goal, we rely on a combination of dynamic program analysis and semantic generalization to produce models of infection behavior that are resilient to common malware anti-analysis techniques, such as the use of nondeterminstic file names or the omission of malicious behavior on some runs of the program. Then, we translate these behavior models directly into executable procedures that remediate the effects of a malware infection.

We have implemented our ideas in a prototype tool. Using the prototype, we automatically generated remediation procedures on a corpus of more than 200 binary malware samples belonging to approximately 50 distinct families. We evaluated the practical effectiveness of each procedure by testing its ability to recognize all of the harmful effects of a malware execution (*true positives*) while leaving benign aspects of the system intact (*true negatives*). The results of our evaluation attest to the effectiveness of our technique: *in total, we reversed 98% of the harmful effects while generating only a single false positive*, although we were not able to remediate user-specific resource changes such as deleted documents and personal file mutations. In contrast, the best commercial anti-malware product remediated only 82% of the effects of our corpus.

In summary, we make the following contributions:

- We present an architecture to automatically generate remediation procedures given binary malware samples. To the best of our knowledge, our architecture is the first to work under the assumption that information relating to a specific infection is not available; rather, characteristic infection patterns are observed and *generalized* to produce effective procedures in this setting.

- We evaluated an implementation of our framework on on more than 200 real malware samples and found that it was able to remediate the resulting infections more effectively than existing commercial antivirus products. We have made this implementation available as an open-source package [1].

The rest of this paper is organized as follows: In Section 2 we discuss related work. In Section 3.1, we describe the problem that our architecture solves by presenting a realistic example, and in Section 3.2 we outline our approach, relating it to the example. In Section 4, we formalize the problem of malware remediation and present the technical details of our approach. In Section 5, we evaluate the effectiveness of our approach by testing a prototype implementation against real malware. In Section 6 we discuss the limitations of our approach, the security implications, and potential avenues for future work. We present concluding remarks in Section 7.

2 Related Work

Our contributions relate to ongoing research on behavior-based malware analysis, on the execution of untrusted applications in trusted systems, and on the automatic generation of signatures to detect malicious network traffic.

Behavior-Based Malware Analysis: The prevalence of packed, polymorphic, and metamorphic malware highlights the deficiencies of traditional detection approaches based on syntactic signatures. This has urged researchers and security practitioners to focus on solutions that base policy on the behavior exhibited by untrusted software. Behavior-based techniques attempt to infer security-relevant information about an untrusted program either by analyzing it statically [16] or by observing its operation dynamically [1, 11, 21]. The major drawback of current behavior-based techniques is their high computational overhead. Recently, Kolbitsch et al. developed an efficient analysis solution intended to replace traditional anti-malware on the desktop [8]. Closer in spirit to the work presented here is that of Christodorescu et al. [4]. They described an automatic approach that derives formal specifications of malicious behavior by comparing the observed dynamic behavior of malicious and benign applications. Their technique uses dependence graphs, which express the relationships among various low-level behavior events, and is similar in many ways to our high-level behavior abstraction component (see Section 4.2.1). Another area of much recent activity is that of automatic classification of malware into families [2, 18]. For that type of work, malware is grouped into clusters, which correspond to families, by some notion of behavioral similarity. Our technique uses a form of behavioral grouping as a means to remediate a system, but we go further than malware classification by attempting to remove the harmful effects of the malware on the system.

Execution of Untrusted Applications: In addition to work that attempts to detect or prevent the execution of malicious software, some work has been done to mitigate the harmful effects of software *a posteriori*. Hsu et al. presented a framework for automatically repairing an infected system after monitoring the execution of the malware [7]. The actual work of remediating a system given a detailed description of the malicious execution is similar to the way that we construct remediation procedures from generalized behavior models. Liang et al. described an alternative approach called Alcatraz [10]. In Alcatraz, an untrusted application is executed inside of a sandbox, and any change it makes is not committed until the program is confirmed to be innocuous. The manner in which a program is deemed innocuous is considered orthogonal to the main issue of sandboxing. The idea was later tweaked [19] so that all state changes made

[1]The URL for this tool is `http://www.cs.wisc.edu/~mfredrik/remediate`

by an application are *cached*, and upon program termination the user decides whether or not to keep any changes. The primary differences between these techniques and the one presented in this paper is that they rely on information regarding specific execution traces, whereas our remediation procedures use generalized notions of the behavior of a malware instance. As such, our system can remediate harmful effects of malware, including some effects that were not observed in a trace.

Automatic Signature Generation: The generalized behavior models that we use to construct executable remediation procedures can be viewed as generic signatures relating the effects of a malicious program on system resources. Different approaches have been proposed for automatically generating attack signatures. Polygraph [14] is one of the first systems proposed by researchers to address the problem of generating network signatures to detect polymorphic worms. Polygraph identifies invariant fragments of packets that are found in all the network flows generated by the same worm, since they are necessary for the worm to successfully exploit a given vulnerability. These fragments are then combined into signatures using different techniques. Hamsa [9] addresses the same problem using a different algorithm that identifies and combines invariants. Hamsa's signatures have better accuracy and are more resilient against attacks than Polygraph. Finally, Nemean [20] generates semantics-aware signatures to detect network intrusions. Nemean's methodology, consisting of high-level network traffic abstraction, clustering, and generalization using automata learning, is similar to ours. However, we operate on a fundamentally different domain than Nemean, which generates signatures of network packet traces.

3 Overview

In this section, we motivate our work using a realistic example of a malware infection and present our architecture by walking through the steps that it takes to remediate the example.

3.1 Motivation

Consider the malware whose pseudo-code is shown in Figure 1. This program generates a random filename located in the system directory, drops a malicious payload into the file, creates a new registry value that causes the payload to be executed at system boot time, tampers with the system's network name resolver (`c:\...\etc\hosts`), and infects a benign system library (`c:\windows\user32.dll`). Our goal is to generate a procedure that remediates infections caused by any possible execution of this code. In this case, recovery includes: (1) deleting the file containing a copy of the malicious payload, (2) deleting the registry key created to start the malware at boot, (3) disinfecting `c:\windows\user32.dll`, and (4) restoring the original configuration of the name resolver `c:\...\etc\hosts`. It is important that the effects of *all* malicious actions taken by the malware are removed. For example, consider what happens when (1), (2), and (3) are remediated, but not (4). In this case, all internet traffic on the host remains subject to hijacking by the malware, so the system is still in a dangerous configuration. Many commercial products would leave the system in this configuration [15].

Completely remediating the effects of the malware in Figure 1 is not as straightforward as the example might suggest. First, high-level source code is usually not available when dealing with real malware. Given the well-known difficulty of statically analysing adversarial binary code [13], this means we must partially rely on dynamic information. Although this example does not illustrate it, there is a possibility that the malware contains paths that are rarely executed under normal circumstances. Any harmful effects produced on such a path would be difficult to account for in a remediation procedure, because the problem of discovering such an effect dynamically is extremely difficult. Secondly, malware can appear to be nondetermistic by relying on subtle details in its environment, such as the system clock or pseudorandom number generator. This behavior is often present even on common paths, and is apparent in our example, despite its simplicity: Both the filename of the malicious payload and the name of the registry value used to activate the payload depend on randomness.

