Exploring Multiple Execution Paths for Malware Analysis

Andreas Moser, Christopher Kruegel, and Engin Kirda
Secure Systems Lab
Technical University Vienna
{andy,chris,ek}@seclab.tuwien.ac.at

Abstract

Malicious code (or malware) is defined as software that fulfills the deliberately harmful intent of an attacker. Malware analysis is the process of determining the behavior and purpose of a given malware sample (such as a virus, worm, or Trojan horse). This process is a necessary step to be able to develop effective detection techniques and removal tools. Currently, malware analysis is mostly a manual process that is tedious and time-intensive. To mitigate this problem, a number of analysis tools have been proposed that automatically extract the behavior of an unknown program by executing it in a restricted environment and recording the operating system calls that are invoked.

The problem of dynamic analysis tools is that only a single program execution is observed. Unfortunately, however, it is possible that certain malicious actions are only triggered under specific circumstances (e.g., on a particular day, when a certain file is present, or when a certain command is received). In this paper, we propose a system that allows us to explore multiple execution paths and identify malicious actions that are executed only when certain conditions are met. This enables us to automatically extract a more complete view of the program under analysis and identify under which circumstances suspicious actions are carried out. Our experimental results demonstrate that many malware samples show different behavior depending on input read from the environment. Thus, by exploring multiple execution paths, we can obtain a more complete picture of their actions.

1 Introduction

Malware is a generic term used to describe all kinds of malicious software (e.g., viruses, worms, or Trojan horses). Malicious software not only poses a major threat to the security and privacy of computer users and their data, but is also responsible for a significant amount of financial loss. Unfortunately, the problem of malicious code is likely to continue to grow in the future, as malware writing is quickly turning into a profitable business [26]. Malware authors often sell their creations to miscreants, who then use the malicious code to compromise large numbers of machines that are linked together in so-called botnets. These botnets are then abused as platforms to launch denial-of-service attacks or as spam relays. An important indication of the significance of the problem is that even people without any particular interest in computers are aware of worms such as CodeRed or Sasser. This is because security incidents affect millions of users and regularly make the headlines of mainstream news sources.

The most important line of defense against malicious code are virus scanners. These scanners typically rely on a database of signatures that characterize known malware instances. Whenever an unknown malware sample is found in the wild, it is usually necessary to update the signature database accordingly, so that this novel malware piece can be detected by the scan engine. To this end, it is very important to be able to quickly analyze an unknown malware sample and understand its behavior and effect on the system. In addition, the knowledge about the functionality of malware is important for its removal. That is, to be able to effectively remove a piece of malware from an infected machine, it is usually not sufficient to delete the binary itself. Often, it is also necessary to remove the residues left behind by the malicious code (such as undesirable registry entries, services, or processes) and undo changes made to legitimate files. All these actions require a detailed understanding of the malicious code and its behavior.

The traditional approach to analyze the behavior of an unknown program is to execute the binary in a restricted environment and observe its actions. The restricted environment is often a debugger, used by a human analyst to manually step through the code in order to understand its functionality. Unfortunately, anti-virus companies receive up to *several hundred* new malware samples each day. Clearly, the analysis of these malware samples cannot be performed completely manually.

In a first step towards an automated solution, a number of dynamic malware testing systems were proposed. These systems, such as CWSandbox [29], the Norman SandBox [25], TTAnalyze [2], or Cobra [28] automatically load the sample to be analyzed into a virtual machine environment and execute it. While the program is running, its interaction with the operating system is recorded. Typically, this involves recording which system calls are invoked, together with their parameters. The result of an automated analysis is a report that shows what operating system resources (e.g., files or Windows registry entries) a program has created or accessed. Some tools also allow the system to connect to a local network (or even the Internet) and monitor the network traffic. Usually, the generated reports provide human analysts with an overview on the behavior of the sample and allow them to quickly decide whether a closer, manual analysis is required. Hence, these automated systems free the analysts of the need to waste time on already known malware. Also, some tools are already deployed on the Internet and act as live analysis back-ends for honeypot installations such as Nepenthes [1]. Unfortunately, current analysis systems also suffer from a significant drawback: their analysis is based on a *single* execution trace only. That is, their reports only contain the interaction that was observed when the sample was run in a particular test environment at a certain point in time. Unfortunately, this approach has the potential to miss a significant fraction of the behavior that a program might exhibit under varying circumstances.

Malware programs frequently contain checks that determine whether certain files or directories exist on a machine and only run parts of their code when they do. Others require that a connection to the Internet is established or that a specific mutex object does not exist. In case these conditions are not met, the malware may terminate immediately. This is similar to malicious code that checks for indications of a virtual machine environment, modifying its behavior if such indications are present in order to make its analysis in a virtual environment more difficult. Other functionality that is not invoked on every run are malware routines that are only executed at or until a certain date or time of day. For example, some variants of the Bagle worm included a check that would deactivate the worm completely after a certain date. Another example is the Michelangelo virus, which remains dormant most of the time, delivering its payload only on March 6 (which is Michelangelo's birthday). Of course, functionality can also be triggered by other conditions, such as the name of the user or the IP address of the local network interface. Finally, some malware listens for certain commands that must be sent over a control channel before an activity is started. For example, bots that automatically log into IRC servers often monitor the channel for a list of keywords that trigger certain payload routines.

When the behavior of a program is determined from a single run, it is possible that many of the previously mentioned actions cannot be observed. This might lead a human analyst to draw incorrect conclusions about the risk of a certain sample. Even worse, when the code fails at an early check and immediately exits, the generated report might not show any malicious activity at all. One possibility to address this problem is to attempt to increase test coverage. This could be done by running the executable in different environments, maybe using a variety of operating system versions, installed applications, and data/time settings. Unfortunately, even with the help of virtual machines, creating and maintaining such a testing system can be costly. Also, performing hundreds of tests with each sample is not very efficient, especially because many environmental changes have no influence on the program execution. Moreover, in cases where malicious code is expecting certain commands as input or checking for the existence of non-standard files (e.g., files that a previous exploit might have created), it is virtually impossible to trigger certain actions.