Given the limited nature of dynamic program information, it may be hard to generate a remediation procedure that precisely accounts for all of the nondeterminism in a program. Procedures that do so may mistakenly identify benign system resources as malicious and attempt to remediate them. Consider a remediation procedure that attempts to account for the nondeterminism in our example by looking for all files in the system directory with the suffix `.exe`. While this policy would effectively capture the nondeterminism in the payload filename, any attempt to remediate resources based on it would result in the unacceptable removal of benign executables. Conversely, procedures that do not attempt to generalize execution behavior are likely to miss some malicious effects that must be remediated. For example, after running the sample malware once, we might find that the payload is delivered in `c:\windows\poqwz.exe`. If a remediation procedure does not generalize this information and only ever looks for this file when remediating infections caused by other executions of this malware, then it will miss the payload file most of the time, as it is not

```
// generate random file and value names
filename = "po" + random_alpha() + random_alpha() + random_alpha() + ".exe";
valuename = (random_int() % 2) ? "qv" : "vq";
...
// drop malicious code
f = CreateFile(”c:\windows\” + filename, GENERIC_WRITE, ...);
WriteFile(f, malicious_buf, ...);
WriteFile(f, other_malicious_buf, ...);
...
// start the newly created executable at boot
RegOpenKey(HKEY_LOCAL_MACHINE, "...\Windows\CurrentVersion\Run", &r);
if (RegQueryValueEx(r, valuename, NULL, REG_SZ, ...) == ERROR_FILE_NOT_FOUND)
  RegSetKeyValue(r, valuename, REG_SZ, filename, ...);
...
// infect user32.dll
g = CreateFile("c:\windows\user32.dll", FILE_APPEND_DATA, ..., OPEN_EXISTING, ...);
WriteFile(g, malicious_buf, ...);
...
// hijack HTTP connections to www.google.com and www.citibank.com
h = CreateFile("c:\windows\system32\drivers\etc\hosts", ..., OPEN_EXISTING, ...);
ReadFile(h, buf, ...);
WriteFile(f, "67.42.10.3 www.google.com\n67.42.10.3 www.citibank.com", ...);
...
// delete main executable
DeleteFile("c:\malware.exe");
```

Figure 1: Pseudo-code of a sample malicious program.

possible to observe the malware long enough to see all possible variants of the payload file name.

3.2 Architecture Overview

The architecture we have developed for generating remediation procedures from malware binaries is shown in Figure 2. It has three primary components: (1) an execution monitor that infers the malware's high-level behaviors from a low-level trace, (2) a component that *generalizes* the high-level behaviors from multiple executions of the malware, and (3) a component that produces executable remediation procedures from generalized behaviors. The entire system works sequentially, with each component using the information produced by the one preceding it.

High-Level Behavior Extraction: The high-level behavior extraction component (numbered 1 in Figure 2) analyzes the semantics of a program to produce a sequence of meaningful behaviors relevant to remediation. Because malware authors usually obfuscate their binaries, we rely on dynamic information to infer these behaviors; we execute binaries in a special environment (an emulator) to extract a low-level execution trace, perform analysis using manually constructed rules, and arrive at a high-level trace [11]. Table 1 lists the high-level behaviors we consider. Each behavior modifies the state of the system in some way and is parameterized by a set of arguments that determine which aspects of the system state are affected. The behaviors currently listed correspond to those that commonly occur in malware, that are mandatory to infect a system, and were constructed manually to reflect the salient behavioral features of most malware. However, our technique can be extended to operate over a wider set of high-level behaviors.

The environment in which a program runs typically affects its behavior, and malware often exhibits a certain degree of nondeterminism. To account for these factors, we collect several high-level behavior traces for each sample. To do so, we vary the environment by changing factors that malware typically rely on, such as locale, service pack level, and so forth. Although not supported by our current implementation, path exploration techniques [12] can be applied in this component to account for a more complete subset of the malware's behavior, as in Bouncer [5]. The lack of path exploration techniques is not a fundamental limitation of our system, and can be easily *plugged into* our system.

Our high-level behavior extractor would infer that the sample malware from Figure 1 demonstrates the `FileCreation`, `RegistryCreation`, `DropAndAutostart`, and `FileInfection` behaviors, with different arguments for `FileCreation`, `RegistryCreation`, and `DropAndAutostart` on each execution.

Behavior Generalization: After producing a set of high-level behavior traces for a malware sample, we attempt to account for nondeterminism by creating a general, abstract model of behavior that accounts for all of the concrete traces we observed (numbered 2 in Figure 2). Note that generalization attempts to *overapproximate* existing paths, thus encompassing future paths, rather than explore as many new paths as possible. In effect, this *patches* some of the incompleteness of dynamic

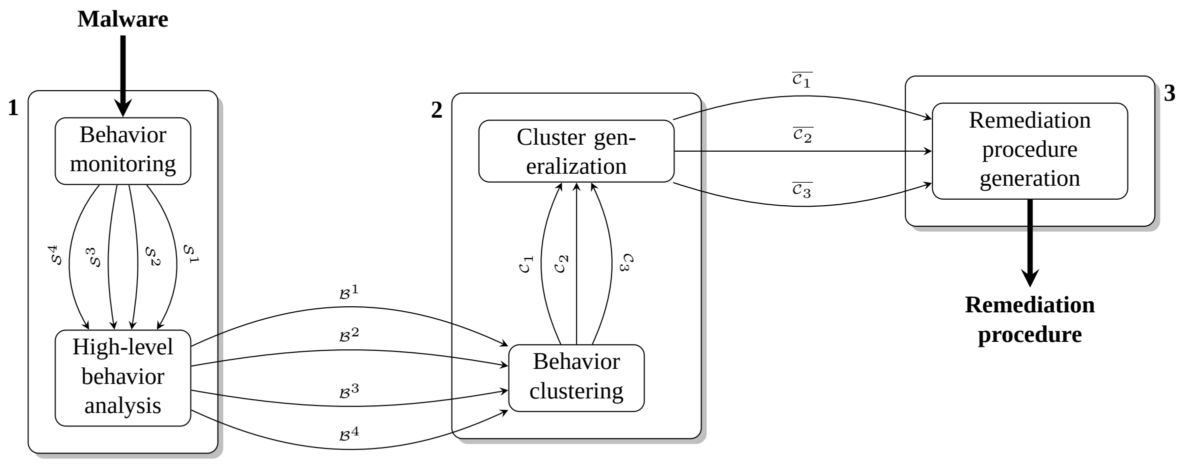


Figure 2: Architecture of the system for generating remediation procedures. In this figure, $\mathcal{S}$ denotes a system call trace, $\mathcal{B}$ denotes a high-level behavior trace inferred from a system call trace, $\mathcal{C}$ denotes a cluster, and $\bar{\mathcal{C}}$ denotes a *generalized* cluster.

analysis by extrapolating observed information to future, unseen executions of the malware. This is accomplished by recognizing when distinct behaviors from multiple high-level traces, with possibly different arguments, are actually instances of the same malicious activity. We refer to this matching of behaviors as *clustering*. When a cluster is identified, the arguments of its constituent behaviors are generalized to tolerate any differences that may be present in the actual values. Thus, nondeterminism is accounted for via overapproximation by ensuring that this generalization extends to future, unseen executions.

In the malware from Figure 1, our technique would cluster all instances of the same high-level behavior together. For example, all instances of `DropAndAutostart` would be clustered together and all instances of `FileInfection` would be clustered together. Because there is likely variation among the arguments of `DropAndAutostart`, we construct a regular expression to tolerate minor differences while ensuring that benign files are not mistakenly identified. The final result of the computation for this behavior would be a `DropAndAutoStart` behavior with generic file argument `c:\windows\po[[:alpha:]]{3}.exe` to generalize the random filename at line 2, generic registry key/value pair `...\CurrentVersion\Run` for the registry touched at line 11, and `(qv|vq)` for the registry value randomly created at line 3.