In this paper, we propose a solution that addresses the problem of test coverage and that allows automated malware analysis systems to generate more comprehensive reports. The basic idea is that we explore multiple execution paths of a program under test, but the exploration of different paths is driven by monitoring how the code uses certain inputs. More precisely, we dynamically track certain input values that the program reads (such as the current time from the operating system, the content of a file, or the result of a check for Internet connectivity) and identify points in the execution where this input is used to make control flow decisions. When such a decision point is identified, we first create a snapshot of the current state of the program execution. Then, the program is allowed to continue along one of the execution branches, depending on the actual input value. Later, we return to the snapshot and rewrite the input value such that the other branch is taken. This allows us to explore both program branches. In addition, we can determine under which conditions certain code paths are executed.

For a simple example, consider a program that checks for the presence of a file. During execution, we track the result of the operating system call that checks for the existence of that file. When this result is later used in a conditional branch by the program, we store a snapshot of the current execution state. Suppose, for example, that the file does not exist, and the program quickly exits. At this point, we rewind the process to the previously stored state and rewrite the result such that it does reports the file's existence. Then, we can explore the actions that the program performs under the condition that the file is there.

We have developed a system for Microsoft Windows that allows us to dynamically execute programs and track the input that they read. Also, we have implemented a mechanism to take snapshots of executing processes and later revert to previously stored states. This provides us with the means to explore the execution space of malware programs and to observe behavior that is not seen by traditional malware analysis environments. To demonstrate the feasibility of our approach, we analyzed a large number of real-world malware samples. In our experiments, we were able to identify time checks that guarded damage routines and different behavior depending on existence of certain files. Also, we were able to automatically extract a number of command strings for a bot with their corresponding actions.

To summarize, the contributions of this paper are as follows:

- We propose a dynamic analysis technique that allows us to create comprehensive reports on the behavior of malicious code. To this end, our system explores multiple program paths, driven by the input that the program processes. Also, our system reports the set of conditions on the input under which particular actions are triggered.
- We developed a tool that analyzes Microsoft Windows programs by executing them in a virtual-machine-based environment. Our system keeps track of user input and can create snapshots of the current process at control flow decision points. In addition, we can reset a running process to a previously stored state and consistently modify its memory such that the alternative execution path is explored.
- We evaluated our system on a large number of real-world malware samples and demonstrate that we were able to identify behavior that cannot be observed in single execution traces.

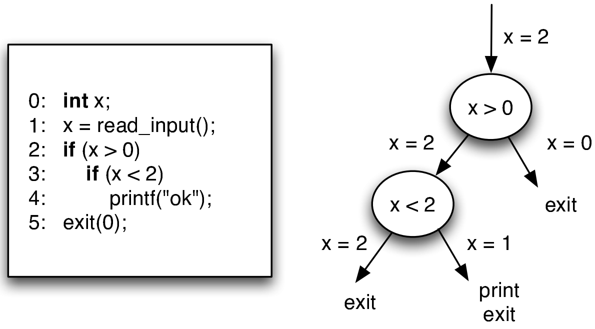
2 System Overview

The techniques described in this paper are an extension to an existing system for automated malware analysis [2]. This tool is based on Qemu [3], a fast virtual machine emulator. Using Qemu's emulation of an Intel x86 host system, a Windows 2000 guest operating system is installed. The choice of Windows and the Intel x86 architecture was motivated by the fact that the predominant fraction of malware is developed for this platform. The analysis works by loading the (malware-)program into the emulated Windows environment, starting its execution, and subsequently monitoring its activity. To this end, the analysis tool analyzes all operating system calls that are invoked by the binary. For each system call, the analysis tool records the type of service requested and the corresponding arguments. Based on the system calls observed during execution, a report is generated that summarizes the security-relevant actions. These actions currently include the creation and modification of files and Windows registry entries, interprocess communication, and basic network interaction.

The existing analysis tool implements some virtual machine introspection capabilities; in particular, it is able to attribute each instruction that is executed by the emulated processor to an operating system process (or the kernel) of the guest system. This allows us to track only those system calls that are invoked by the code under analysis. Also, the system provides a mechanism to copy the content of complex data structures, which can contain pointers to other objects in the process' virtual address space, from the Windows guest system into the host system. This is convenient in order to be able to copy the system call arguments from the emulated system into the analysis environment. Unfortunately, the existing system only collected a single execution trace.

Multiple execution paths. To address the problem that a single execution trace typically produces only part of the complete program behavior, we extended the analysis tool with the capability to explore multiple execution paths. The goal is to obtain a number of different execution paths, and each path possibly reveals some specific behavior that cannot be observed in the other traces. The selection of *branching points* – that is, points in the program execution where both alternative continuations are of interest – is based on the way the program processes input data. More precisely, when a control flow decision is based on some input value that was previously read via a system call, the program takes one branch (which depends on the outcome of the concrete check). At this point, we ask ourselves the following question: Which behavior could be observed if the input was such that the other branch was taken?

To answer this question, we label certain inputs of interest to the program and dynamically track their propagation during execution. Similar to the propagation of taint information used by other authors in previous work [12, 23], our system monitors the way these input values are moved and manipulated by the process. Whenever we detect a control flow decision based on a labeled value, the current content of the process address space is stored. Then, execution continues normally. When the process later wishes to terminate, it is automatically reset to the previously stored snapshot. This is done by replacing the current content of the process address space with the previously stored values. In addition, we rewrite the input value that was used in the control flow decision such that the outcome of this decision is reversed. Then, the process continues its execution along the other branch. Of course, it is possible that multiple branching in a row are encountered. In this case, the execution space is explored by selecting continuation points in a depth-first order.



```
0:  int x;
1:  x = read_input();
2:  if (x > 0)
3:      if (x < 2)
4:          printf("ok");
5:  exit(0);
```

Figure 1. Exploration of multiple execution paths.

For an example on how multiple execution paths of a program can be explored, consider Figure 1. Note that although this example is shown in C code (to make it easier to follow), our system works directly on x86 binaries. When the program is executed, it first receives some input and stores it into variable *x* (on Line 1). Note that because *x* is considered interesting, it is labeled. Assume that in this concrete run, the value stored into *x* is 2. On Line 2, it is compared to 0. At this point, our system detects a comparison operation that involves labeled data. Thus, a snapshot of the current process is created. Then, the process is allowed to continue. Because the condition is satisfied, the if-branch is taken and we record the fact that *x* has to be larger than 0. On Line 3, the next check fails. However, because the comparison again involves labeled data, another snapshot is created. This time, the process continues on the else-branch and is about to call `exit`. Because there are still unexplored paths (i.e., there exist two states that have not been visited), the process is reverted to the previous (second) state. Our system inspects the comparison at Line 3 and attempts to rewrite *x* such that the check succeeds. For this, the additional constraint $x > 0$ has to be observed. This yields a solution for *x* that equals 1. The value of *x* is updated to 1 and the process is restarted. This time, the `print` statement on Line 4 is invoked. When the process is about to exit on Line 5, it is reset to the first snapshot. This time, the system searches a value for *x* that fails the check on Line 2. Because there are no additional constraints for *x*, an arbitrary, non-positive integer is selected and the process continues along the else-branch. This time, the call to `exit` is permitted, and the analysis process terminates with a report that indicates that a call to `print` was found under the condition that the input *x* was 1 (but not 0 or 2).

Consistent memory updates. Unfortunately, when rewriting a certain input value to explore an alternative execution path, it is typically not sufficient to change the single memory location that is used by the control flow decision. Instead, it is necessary to consistently update (or rewrite) all values in the process address space that are related to the input. The reason is that the original input value might have been copied to other memory locations, and even used by the program as part of some previous calculations. When only a single instance of the input is modified, it is possible that copies of the original value remain in the program's data section. This can lead to the execution of invalid operations or the exploration of impossible paths. Thus, whenever an input value is rewritten, it is necessary to keep the program state consistent and appropriately update all copies of the input, as well as results of previous operations that involve this value. Also, we might not have complete freedom when choosing an alternative value for a certain input. For example, an input might have been used in previous comparison operations and the resulting constraints need to be observed when selecting a value that can revert the control flow decision at a branching point. It is even possible that no valid alternative value exists that can lead to the exploration of the alternative path. Thus, to be able to consistently update and input and its related values, it is necessary to keep track of *which* memory locations depend on a certain input and *how* they depend on this value.

3 Path Exploration

To be able to explore multiple program paths, two main components are required. First, we need a mechanism to decide when our system should analyze both program paths. To this end, we track how the program uses data from certain input sources. Second, when an interesting branching point is located, we require a mechanism to save the current program state and reload it later to explore the alternative path. The following two subsections discuss these two components in more detail.

3.1 Tracking Input

In traditional taint-based systems, it is sufficient to know that a certain memory location depends on one or more input values. To obtain this information, such systems typically rely on three components: a set of taint sources, a shadow memory, and extensions to the machine instructions that propagate the taint information.

Taint sources are used to initially assign labels to certain memory locations of interest. For example, Vigilante [11] is a taint-based system that can detect computer worms that propagate over the network. In this system, the network is considered a taint source. As a result, each new input byte that is read from the network card by the operating system receives a new label. The shadow memory is required to keep track of which labels are assigned to which memory locations at a certain point in time. Usually, a shadow byte is used for each byte of physical machine memory. This shadow byte stores the label(s) currently attached to the physical memory location. Finally, extensions to the machine instructions are required to propagate taint information when an operation manipulates or moves labeled data. The most common propagation policy ensures that the result of an operation receives the union of the labels of the operation's arguments. For example, consider an `add` machine instruction that adds the constant value 10 to a memory location M_1 and stores the result at location M_2. In this case, the system would use the shadow memory to look up the label attached to M_1 and attach this label to M_2. Thus, after the operation, both locations M_1 and M_2 share the same label (although their content is different).

In principle, we rely on a taint-based system as previously described to track how the program under analysis processes input values. That is, we have a number of taint sources that assign labels to input that is read by the program, and we use a shadow memory to keep track of the current label assigned to each memory location (including the processor registers). Taint sources in our system are mostly system calls that return information that we consider relevant for the behavior of malicious code. This includes system calls that access the file system (e.g., check for existence of file, read file content), the Windows registry, and the network. Also, system calls that return the current time or the status of the network connection are interesting. Whenever a relevant function (or system call) is invoked by our program, our system automatically assigns a new label to each memory location that receives this function's result. Sometimes, this means that a single integer is labeled. In other cases, for example, when the program reads from a file or the network, the complete return buffer is labeled, using one unique label per byte.

Inverse mapping. In addition to the shadow memory, which maps memory locations to labels, we also require an *inverse mapping*. The inverse mapping stores, for each label, the addresses of all memory locations that currently hold this label. This information is needed when a process is reset to a previously stored state and a certain input variable must be rewritten. The reason is that when a memory location with a certain label is modified, it is necessary to simultaneously change all other locations that have the same label. Otherwise, the state of the process becomes inconsistent. For example, consider the case in which the value of labeled input x is copied several times before it is eventually stored at memory location y. Furthermore, assume that y is used as argument by a conditional branch. To explore the alternate execution branch, the content of y must be changed. However, via a chain of intermediate locations, this value ultimately depends on x. Thus, all intermediate locations need to be modified appropriately. To this end, a mapping is required that helps us to quickly identify all locations that currently share the same label.

```
0:   ...
1:   x = read_input();
2:   check(x);
3:   printf("%d", x);
4:   ....
5:
6:   void check(int magic) {
7:      if (magic != 0x1508)
8:          exit(1);
9:   }
```

Figure 2. Consistent memory updates.