Remediation Procedure Generation: The third component of our architecture (numbered 3 in Figure 2) generates executable remediation procedures from the generalized behaviors produced in the previous step. The resulting procedure examines the state of the system on which it runs in search of symptoms of an infection, and removes the symptoms whenever possible. It attempts to match each resource (file, process, or registry key) on the system against the constraints associated with each generalized high-level behavior. For our running example, each file is matched against the regular expression `c:\windows\po[[:alpha:]]{3}.exe` associated with the first argument of the `DropAndAutoStart` behavior, another regular expression associated with the second argument, and a final one describing the content of the file. If such a file is found, then the registry values under the key `...\CurrentVersion\Run` are matched against the regular expression `(qv|vq)`. If such a value is found and its data matches the current filename being considered, then all of the resources (the file and registry key pair) are removed. Currently, we only produce remediation procedures that operate on system files. For technical reasons explained in Section 4, we do not handle user-specific files and resources. While this is a limitation of our current approach, we hope to remove it in future work.

4 Generating Remediation Procedures

In this section, we present the details of our system for generating remediation procedures. We begin by formalizing the problem solved by our system and continue component-by-component describing the algorithms used to solve the problem.

4.1 Problem Description

When malware runs on a system, it may infect the system by changing its persistent state in an undesirable way.

	Behavior	Arguments	Description
Resource creation	`FileCreation`	File name and content	Creation of a new file
	`RegistryCreation`	Key name and content	Creation of a new registry value
	`DropAndAutostart`	File name and content. Key name and content	Creation of a new file and of a registry value containing its name (to execute the file automatically at every boot)
	`DropAndExecute`	File and process name	Creation and execution of a new executable
Resource infection	`FileInfection`	File name and content. List of preserved regions	Infection of an existing file
	`RegistryInfection`	Key name and content	Replacement of an existing registry value
Resource deletion	`FileDeletion`	File name	Deletion of an existing file
	`RegistryDeletion`	Key name	Deletion of an existing registry value

Table 1: High-level behaviors considered for remediation.

For our purposes, the state S of a system is modeled as an association from resource names N to data from a domain D. Individual elements of S are referred to as *resources*. To simplify notation, we let $\mathcal{S}$ stand for the set of possible system states. Because most malware is written for Windows platforms, our targeted resource namespace consists of Windows filenames, registry key and value names, and process names. The data domain is the set of all finite-length bit strings.

The infection behavior of a malware can be understood as a transition relation between system states. There are three ways in which the malware can modify the state of a system: *(1)* resources may be completely removed from the system, *(2)* new resources may be added to the system, and *(3)* the data corresponding to existing resources may be mutated. Because the infection behavior of a malware can be succinctly described in terms of these three operations and the resources over which they operate, we represent it using an *infection relation* $R \subseteq \mathcal{S} \times N \times \mathcal{S} \times \mathcal{S}$ that encodes this information. Intuitively, the infection relation describes the way in which a particular malware changes the state of a system. Given an element $(S, N_{rem}, S_{add}, S_{mut}) \in R$, the malware transforms state S into a new state by removing the resources labeled by N_{rem}, adding the resources in S_{add}, and modifying the resources in S_{mut}. Note that the infection behavior is described as a relation rather than a function mapping. This is because of the fact that malware may behave nondeterministically when it infects a system—it may infect the same system state in different ways on two distinct executions.

After a given piece of malware has infected a system, the goal of remediation is to undo the effects of the infection, returning the system to a clean state. More precisely, given a malware binary, we seek to construct an infection relation for that malware that describes its behavior. We can then use the information in the infection relation to enact changes on the system that remediate the effects of the malware: restoring any files that were removed (N_{rem}) or mutated (S_{mut}), and removing files that were added (S_{add}). We package this functionality as an executable remediation procedure, as described in Section 3.2. In general, there are a number of approaches that may realize the goal of constructing the infection relation corresponding to a given malware. In this paper, we focus on applying dynamic analysis to the malware sample to extract the information necessary to construct the infection relation.

In practice, it is not usually possible to reconstruct the true infection relation from a malware binary. Rather, we compute a relation that *overapproximates* the actual behavior for a finite set of execution paths exhibited by the malware. For example, we overapproximate the resource names involved in the `DropAndAutoStart` behavior of Figure 1 by creating a regular expression that matches all of the resource names on the set of execution traces we observed. Furthermore, our approximate infection relations do not contain information regarding the removal or mutation of non-system files, as it is generally not possible to restore this state without additional information not encoded in the malware. Of course, using an approximate infection relation for remediation introduces the possibility of *false negatives* and *false positives*. A false negative occurs when the remediation fails to properly reverse the changes left by the malware. Similarly, a false positive occurs when remediation affects resources that were not touched by the malware. Both types of error are possible given the way we construct approximate infection relations. For example, false positives may result from the overapproximation of resource names with regular expressions, whereas false negatives may result from the fact that we do not account for all possible execution paths in the malware. Thus, it is our goal to construct an approximate infection relation that minimizes false positives and false negatives

4.2 System Details

This section details the specific algorithms and subsystems used in the three main components of our system (depicted in Figure 2).

4.2.1 High-Level Behavior Extraction

Intuitively, the problem of high-level behavior extraction is to derive a concise description of the behavior semantics demonstrated by a malware sample. Given a malware sample m and a set D of *high-level behavior templates* that describe events related to system state modification, the goal of this task is to produce a sequence of instances of the members of D, along with a corresponding low-level description of system events that match each template instance.

The set of behavior templates used in our prototype is given in Table 1. To infer high-level behaviors from a stream of system calls, we use *multilayer behavior specifications*, as proposed in previous work [11]. Although the details of the inference algorithm are beyond the scope of this paper, we give a brief account of the main points here. Each high-level behavior is described in terms of a hierarchical model. Each level of the hierarchy is composed of a set of *behavior summaries* and their accompanying *behavior graphs*. The graph for a given behavior summary encodes the behavior operationally, in terms of events and the dependencies among them. The events in a graph at a particular level are defined in terms of the summaries of levels lower in the hierarchy. The top level of the hierarchy corresponds to the final output of the inference, and the layers beneath it provide details of incremental specificity, until the lowest level is reached. In our prototype, the lowest level corresponds to a system call trace collected in a virtual environment. We use a modified version of QEMU [3] to monitor an application for its system call trace.

The nodes in the behavior graphs at each layer correspond to events that are observed by the monitor, and the edges correspond to data dependencies between the events. For example, in the graphs at the lowest level, system calls that operate on the same resource handle have edges between their representative nodes that reflect this dependency. At the highest level, this relationship is preserved by edges that denote the fact that the corresponding set of high-level behaviors operate on the same file. Representing high-level behavior graphs hierarchically has one crucial advantage: the same high-level behavior can be described in terms of multiple alternative intermediate behaviors. For example, our high-level behavior `DropAndAutostart` can be represented in terms of all possible low-level system call sequences that create a new file, write executable content into it, and then change the system configuration to activate the dropped file at boot time. Because there are numerous distinct ways to accomplish this high-level task in terms of system calls, it is important to account for all of them in a clean and straightforward way. Our heirarchical behavior model formalism allows this, and thus makes our system more resilient to this type of evasion.