To underline the importance of a consistent memory update, consider the example in Figure 2. Assume that the function `read_input` on Line 1 is a taint source. Thus, when the program executes this function, variable *x* is labeled. In our example, the program initially reads the value 0. When the `check` routine is invoked, the value of variable *x* is copied into the parameter *magic*. As part of this assignment, the variable *magic* receives the label of *x*. When *magic* is later used in the check on Line 7, a snapshot of the current state is taken (because the outcome of a conditional branch depends on a labeled value). Execution continues but quickly terminates on Line 8. At this point, the process is reverted to the previously stored snapshot and our system determines that the value of *magic* has to be rewritten to $0x1508$ to take the if-branch. At this point, the new value has to be propagated to all other locations that share the same label (in our case, the variable *x*). Otherwise, the program would incorrectly print the value of 0 instead of $0x1508$ on Line 3.

Linear dependencies. In the previous discussion, the initial input value was copied to new memory locations before being used as an argument in a control flow decision. In that case, rewriting this argument implied that all locations that share the same label had to be updated with the same value. So far, however, we have not considered the case when the initial input is not simply copied, but used as operand in calculations. Using the straightforward taint propagation mechanism outlined above, the result of an operation with a labeled argument receives this argument's label. This also happens when the result of an operation has a different value than the argument. Unfortunately, that leads to problems when rewriting a variable at a snapshot point. In particular, when different memory locations share the same label but hold different values, one cannot simply overwrite these memory locations with a single, new value.

We solve this problem by assigning a *new label* to the result of any operation (different than copying) that involves labeled arguments. In addition, we have to record how the value with the new label depends on the value(s) with the old label(s). This is achieved by creating a new constraint that captures the relationship between the old and new labels, depending on the semantics of the operation. The constraint is then added to a *constraint system* that is maintained as part of the execution state of the process. Consider the simple example where a value with label l_0 is used by an add operation that increases this value by the constant 10. In this case, the result of the operation receives a new label l_1. In addition, we record the fact that the result of the operation with l_1 is equal to the value labeled by l_0 plus 10. That is, the constraint $l_1 = l_0 + 10$ is inserted into the constraint system. The approach works similarly when two labeled inputs, one with label l_0 and the other with label l_1 are summed up. In this case, the result receives a new label l_2 and we add the constraint $l_2 = l_0 + l_1$.

In our current system, we can only model linear relationships between input variables. That is, our constraint system is a linear constraint system that can store terms in the form of $\{c_n * l_n + c_{n-1} * l_{n-1} + \ldots + c_1 * l_1 + c_0\}$ where the c_i are constants. These terms can be connected by equality or inequality operators. To track linear dependencies between labels, the taint propagation mechanism of the machine instructions responsible for addition, subtraction, and multiplication had to be extended.

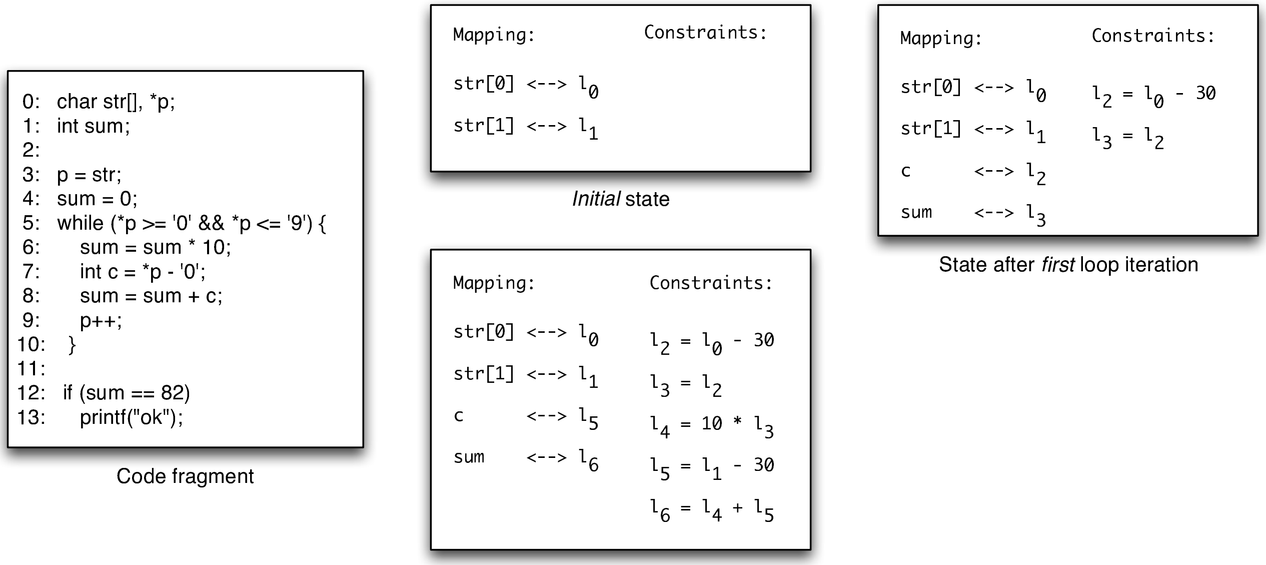
Using the information provided by the linear constraint system, it is possible to correctly update all memory locations that depend on an input value x via linear relationships. Consider the case where a conditional control flow decision uses a value with label l_n. To explore the alternative branch of this decision, we have to rewrite the labeled value such that the outcome of the condition is reverted. To do this consistently, we first use the linear constraint system to identify all labels that are related to l_n. This provides us with the information which memory locations have some connection with l_n, and thus, must be updated as well. In a second step, a linear constraint solver is used to determine concrete values for these memory locations.

Two labels l_s and l_t are related either (a) when they appear together in the same constraint or (b) when there exists a sequence of labels $\{l_{i_0}, \ldots, l_{i_n}\}$ such that $l_s = l_{i_0}$, $l_t = l_{i_n}$, and $l_i, l_{i+1}\ \forall_{i=0}^{n-1}$ appear in the same constraint. More formally, the binary relation *related* is the transitive closure of the binary relation *appears in the same constraint*. Thus, when a value with label l_n should be rewritten, we first determine all labels that are *related* to l_n in the constraint system. Then, we extract all constraints that contain at least one of the labels related to l_n. This set of constraints is then solved, using a linear constraint solver (we use the Parma Polyhedral Library).

When the constraint system has no solution, the labeled value cannot be changed such that the outcome of the condition is reverted. In this case, our system cannot explore the alternative path, and it continues with the next snapshot stored. When a solution is found, on the other hand, this solution can be directly used to consistently update the process' state. To this end, we can directly use, for each label, the value that the solver has determined to update the corresponding memory locations. This works because all (linear) dependencies between values are encoded by the respective constraints in the constraint system. That is, a solution of the constraint system respects the relationships that have to hold between memory locations. All memory locations that share the same label receive the same value. However, as expected, when memory locations have different labels, they can also receive different values. These values respect the relationships introduced by the operation previously executed by the process and captured by the corresponding constraints in the constraint system.

To illustrate the concept of linear dependencies between values and to show how their dependencies are captured by the constraint system, consider Figure 3. The example shows the labels and constraints that are introduced when a simple atoi function is executed. The goal of this function is to convert a string into the integer value that this string represents. For this example, we assume that the function is executed on a string *str* with three characters; the first two are the ASCII character equivalent of the number 0 (which is 30). The third one has the value 0 and terminates the string. We assume that interesting input was read into the string; as a result, the first character *str[0]* has label l_0 and *str[1]* has label l_1.

The figure shows the initial mapping between program variables and labels. For this initial state, no constraints have been identified yet. After the first loop iteration, it can be seen that the variables *c* and *sum* are also labeled. This results from the operations on Line 7 and Line 8, respectively. The relationship between the variables are captured by the two constraints. Because *sum* was 0 before this loop iteration, variables *sum* and *c* hold the same value. This is expressed by the constraint $l_3 = l_2$. Note that this example is slightly simplified. The reason is that the checks performed by the while-statement on Line 5 lead to the creation of additional constraints that ensure that the values of *str[0]* and *str[1]* are between 30 (ASCII value for '0') and 39 (ASCII value for character '9'). Also, because the checks operate on labeled data, the system creates snapshots for each check and attempts to explore additional paths later. For these alternative paths, the string elements are rewrit-



```
 0:  char str[], *p;
 1:  int sum;
 2:
 3:  p = str;
 4:  sum = 0;
 5:  while (*p >= '0' && *p <= '9') {
 6:      sum = sum * 10;
 7:      int c = *p - '0';
 8:      sum = sum + c;
 9:      p++;
10:  }
11:
12:  if (sum == 82)
13:      printf("ok");
```

Code fragment

```
Mapping:                Constraints:

str[0] <--> l0

str[1] <--> l1
```

Initial state

```
Mapping:                Constraints:

str[0] <--> l0          l2 = l0 - 30

str[1] <--> l1          l3 = l2

c      <--> l2

sum    <--> l3
```

State after *first* loop iteration

```
Mapping:                Constraints:

str[0] <--> l0          l2 = l0 - 30

str[1] <--> l1          l3 = l2

c      <--> l5          l4 = 10 * l3

sum    <--> l6          l5 = l1 - 30

                        l6 = l4 + l5
```

State after *second* loop iteration

Figure 3. Constraints generated during program execution.

ten to be characters that do not represent numbers. In these cases, the while-loop would terminate immediately.

In the example, the program reaches the check on Line 11 after the second loop iteration. Given the original input for *str*, *sum* is 0 at this point and the else-branch is taken. However, because this conditional branch involves the value *sum* that is labeled with l_6, a snapshot of the current program state is created. When this snapshot is later restored, our system needs to rewrite *sum* with the value 82 be able to take the if-branch. To determine how the program state can be updated consistently, the constraint system is solved for $l_6 = 82$. A solution to this system can be found ($l_0 = 38$, $l_1 = 32$, $l_2 = l_3 = 8$, $l_4 = 80$, and $l_5 = 2$). Using the mappings, this solution determines how the related memory locations can be consistently modified. As expected, *str[0]* and *str[1]* are set to the characters '8' and '2', respectively. The variable *c* is also set to 2.

Non-linear dependencies. The `atoi` function discussed previously represents a more complex example of what can be captured with linear relationships. However, it is also possible that a program performs operations that cannot be represented as linear constraints. These operations involve, for example, bitwise operators such as `and`, `or` or a lookup in which the input value is used as an index into a table. In case of a non-linear relationship, our current system cannot infer the assignment of appropriate values to labels such that a certain memory location can be rewritten as desired. Thus, whenever an operation creates a non-linear dependency between labels l_i and l_j, we no longer can consistently update the state when any label related to l_i or l_j should be rewritten. To address this problem, we maintain a set N that keeps track of all labels that are part of non-linear dependencies. Whenever a label should be rewritten, all related labels are determined. In case any of these labels is in N, the state cannot be consistently changed and the alternative path cannot be explored.

3.2 Saving and Restoring Program State

The previous section explained our techniques to track the propagation of input values during program execution. Every memory location that depends on some interesting input has an attached label, and the constraint system determines how values with different labels are related to each other. Based on this information, multiple paths in the execution space can be explored. To this end, our system monitors the program execution for conditional operations that use one (or two) labeled arguments. When such a branch instruction is identified, a snapshot of the current process state is created.

The snapshot of the current execution state contains the content of the complete virtual address space that is in use. In addition, we have to store the current mappings and the constraint system. But before the process is allowed to continue, one additional step is needed. In this step, we have to ensure that the conditional operation itself is taken into account. The reason is that no matter which branch is actually taken, this conditional operation enforces a constraint on the possible value range of the labeled argument. We call this constraint a *path constraint*. The path constraint has to be remembered and taken into account in case the labeled value is later rewritten (further down the execution path). Otherwise, we might create inconsistent states or reach impossible paths. When the if-branch of the conditional is taken (that is, it evaluates to true for the current labeled value), the condition is directly used as path constraint. Otherwise, when the else-branch is followed, the condition has to be reversed before it is added to the constraint system. To this end, we simply take the condition's negation.