Figure 3 shows a sample system call trace and two of the high-level behaviors extracted from it. The figure shows both the concrete graphs and the template instances that were matched. The first four system calls in the trace (members s_1 through s_4) are executed by the malware sample to replicate its payload into a new file. These calls are associated with the layer-1 behavior `FileCreation`. Similarly, the system calls s_{11}, s_{13}, and s_{14} are associated with the layer-1 behavior `RegistryCreation`, and the last system call (s_{41}) with the behavior `FileDeletion`. Since the behaviors `FileCreation` and `RegistryCreation` are related, the algorithm infers the high-level layer-2 behavior `DropAndAutostart`, which represents the fact that the malware replicates and configures the system to execute the malicious payload at boot. Note that this high-level behavior was inferred hierarchically; the fact that `DropAndAutoStart` is present in the trace was inferred only from layer-1 behaviors, which were in turn inferred from system calls originally found in the trace. By modularizing the template definitions in this way, our high-level behavior inference technique gains a certain amount of resilience to obfuscations and differences in malware implementation [11].

4.2.2 Behavior Clustering

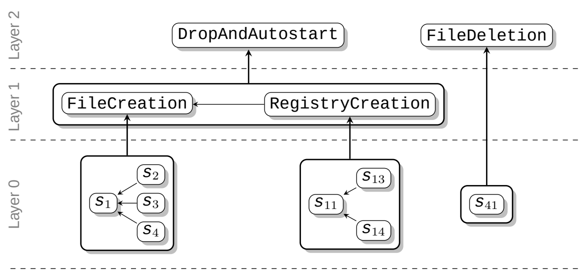
Given a set of high-level behavior traces $\{\mathcal{B}^1, \ldots, \mathcal{B}^m\}$ corresponding to multiple executions of the same malware sample, *behavior clustering* identifies elements of distinct traces that correspond to the same malicious activity. An *admissible clustering* for a given set of traces is a set of behavior sets $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ that satisfies two conditions:

1. All behaviors in a given cluster $\mathcal{C}_i$ have the same *type*. For example, all behaviors are of type `DropAndAutostart`.

2. The clustering *partitions* the set of all events in every execution trace: no behavior is in more than one cluster, and each behavior is in some cluster.

```
s1   NtCreateFile("poqwz.exe") → f
s2   NtWriteFile(f, "...malicious code...")
s3   NtWriteFile(f, "...other malicious code...")
s4   NtClose(f)
...
s11  NtOpenKey("Run")  →  r
s12  NtQueryValueKey(r, "vq") → FAILURE
s13  NtSetValueKey(r, "vq", "poqwz.exe")
s14  NtClose(r)
...
s21  NtOpenFile("...\system32\user32.dll")  →  g
s22  NtWriteFile(g, "...malicious data...")
s23  NtClose(g)
...
s31  NtOpenFile("c:\windows\hosts")  →  h
s32  NtReadFile(h, 1024) → "# Copyright (c)..."
s33  NtWriteFile(h, "67.42.10.3  www.google.com...")
s34  NtWriteFile(h, "67.42.10.3  www.citibank.com...")
s35  NtClose(h)
...
s41  NtDeleteFile("c:\malware.exe")
```

(a)



High-Level Behavior Summaries

```
DropAndAutostart("poqwz.exe", data, "Run", "vq",
                 "poqwz.exe")
FileCreation("poqwz.exe", data)
FileDeletion("c:\malware.exe")
RegistryCreation("Run", "vq", "poqwz.exe")
```

(b)

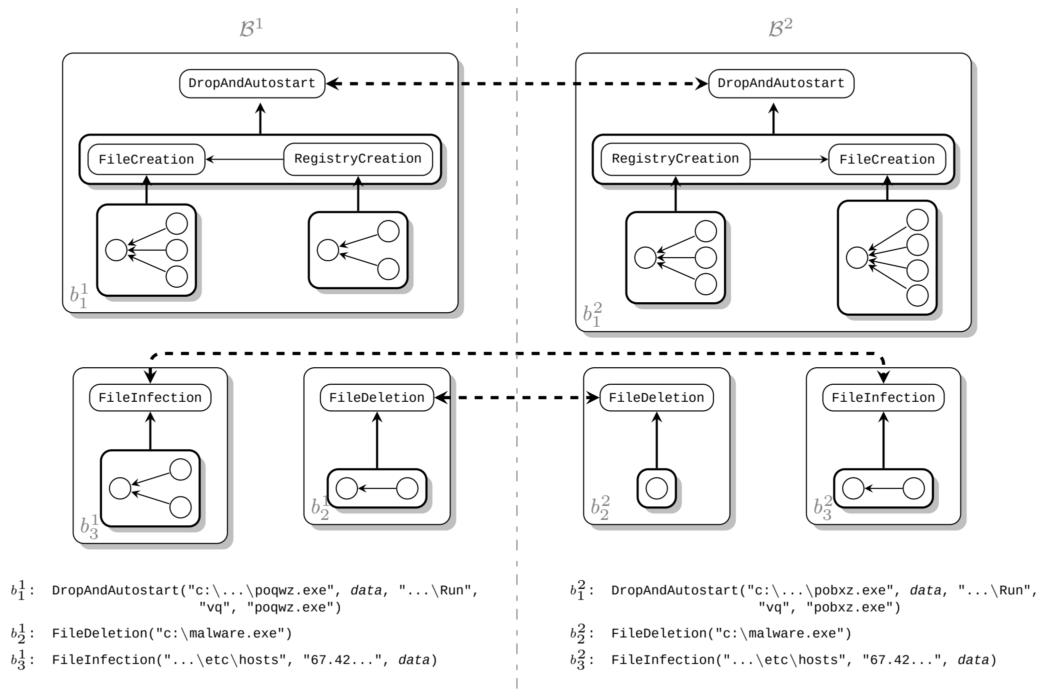
Figure 3: The system call trace for our sample `malware.exe` (a) and high-level behaviors generated from the trace (b).

In later stages of the system, it generalizes behaviors in the same cluster by overapproximating their argument values. Thus, desirable clusterings are those that lead to tighter overapproximations, while still grouping related behaviors together in order to allow generalization. As an example, Figure 4 shows two high-level traces of our sample malicious program. We denote the j^{th} behavior observed in the i^{th} execution trace as b^i_j. For these traces, we want to group behaviors b^1_1 and b^2_1 because they correspond to the same activity, and generalizing their arguments leads to a tight overapproximation: we can use regular expressions that match a fairly small set of strings (namely, `po[[: alpha :]]{3}.exe`). Similarly, we want to group b^1_2 with b^2_2 and b^1_3 with b^2_3. However, had the second trace contained another `DropAndAutostart` behavior for an executable named `avkiller.exe`, then clustering b^1_1 with this behavior would have resulted in a poor generalization. An optimal clustering is one that includes all related high-level behaviors so that generalization will create a powerful regular expression that finds all traces of a malicious behavior. On the other hand, an optimal clustering must not include unrelated high-level behaviors, as a generalization of such a cluster is likely to match benign system resources.

Cluster Formation: Exhaustively searching for the optimal clustering of $\{\mathcal{B}^1, \ldots, \mathcal{B}^m\}$ is infeasible, as there are an exponential number of possibilities. Thus, we do not attempt to find an optimal clustering and instead rely on the heuristic method shown in Algorithm 1. The algorithm begins by finding the execution trace with the greatest number of high-level behaviors $\mathcal{B}^{max}$, and creating an initial clustering by placing each b^{max}_i in its own cluster $\mathcal{C}_i$. Then, for each remaining behavior trace B^j, the events are enumerated in execution order and added to the first cluster that satisfies the admissibility criterion discussed above. We discuss the details of matching event types below. If an event cannot be added to any existing cluster, then a new cluster is initialized with the current event. This process is repeated until no traces remain, at which point the current set of clusters is returned as the final result.

Intuitively, the heuristics in this algorithm rely on two asssumptions: *(1)* distinct executions of the malware exhibit similar malicious behaviors, and *(2)* the ordering of malicious behaviors between executions is similar. By selecting the trace with the greatest number of events to seed the clustering process and assuming that different executions contain a similar set of behaviors, we seek clusterings that group as many behaviors together as possible. By adding events to existing clusters in execution order and assuming that the order does not vary substantially between executions, we seek clusterings that match similar argument values, thus resulting in tighter overapproximations in the behavior generalization phase of the system. Furthermore, these heuristics allow our algorithm to operate efficiently: Algorithm 1 runs in time linear in the number of execution traces and the length of the traces.

For an example of how Algorithm 1 works, consider the two high-level execution traces depicted in Figure 4. As both traces are of equal length, the first is chosen, in this case $\mathcal{B}^1$. Clusters $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are initialized with behaviors b^1_1, b^1_2, and b^1_3, respectively. b^1_1 and b^2_1 can then be matched, as they are both instances of the `DropAndAutostart` high-level behavior. Similarly, b^1_2 is matched to b^2_2, and b^1_3 is matched to b^2_3. Finally, the algorithm returns clusters $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ where



b_1^1: `DropAndAutostart("c:\...\poqwz.exe",` *data*`, "...\Run", "vq", "poqwz.exe")`

b_2^1: `FileDeletion("c:\malware.exe")`

b_3^1: `FileInfection("...\etc\hosts", "67.42...",` *data*`)`

b_1^2: `DropAndAutostart("c:\...\pobxz.exe",` *data*`, "...\Run", "vq", "pobxz.exe")`

b_2^2: `FileDeletion("c:\malware.exe")`

b_3^2: `FileInfection("...\etc\hosts", "67.42...",` *data*`)`

Figure 4: High-level behavior clustering.

$\mathcal{C}_1 = \{b_1^1, b_1^2\}$ represents `DropAndAutostart` behaviors, $\mathcal{C}_2 = \{b_2^1, b_2^2\}$ represents `FileDeletion` behaviors, and $\mathcal{C}_3 = \{b_3^1, b_3^2\}$ represents `FileInfection` behaviors.

Behavior Comparison: Our clustering algorithm requires a sub-algorithm, *isomorphic*, to compare two behaviors. Intuitively, we perform this comparison by *normalizing* the graphs corresponding to each behavior and then checking whether the resulting normalized graphs are isomorphic. There is an important advantage in comparing the behavior graphs rather than their high-level summaries: nondeterminism in a malicious program typically affects the summary of the behavior, but not the low-level operations used to achieve the behavior. Therefore, this approach is more resilient to nondeterminism and performs a more thorough comparison, eventually yielding more precise results.

The normalization we perform on each graph mainly consists of abstracting away details of the behavior that are likely affected by nondeterminism. System call arguments that represent resource names are replaced by constants that denote their type. For example, we use a different constant for each file and registry type. Sequences of system calls that operate sequentially on the same resource are replaced with a single, *batch* call that is semantically identical. Finally, we ignore system calls whose effects are later *killed*, i.e. overwritten or otherwise reversed. In this way, our normalization step produces more succinct graph representations of the malware's behavior that are largely independent of common forms of nondeterminism.

After normalizing two graphs for comparison, we use the VFlib2 graph isomorphism algorithm [6]. Although isomorphism is a difficult problem and may be inefficient to compute on large graphs, we point out that the normalized behavior graphs resulting from real-world programs are typically quite small, comprising no more than a few dozen nodes.

4.2.3 Behavior Generalization

After clustering, we have several sets of behaviors grouped by semantic similarity but still differing in certain details. For example, when we build clusters we group together behaviors that differ in the specific resources they identify. The goal of behavior generalization is to produce a single canonical behavior that represents all of the members of a given cluster, as well as variations of the members that are likely to result from other executions of the malware. In terms of the definitions presented in Section 4.1, behavior generalization produces high-level behaviors with arguments constructed to accurately represent the resources modified by observed executions, while generalizing to potential future executions.

Algorithm 2 presents $\mathit{Generalize}$, our procedure for generalizing a behavior cluster. Intuitively, generaliza-

Algorithm 1 $Cluster(\mathcal{B}, \mathcal{B}^{\max})$

```
Require: $\mathcal{B}$ is a set of high-level behavior traces $\{\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^m\}$
    $\mathcal{B}^{\max}$ is the high-level behavior trace containing the maximum number of high-level behaviors
Result: A set of clusters of high-level behaviors of $\{\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^m\}$
  $\mathcal{C} \leftarrow \varnothing$
  for $b_j^{\max} \in \mathcal{B}^{\max}$ do
    add new cluster $\{b_j^i\}$ to $\mathcal{C}$
  end for
  for all $\mathcal{B}^i \in \mathcal{B}/\mathcal{B}^{\max}$ do
    {Traces are enumerated in the order of collection.}
    for all $b_j^i \in \mathcal{B}^i$ do
      {Behaviors are enumerated in execution order.}
      for all $\mathcal{C}_k \in \mathcal{C}$ do
        if $isomorphic(b_j^i, b_k)$ where $b_k$ is a behavior in $\mathcal{C}_k$ then
          $\mathcal{C}_k \leftarrow \mathcal{C}_k \cup \{b_j^i\}$
        end if
      end for
      if $b_j^i$ is not in any cluster then
        add new cluster $\{b_j^i\}$ to $\mathcal{C}$
      end if
    end for
  end for
  return ($\mathcal{C}$)
```

Algorithm 2 $Generalize(C, \mathcal{G}, \delta)$

```
Require: $C$ is a cluster of behaviors that differ only in argument values, $\mathcal{G}$ is a set of generalization rules, $\delta$ is the density threshold.
Result: A generalized high-level behavior.
  {Loop through all arguments for behaviors in cluster $C$}
  for $i = 0$ to $|args(C_0)|$ do
    $A_i \leftarrow \varnothing$
    {Gather all values for current argument}
    for $c$ in $C$ do
      $A_i \leftarrow A_i \cup args_i(c)$
    end for
    {Generate PFSA that captures argument values}
    $(V, E) \leftarrow \textsc{Pfsa}(A_i)$
    {Find dense regions in the PFSA}
    for $(n_1, n_2)$ in $V \times V - \{(n, n) \mid n \in V\}$ do
      if $\neg idom(e_1, e_2)$ or $\neg ipdom(n_2, n_1)$ or $numpaths(n_1, n_2) < \delta$ then
        continue
      end if
      for $r$ in $\mathcal{G}$ do
        $E' \leftarrow r(paths(n_1, n_2))$
        $E \leftarrow (E - paths(n_1, n_2)) \cup E'$
      end for
    end for
    {Build regular expression for the current arguments}
    $G_i \leftarrow regexp(E)$
  end for
  {Return new behavior with type matching $C$, and generalized reg. exp. arguments}
  return $name(C_0)(G_0, \ldots, G)$, $0 \leq n < |args(C_0)|$
```

tion is performed on each high-level behavior argument individually, and the individual results are eventually combined to produce the generalized behavior. Because each cluster member represents the same high-level behavior, and therefore has the same number of arguments as the others, we are assured that all of the relevant information is included in the generalization. Furthermore, because all arguments for the behaviors that we are interested in have straightforward canonical representations as strings, the problem of generalizing each argument can be reduced to the problem of generalizing sets of strings. $Generalize$ proceeds in this vein, iterating over each argument for the behaviors in a given cluster C. After collecting each string for a given argument in a set A_i, a probabilistic finite-state automaton (PSFA) that accepts all of the strings in A_i is constructed using the *simulated beam annealing* algorithm [17]. By merging states that are probabilistically very similar, the resulting automaton accepts a superset of A_i, thus resulting an initial generalization.