For example, recall the first program that we showed in Figure 1. This program uses two checks to ensure that $x > 0$ and $x < 2$ before the `print` function is invoked. When the first if-statement is reached on Line 2, a snapshot of the state is created. Because x had an initial value of 2, the process continues along the if-branch. However, we have to record the fact that the if-branch can only be taken when the labeled value is larger than 0. Assume that the label of x is l_0. Hence, the appropriate path constraint $l_0 > 0$ is added to the constraint system. At the next check on Line 3, another snapshot is created. This time, the else-branch is taken, and we add the path constraint $l_0 >= 2$ to the constraint system (which, because of the else-branch, is the negation of the conditional check $x < 2$). When the process is about to terminate on Line 5, it is reset to the previously stored state. This time, the if-branch on Line 3 must be taken. To this end, we add the path constraint $l\ _0 < 2$ to the constraint system. At this point, the constraint system contains *two* entries. One is the constraint just added (i.e., $l\ _0 < 2$). The other one stems from the first check and requires that $l_0 > 0$. When these constraints are analyzed, our solver determines that $l_0 = 1$. As a result, x is rewritten to 1 and the program continues with the call to `print`.

When a program state is restored, the first task of our system is to load the previously saved content of the program's address space and overwrite the current values with the stored ones. Then, the saved constraint system is loaded. Similar to the case in which the first branch was taken, it is also necessary to add the appropriate path constraint when following the alternative branch. To this end, the path constraint that was originally used is reversed (that is, we take its negation). This new path constraint is added to the constraint system and the constraint solver is launched. When a solution is found, we use the new values for all related labels to rewrite the corresponding memory locations in a consistent fashion. As mentioned previously, when no solution is found, the alternative branch cannot be explored.

Note that at any point during the program's execution, the solution space of the constraint system specifies all possible values that the labeled input can have in order to reach this point in the program execution. This information is important to determine the conditions under which certain behavior is exhibited. For example, consider that our analysis observes an operating system call that should be included into the report of suspicious behavior. In this case, we can use the solution(s) to the constraint system to determine all values that the labeled input can take to reach this call. This is helpful to understand the conditions under which certain malicious behavior is triggered. For example, consider a worm that deactivates itself after a certain date. Using our analysis, we can find the program path that exhibits the malicious behavior. We can then check the constraint system to determine under which circumstances this path is taken. This yields the information that the current time has to be before a certain date.

4 System Implementation

We implemented the concepts introduced in the previous section to explore the execution space of Windows binaries. More precisely, we extended our previous malware analysis tool [2] with the capability to automatically label input sources of interest and track their propagation using standard taint analysis (as, for example, realized in [12, 23]). In addition, we implemented the mechanisms to consistently save and restore program states. This allows us to automatically generate more complete reports of malicious behavior than our original tool. The reports also contain the information under which circumstances a particular behavior is observed. In this section, we describe and share implementation details that we consider interesting for the reader.

4.1 Creating and Restoring Program Snapshots

Our system (and the original analysis tool) is built on top of the system emulator Qemu [3]. Thus, the easiest way to save the execution state of a program would be to save the state of the complete virtual machine (Qemu already supports this functionality). Unfortunately, when a sample is analyzed, many snapshots have to be created. Saving the image of the complete virtual machine costs too much time and resources. Thus, we require a mechanism to take snapshots of the process' image only. To this end, we developed a Qemu component that can identify the active memory pages of a process that is executing in the guest operating system (in our case, Microsoft Windows). This is done by analyzing the page table directory that belongs to the Windows process. Because Qemu is a PC emulator, we have full access to the emulated machine's physical memory. Hence, we can access the Windows kernel data structures and perform the same calculations as the Windows memory management code to determine the physical page that belongs to a certain virtual address of the process under analysis. Once we have identified all pages that are memory mapped for our process, we simply copy the content of those that are flagged valid. In addition, when creating a snapshot of a process, we have to make a copy of the virtual CPU registers, the shadow memory, and the constraint system.

The method described above has one limitation. We cannot store (or reset) memory that is paged out on disk. This limitation stems from the fact that although we can access the complete main memory from outside, we cannot read the content on the virtual hard disk (without understanding how the Windows file system and swapping is implemented). Thus, we have to disable swapping and prevent the guest OS from moving memory pages to the disk where they can no longer be accessed. In our experiments, we found that this limitation is not a problem as our malware samples had very modest memory demands and never exceeded the 256 MB main memory of the guest OS.

To reset a process such that it continues from a previously saved snapshot, we use a procedure that is similar to the one for storing the execution state. First, we identify all mapped pages that belong to our process of interest. Then, for each page that was previously saved, we overwrite the current content with the one that was stored. When the pages are restored, we also reset the virtual CPU to its saved state. Note that it is possible that the process has allocated more pages than were present at the time when the snapshot was taken. This is the case when the program has requested additional memory from the operating system. Of course, these new pages cannot be restored. Fortunately, this is no problem and does not alter the behavior of the process. The reason is that all references in the original pages that now point to the new memory areas are reverted back to the values that they had at the time of the snapshot (when the new pages did not exist yet). The only problem is that the newly allocated pages are lost for the process, but still considered in use by the operating system. This "memory leak" might become an issue when, for example, a memory allocating routine is executed various times when different execution paths are explored. Although we never experienced problems in our experiments, one possible solution would be to inject code into the guest OS that releases the memory.

An important observation is that a process can only be reset to a previously stored state when it is executing in user mode. When a process is executing kernel code, reverting it back to a user mode state can leave data structures used by the Windows kernel in an inconsistent state. The same is true when the operating system is executing an interrupt handling routine. Typically, resetting the process when not in user mode leads to a crash or freezes the system.