After building the PFSA, certain regions of the state transition diagram are examined for reduction using a set $\mathcal{G}$ of *generalization rules*, which are templates for generating regular expressions that overapproximate high-level behavior arguments. We refer to a *single-entry single-exit region* as one whose entry is composed of a node n_1 that is the immediate dominator of the exit node n_2, which is the immediate postdominator of n_1. Furthermore, we require that the number of paths between n_1 and n_2 be at least δ. The actual value of δ is estimated empirically. This information is represented in Algorithm 2 with the relations $idom_E$ and $ipdom_E$, as well as the function $numpaths_E$. When a suitable single-entry single-exit region is found, each rule in $\mathcal{G}$ is applied in an attempt to generalize it. The generalization rules that we use have been chosen on the basis of experience and consider information such as the number of paths in the region, the probabilities associated with the paths, the lengths of the paths, and the characters composing the strings associated with each path. If a rule is able to generalize the region, then it returns a smaller set of edges that are used to replace the original region. Otherwise, the rule returns the original region, and the next rule is applied. After all rules in $\mathcal{G}$ have been applied, a regular expression is built from the resulting PFSA, which is eventually used as an argument in the final generalized behavior. The final behavior is represented in Algorithm 2 by $name(C_0)(G_0, \ldots, G_n)$. Here, $name(C_0)$ returns the behavior name of the high-level behavior C_0, which is used to build the final generalized behavior from

```
DropAndAutostart("c:\windows\...poagp.exe",data,"...Windows\CurrentVersion\Run","vq","poagp.exe")
DropAndAutostart("c:\windows\...pobxz.exe",data,"...Windows\CurrentVersion\Run","vq","pobxz.exe")
DropAndAutostart("c:\windows\...pocra.exe",data,"...Windows\CurrentVersion\Run","qv","pocra.exe")
DropAndAutostart("c:\windows\...pomfq.exe",data,"...Windows\CurrentVersion\Run","vq","pomfq.exe")
DropAndAutostart("c:\windows\...pommp.exe",data,"...Windows\CurrentVersion\Run","qv","pommp.exe")
DropAndAutostart("c:\windows\...popwz.exe",data,"...Windows\CurrentVersion\Run","qv","popwz.exe")
DropAndAutostart("c:\windows\...pouwk.exe",data,"...Windows\CurrentVersion\Run","vq","pouwk.exe")
```

Figure 5: Sample cluster grouping seven different occurrences of the `DropAndAutostart` behavior manifested by our sample malware (the corresponding graphs are omitted for conciseness).

the individual argument generalizations.

As an illustration of this algorithm, consider the cluster presented in Figure 5. We apply the PFSA algorithm to the first argument to arrive at the minimal automaton shown in Figure 6. The automaton contains a single-entry single-exit region with several paths, as highlighted in the figure, that encodes the variable substring of the filename. One of the generalization rules that we use is triggered by the fact that this region is *dense*, i.e. it contains many paths from entry to exit, as well as the fact that it contains only alphabetic characters. Thus, it returns a single edge labeled $[[:\mathtt{alpha}:]]\{3\}$, which is a wildcard sequence that denotes all alphabetic strings of length three. The generalized PFSA results in the regular expression $\mathtt{c}:\backslash\mathtt{windows}\backslash\mathtt{po}[[:\mathtt{alpha}:]]\{3\}.\mathtt{exe}$, which is capable of identifying all the names of the files that our sample malicious program could touch on the system. After applying *Generalize* to all arguments of `DropAndAutostart`, we obtain a generic model of the cluster behavior represented by $\mathtt{DropAndAutostart}(\text{“}\mathtt{c}:\backslash\mathtt{windows}\backslash\mathtt{po}[[:\mathtt{alpha}:]]\{3\}.\mathtt{exe}\text{”}, \mathit{data}, \text{“}...\mathtt{Windows}\backslash\mathtt{CurrentVersion}\backslash\mathtt{Run}\text{”}, \text{“}(\mathtt{vq}|\mathtt{qv})\text{”})$.

4.2.4 Generating Concrete Remediation Procedures

Each generalized high-level behavior must be remediated differently. Our approach to generating executable remediation procedures may be understood conceptually in two parts. First, the generalized high-level behaviors for each cluster are used to construct an approximate infection relation R as discussed in Section 4.1. Then, we use a generic procedure that scans the infection relation, and changes the state of the system based on the contents of each entry. When constructing the infection relation, our procedure uses a model of a clean, bare installation of the operating system installed on the machine for the first system state component of each tuple. The use of a bare installation enables us to remediate infected system resources up to the correct service pack installed on the system, but not personal or application-specific resources.

The remainder of this section details the way that specific high-level behaviors are translated into entries in the abstract infection relation, as well as the way that the abstract infection relation is used to generate a concrete (executable) remediation procedure.

Algorithm 3 $Remediate(S, R)$

```
$(S_{abs}, N_{rem}, S_{add}, S_{mut}) \leftarrow (S_{abs}, N_{rem}, S_{add}, S_{mut}) \in R$ such that $S$ has the same operating system version as $S_{abs}$
for $s$ in $S_{add}$ do
  cases $s$:
    $(name, data)$ : if file $name$ exists, with contents
                     matching $data$ then remove $name$.
    $((key, value), data)$ : if $(key, value)$ exists with
                             contents matching $data$
                             then remove $(key, value)$.
    $((file, key, value), (data, regdata))$ :
          if $file$ exists and is a suffix of some element of
          $D_{regdata}$ that also exists in a key matching
          $(key, value)$ then remove $file$ and
          $(key, value)$.
    $((file, procname), data)$ :
          if $procname$ and $(file, data)$ exist matching
          $file$, $data$, $procname$ and $procname$ is a
          suffix of $file$ then remove $file$ and
          kill $procname$.
  end cases
end for
for $i$ in $I_{mut}$ do
  cases $i$:
    $(file, data)$ : Remove $(file, data)$ and replace it
                     with $(file, data') \in \beta(S)$
    $((key, value), data)$ : Remove $((key, value), data)$
                             and replace it with
                             $((key, value), data) \in \beta(S)$.
  end cases
end for
```

Newly-Created Resources: Remediating resources that are created by malware is straightforward, because the remediation procedure only needs information regarding the names and data of newly-created resources to completely remove the corresponding resources from the system. Our remediation procedures are capable of removing files and registry keys. To account for the possibility that the infection could create resources that were not observed in a high-level behavior trace during analysis, we instead use generalized high-level behaviors in the infection relation R.

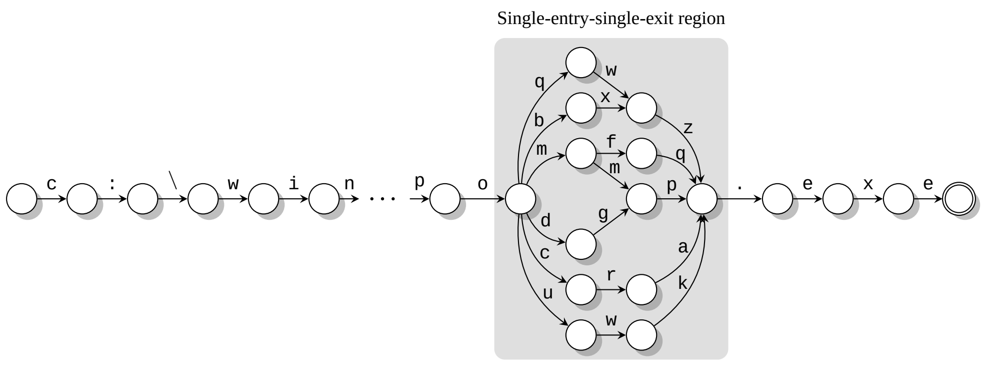


Figure 6: A fragment of the minimized automaton constructed to generalize the first argument of the `DropAndAutostart` behavior, starting from the occurrences of the argument reported in Figure 5.