Our current implementation allows us to reliably reset processes to previous execution states. However, one has to consider the kernel state when snapshots are taken or restored. In particular, we have to address the problem that a resource might be returned to the operating system after a snapshot has been taken. When we later revert to the previously stored snapshot, the resource is already gone, and any handles to it are stale. For example, such a situation can occur when a file is closed after a snapshot is made. To address this problem, we never allow a process to close or free any resource that it obtains from the operating system. To this end, whenever an application calls the `NtClose` function or attempts to return allocated memory to the OS, we intercept the function and immediately return to the user program. From the point of view of the operating system, no handle is ever closed. Thus, when the process is reset to an old state, the old handles are still valid.

4.2 Identification of Program Termination

The goal of our approach is to obtain a comprehensive log of the activities of a program on as many different execution paths as possible. Thus, before reverting to a previously stored state, the process is typically allowed to run until it exits normally or crashes. Of course, our system cannot allow the process to actually terminate. Otherwise, the guest operating system removes the process-related entries from its internal data structures (e.g., scheduler queue) and frees its memory. In this case, we would lose the possibility to revert the image to a snapshot we have taken earlier.

To prevent the program from exiting normally, we intercept all calls to the `NtTerminateProcess` system service function (provided by the `ntdll.dll` library). This is done by checking whether the program counter of the emulated CPU is equal to the start address of the `NtTerminateProcess` function. Whenever the inspected process calls this function, we assume that it wishes to terminate. In this case, we can revert the program to a previous snapshot (in case unexplored paths are left).

Segmentation faults (i.e., illegal memory accesses) are another venue for program termination that we intercept. To this end, we hook the page fault handler and examine the state of the emulated CPU whenever a page fault occurs. If an invalid memory access is detected, the process is reverted to a stored snapshot. Interestingly, invalid memory accesses occur relatively frequently. The reason is that during path exploration, we often encounter checks that ensure that a pointer is not null. In order to explore the alternative path, the pointer is set to an arbitrary non-null value. When this value is later dereferenced, it very likely refers to an unmapped memory area, which results in an illegal access.

Often, we encounter the situation that malicious code does not terminate at all. For example, spreading routines are typically implemented as endless loops that do not stop scanning for vulnerable targets. In such cases, we cannot simply end the analysis, because we would fail to analyze other, potentially interesting paths. To overcome this problem, we set a timeout for each path that our system explores (currently, 20 seconds). Whenever a path is still executed when the timeout expires, our system waits until the process is in a safe state and then reverts it to a previous snapshot (until there are no more unexplored paths left). As a result, it is also not possible for an attacker to thwart our analysis by deliberating inserting code on unused execution paths that end in an endless loop.

4.3 Optimization

One construct that frequently occurs in programs are string comparisons. Usually, two strings are compared by performing a sequence of pairwise equality checks between corresponding characters in the two strings. This can lead to problems when one of the strings (or both) are labeled. Note that each character comparison operates on labeled arguments and thus, is a branching point. As a result, when a labeled string of n characters is compared with another string, we create n states. Each of the states $s_i : 0 \leq i \leq n$ represents the case in which the first i characters of both strings match, while the two characters with the offset $i + 1$ differ. For practical purposes, we typically do not need this detailed resolution for string comparisons. The reason is that most of the time, a program only distinguishes between the two cases in which both strings are either equal or not equal. To address this problem, we implemented a heuristics that attempts to recognize string comparisons. This is implemented by checking for situations in which the same compare instruction is executed repeatedly, and the arguments of this compare have addresses that increase by one on every iteration. When such a string comparison is encountered, we do not branch on every check. Instead, we explore one path where the first characters are immediately different, and a second one in which the two strings match. This optimization avoids the significant increase of the overall number of states that would have to be processed otherwise (often without yielding any additional information).

4.4 Limitations

In Section 4.1, we discussed our approach of never returning any allocated resource to the operating system. The goal was to avoid invalid handles that would result when a process first closes a handle and is then reset to a previous snapshot (in which this handle is still valid). Our approach works well in most cases. However, one has to consider situations in which a process creates external effects, e.g., when writing to a file or sending data over a network.

There are few problems when a program writes to a file. The reason is that the file pointer is stored in user memory, and thus, it is automatically reset to the previous value when the process is restored. Also, as mentioned previously, files are never closed. Unfortunately, the situation is not as easy while handling network traffic. Consider an application that opens a connection to a remote server and then exchanges some data (e.g., such as a bot connecting to an IRC server). When reverting to a previous state, the synchronization between the application and the server is lost. In particular, when the program first sends out some data, is later reset, and then sends out this data again, the remote server receives the data twice. Typically, this breaks protocol logic and leads to the termination of the connection. In our current implementation, we solve this problem as follows: All network system calls in which the program attempts to establish a connection or sends out data are intercepted and not relayed to the operating system. That is, for these calls, our system simply returns a success code without actually opening a connection or sending packets. Whenever the program attempts to read from the network, we simply return a string of random characters of the maximum length requested. The idea is that because the results of network reads are labeled, our multiple path exploration technique will later determine those strings that trigger certain actions (e.g., such as command strings sent to a bot).

Another limitation is the lack of support for signals and multi-threaded applications. Currently, we do not record signals that are delivered to a process. Thus, when a signal is raised, this only happens once. When the process is later reverted to a previous state, the signal is not resent. The lack of support for multi-threaded applications is not a problem *per se*. Creating a snapshot for the complete process works independently of the number of threads. However, to ensure deterministic behavior our system would have to ensure that threads are scheduled deterministically.