For the high-level file creation behavior $\texttt{FileCreation}(name, data)$, we find the resource for *name* and *data* and append this pair to S_{add}. Similarly, for the high-level registry creation behavior $\texttt{RegistryCreation}\ (key, value, data)$, we associate the key/value pair to the corresponding data and add them as a pair to S_{add}. As shown in Algorithm 3, the remediation procedure processes these entries in the infection relation R by checking for the existence of the resource names on the system and removing them if they exist with the contents specified by R_α.

Remediating the `DropAndAutostart` and `DropAndExecute` behaviors is more complicated, as doing so involves multiple resources that are related in a constrained manner. To handle a high-level behavior of the form:

$$\texttt{DropAndAutostart}(\mathit{file}, data, key, value, regdata)$$

we group the resource names: *file, key, value* together as a compound resource name for a new element in S_{add}, and group *data* and *regdata* together for the corresponding data component. The remediation procedure acts on such an entry by scanning system resources for names that match the file name and registry key/value pairs. If a match is found, the corresponding resources are removed only if the concrete filename is a suffix of the concrete registry data and the concrete data matches the abstract data.

For example, when the procedure encounters the generalized `DropAndAutostart` from Figure 5, it will augment S_{add} with the following resource:

$$(\texttt{c}:\backslash\texttt{windows}\backslash\texttt{po}[[:\texttt{alpha}:]]\{3\}\backslash.\texttt{exe},$$
$$(...\backslash\texttt{CurrentVersion}, \texttt{Run}),$$
$$(data, \texttt{po}[[:\texttt{alpha}:]]\{3\}\backslash.\texttt{exe}))$$

The remediation procedure will then search the system for a file that matches $\texttt{c}:\backslash\texttt{windows}\backslash\texttt{po}[[:\texttt{alpha}:]]\{3\}.\texttt{exe}$, as well as the registry key $(...\backslash\texttt{CurrentVersion}, \texttt{Run})$, and will remove the resources only if the value of the registry key matches the name of any file that matches the regular expression.

Infected Resources: Remediating infected resources is more challenging than newly-created resources. In general, it is not possible to know the contents of a file before infection takes place, so it is not possible to restore their contents to a clean state. The exception to this fact is with operating system files, which are common to all systems and can thus be known to the remediation procedure *a priori*.

A naive approach to remediating high-level $\texttt{FileInfection}(name, region, data)$ behaviors would be to replace the entire file with the corresponding file in the bare operating system. However, uninfected regions of data may be removed by this technique, which could result in the loss of important system data, or leave the system in an inconsistent state. To avoid this circumstance, high-level behavior traces keep track of uninfected regions *regions* in addition to file name *file* and infected data *data*. We update the S_{mut} component of R to account for a `FileInfection` behavior only if there is an actual file in the clean operating system state whose name matches the *file*. In this case, S_{mut} is updated with the contents of *file* in the bare operating system state, modified by preserving the portions listed in *regions* and overwriting the rest with *data*. As indicated in Algorithm 3, when the remediation procedure finds *file*, it replaces the infected regions with a pristine copy from the bare operating system.

Similarly, when a high-level $\texttt{RegistryInfection}((key, value), data)$ behavior is encountered, and it is determined that a counterpart of $(key, value)$ exists in the bare operating system, S_{mut} is modified by adding the key/value pair together with the modified data to the list of infected resources. As with infected files, Algorithm 3 remediates these resources by locating a pristine copy of $(key, value)$ in the bare operating system and replacing the infected resource with it.

Deleted Resources: Currently, most malware is written with the intent of leveraging infected systems to perpetrate profitable, albeit illicit, activities. Therefore, it is very rare to see malware removing system resources, as doing so would render the system useless for money-making activities. For this reason, our remediation procedures do not handle deleted resources.

5 Evaluation

We applied our remediation procedure generation algorithm to over two hundred malware samples collected in the wild. We evaluated the quality of the generated procedures with respect to two metrics: false positives and false negatives. A false positive occurs when a resource is mistakenly identified as being part of a malware infection and subsequently remediated. A false negative occurs when a resource that was actually involved in an infection is not identified and left untouched by the remediation procedure. The results of our evaluation testify to the effectiveness of our technique: we observed a low false negative rate, with more than 98% of the malicious resources successfully remediated, and only one false positive was encountered. Finally, we compare our results to the remediation capabilities of the three commercial products that performed best in previous experiments [15].

5.1 Experimental Setup

Our experiments were performed over a corpus of 200 malicious programs, obtained through our own honeypot, and a web crawler that crawls known malicious domains for executable files. Several traces for each sample were collected by executing it in multiple distinct environments. To extract a wide range of behaviors from each sample, we modified the environments along a variety of dimensions, including locale, timezone, and the set of installed applications. Specifically, for each sample we performed the following steps:

1. Execute the sample three times in five different environments, collecting a system call trace for each execution. Apply the algorithm described in Section 4.2 to generate a remediation procedure from the collected data.
2. Infect twenty-five test environments, all of them distinct from those used to collect traces, with the sample.
3. Execute the generated remediation procedure in each test environment.
4. Compare the remediated state to the original (clean) state. Tally the false positives and false negatives.

Although we do not attempt to extract all possible execution paths from the malware, this strategy allows us to observe a reasonable range of malware behavior in various settings.

5.2 False Negatives

Figure 7 compares the false negative rate of our automatically-generated remediation procedures with the three top-rated commercial malware detectors evaluated in [15]: Nod32 Anti-Virus 3.0, Panda Anti-Virus 9.0.5, and Kaspersky Anti-Virus 2009. The graph depicts the average number of malicious resources that were remediated over the entire malware corpus. Resources are divided into three categories: files, registry keys, and processes. Each of these classes is further divided into two subcategories: primary and ancillary. Primary resources are composed of executable files, registry keys that activate process creation, and processes that arise from files dropped or infected by the malware sample. Roughly, we argue that all other resources are not as critical to the security of the system, and are thus considered ancillary.

For the majority of these categories and subcategories, our remediation procedures are more complete than commercial anti-malware products. For example, our procedures were able to remediate more than 99% of the primary file resources, whereas the best commercial product we tested reached only 82% in this subcategory. Similarly, our procedures remediated 99% of primary registry activities, while commercial products did not exceed 86%. Furthermore, while ancillary objects are often ignored by commercial remediation procedures, our procedures remediated 95% of ancilliary files and 98% of ancilliary registry activities. The portion of file and registry resources that were not remediated by our procedures correspond to behaviors that were never observed while collecting traces. This illustrates the primary limitation of our dynamic analysis-based approach and highlights a clear avenue for improvement in future work. Finally, our procedures remediated 100% of primary process resources. However, the performance on ancillary processes is significantly lower. This is a result of the fact that our processes do not have access to enough information to discern a benign process from a process spawned by the malware using a pre-existing benign file.

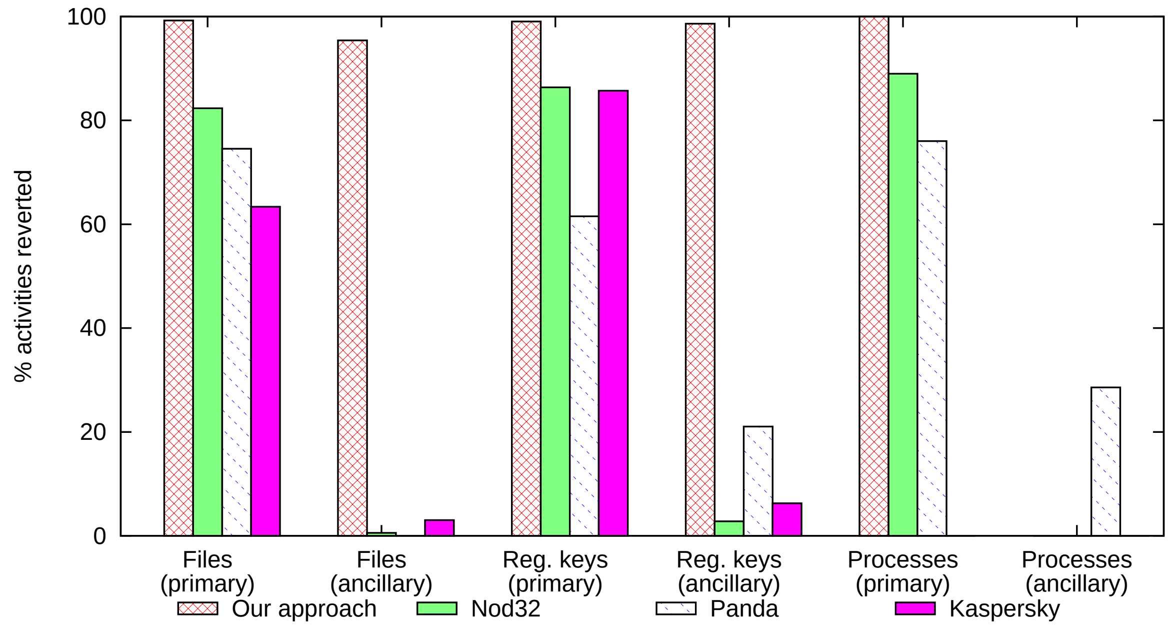


Figure 7: Comparison of the completeness of our automatically generated remediation procedures with the completeness of the procedures employed in three top-rated commercial malware detectors.

5.3 False Positives

To quantify false positives, we compared the set of resources affected by each malware sample in each test environment with the set of resources our procedures remediated in each test environment. Any remediated resource not affected by the corresponding malware sample in at least one trace is considered a false positive. We found that only one of our procedures produced any false positives. The cause of this false positive, not surprisingly, was a high-level behavior argument specified by a very general regular expression. This implies that the nondeterminism demonstrated by the corresponding malware sample was too complex to be easily described by a regular language. Thus, one area for future work is utilizing more expressive language classes, such as context-free grammars, for generalizing argument values.

6 Discussion

We are aware of some limitations of our system. Some of these limitations could be exploited by attackers to cause the system to produce remediation procedures that are of limited value. In this section, we discuss these limitations and present some solutions that we will investigate in the future to address the limitations.

We constructed the models that we use to detect high-level behaviors by leveraging years of experience in malware analysis, and we carefully tested all models to ensure that they cannot be evaded. However, since we cannot prove that these models are perfect, we must take into account the possibility that attackers could find new ways to perform some high-level malicious activities without being detected. Moreover, in our proof-of-concept implementation, multiple execution traces are obtained by executing the same malware in several different operating system configurations. If attackers introduced dangerous behaviors to their malicious programs that are not triggered in our monitoring environment, then the resulting procedure would not be able to remediate such behaviors. Clearly, one area for future work is in expanding the coverage of the dynamic behavioral analysis. While our approach covers some of the potential behavior of the sample, more sophisticated techniques [12, 21] can be applied to increase the likelihood that all relevant paths through the malware are explored.

The high-level behaviors observed in multiple execution traces are clustered to identify the instances of the same behavior. If the clusters we generate did not include all the instances of the same behavior, or if they included instances of different behaviors, then the remediation procedures constructed by generalizing the behaviors associated to each cluster would be too specific or too generic. An attacker could write malicious programs that manifest certain behaviors to break the clustering. Similarly, the regular expressions used by our remediation procedures to identify affected resources are generalized heuristically. Attackers could develop malicious programs that affect resources in a way that induces us to perform very aggressive generalization (e.g. create files with random names anywhere in the file system) and thus to generate remediation procedures that remove benign files. We plan to address these problems in the future. One approach is to introduce a feedback loop while clustering behaviors and generating regular expressions to validate the quality of the results. This feedback loop would repeat the process until no further progress can be made. Finally, we assert that it is not possible to cause our algorithm to generate a procedure that modifies existing files in a harmful way. This follows from the fact that system files are only ever restored to their original state by the procedure, not modified.

We currently generate a remediation procedure for each malware sample we analyze. We plan to extend our system to generate remediation procedures that cover more than one malware sample. For example, it would be useful to generate remediation procedures that are capable of operating on all samples for a given malware family. Because the generated procedures will likely have to account for a much higher degree of nondeterminism than those that target only a single sample, additional care must be taken to ensure that the high-level behaviors models are not too general, thus resulting in false positives.

7 Conclusion

In this paper, we have presented a technique for automatically generating malware remediation procedures. Given a malware binary, our system produces executable code that removes the harmful effects of executing that malware on a system. We use dynamic analysis and *behavior generalization* to account for the difficulties posed by real malware, thus allowing our procedures to effectively remediate many possible executions of the malware without witnessing the actual infection take place. This contribution represents a major break with previous automatic remediation techniques, which required detailed information about the particular infection being targeted. We implemented our technique and evaluated its effectiveness on more than 200 malware binaries. The performance of our prototype is quite good: on average, 98% of the harmful effects are remediated, and we encountered only a single false positive. In the future, we plan to build on this work by extending it to work on entire families, as well as exploring more precise techniques for generalizing observed malware behaviors.

References

[1] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, Hamburg, Germany, Apr. 2006.

[2] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. `http://fabrice.bellard.free.fr/qemu/`.

[4] M. Christodorescu, C. Kruegel, and S. Jha. Mining specifications of malicious behavior. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Dubrovnik, Croatia, 2007.

[5] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[6] P. Foggia. The vflib graph matching library, version 2.0. `http://amalfi.dis.unina.it/graph/db/vflib-2.0/`.

[7] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[8] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, 2009.

[9] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2006.

[10] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference (ACSAC)*, 2003.

[11] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.

[12] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2007.

[13] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[14] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2005.

[15] E. Passerini, R. Paleari, and L. Martignoni. How good are malware detectors at remediating infected systems? In *6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Como, Italy, July 2009.

[16] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30(5):25.1–25.54, Aug. 2008.

[17] A. Raman, P. Andreae, and J. Patrick. A beam search algorithm for PFSA inference. *Pattern Analysis and Applications*, 1(2):121–129, 1998.

[18] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[19] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *12th Symposium on Network and Distributed Systems Security (NDSS)*, 2005.

[20] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *14th USENIX Security Symposium*, Baltimore, MD, 2005.

[21] H. Yin, D. Song, M. Egele, E. Kirda, and C. Kruegel. Panorama: Capturing system-wide information flow for malware detection and analysis. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2007.