It might also be possible for specially-crafted malware programs to conceal some malicious behavior by preventing our system from exploring a certain path. To this end, the program has to ensure that a branch operation depends on a value that is related to other values via non-linear dependencies. For example, malicious code could deliberately apply non-linear operations such as `xor` to a certain value. When this value is later used in a conditional operation, our system would determine that it cannot be rewritten, as the related memory locations cannot be updated consistently. Thus, the alternative branch would not be explored. There are two ways to address this threat. First, we could replace the linear constraint solver by a system that can handle more complex relationships. For instance, by using a SAT solver, we could also track dependencies that involve bitwise operations. Unfortunately, when analyzing a binary that is specifically designed to withstand our analysis, our prototype will never be able to correctly invert all operations encountered. An example for that are one-way hash functions, for which our system cannot infer the original data from the hash value alone. Therefore, a second approach could be to relax the consistent update requirement. That is, we allow our system to explore paths by rewriting a memory location without being able to correctly modify all related input values. This approach leads to a higher coverage of the code analyzed, but we lose the knowledge of the input that is required to drive the execution down a certain path. In addition, the program could perform impossible operations (or simply crash) because of its inconsistent state. However, frequent occurrences of conditional jumps that cannot be resolved by our system could be interpreted as malicious. In this case, we could raise an appropriate warning and have a human analyst perform a deeper investigation.

Finally, specially-crafted malware programs could perform a denial-of-service attack against our analysis tool by performing many conditional branches on tainted data. This would force our system to create many states, which in turn leads to an exponential number of paths that have to be explored. One solution to this problem could be to define a distance metrics that can compare saved snapshots and merge sufficiently similar paths. Furthermore, we could also treat a sudden, abnormal explosion of states as a sign of malicious behavior.

5 Evaluation

In this section, we discuss the results that we obtained by running our malware analysis tool on a set of 308 real-world malicious code samples. These samples were collected in the wild by an anti-virus company and cover a wide range of malicious code classes such as viruses, worms, Trojan horses and bots. Note that we performed our experiments on all the samples we received, without any pre-selection.

The 308 samples in our test set belong to 92 distinct malware families (in certain cases, several different versions of a single family were included in the sample set). We classified these malware families using the free virus encyclopedia available at `viruslist.com`. Analyzing the results, we found that 42 malware families belong to the class of email-based worms (e.g., Netsky, Blaster). 30 families are classified as exploit-based worms (e.g., Blaster, Sasser). 10 malware families belong to the classic type of file infector viruses (e.g., Elkern, Kriz). The remaining 10 families are classified as Trojan horses and backdoors, typically combined with bot functionality (e.g., AceBot, AgoBot, or rBot). To understand how wide-spread our malware instances are, we checked Kaspersky's top-20 virus list for July 2006, the month that we received our test data. We found that our samples cover 18 entries on this list. Thus, we believe that we have assembled a comprehensive set of malicious code samples that cover a variety of malware classes that appear in the wild.

In a first step, our aim was to understand to which extent malware uses interesting input to perform control flow decisions. To this end, we had to define appropriate input sources. In our current prototype implementation, we consider the functions listed in Table 1 to provide interesting input. These functions were chosen primarily based on our previous experience with malware analysis (and also based on discussions with experienced malware analysts working in an anti-virus company). In the past, we have seen malicious code that uses the output provided by one of these functions to trigger actions. Also, note that adding additional input sources, if required, is trivial and is not a limitation of our approach. During the analysis, we label the return values of functions that check for the existence of an operating system resource. For functions that read from a resource (i.e., file, network, or timer), we label the complete buffer that is returned (by using one label for each byte).

Interesting input sources	
Check for Internet connectivity	20
Check for mutex object	116
Check for existence of files	79
Check for existence of registry entry	74
Read current time	134
Read from file	106
Read from network	134

Table 1. Number of samples that access tainted input sources.

After running our analysis on the complete set of 308 real-world malware samples, we observed that 229 of these samples used at least one of the tainted input sources we defined. The breakdown of the usage based on input is shown in Table 1. Of course, reading from a tainted source does not automatically imply that we can explore additional execution paths. For example, many samples copy their own executable file into a particular directory (e.g., the Windows system folder). In this case, our analysis observes that a file is read, and appropriately taints the input. However, the tainted bytes are simply written to another file, but not used for any conditional control flow decisions. Thus, there are no alternative program paths to explore.

Out of the 229 samples that access tainted sources, 172 use some of the tainted bytes for control flow decisions. In this case, our analysis is able to explore additional paths and extract behavior that would have remained undetected with a dynamic analysis only based on a single execution trace. In general, exploring multiple paths results in a more complete picture of the behavior of that code. However, it is unreasonable to expect that our analysis can always extract important additional knowledge about program behavior. For example, several malware instances implement a check that uses a mutex object to ensure that only a single program instance is running at the same time. That is, when the mutex is not found on the first execution path, the malware performs its normal malicious actions. When our system analyzes the alternative path (i.e., we pretend that the mutex exists), the program immediately exits. In such situations, we are only able to increase our knowledge by the fact that the presence of a specific mutex leads to immediate termination. Of course, there are many other cases in which the additional behavior is significant, and reveals hidden functionality not present in a single trace.

Table 2 shows the increase in coverage of the malicious code when we explore alternative branches. More precisely, this table shows the relative increase in the number of basic blocks that are analyzed by our system when considering alternative paths. The baseline for each sample is the number of basic blocks covered when simply running the sample in our analysis environment. For a small number of the samples (21 of 172), the newly detected code regions amount to less than 10% of the baseline. While it is possible that these 10% contain information that is relevant for an analyst, they are mostly due to the exploration of error paths that quickly lead to program termination. For the remaining samples (151 of 172), the increase in code coverage is above 10%, and often significantly larger. For example, the largest increase in code coverage that we observed was 3413.58%, when analyzing the `Win32.Plexus.B worm`. This was because this sample only executes its payload if its file name contains the string `upu.exe`. As this was not the case for the sample uploaded into our analysis system, the malware payload was only run in an alternative path. Anecdotal evidence of the usefulness of our system is provided in the following paragraphs, where we describe interesting behavior that was revealed by alternative paths. However, examining the quantitative results alone, it is evident that almost one half of the malware samples in the wild contain significant, hidden functionality that is missed by a simple analysis.

Relative increase	Number of samples
0 % - 10 %	21
10 % - 50 %	71
50 % - 200 %	37
$>$ 200 %	43

Table 2. Relative increase of code coverage.

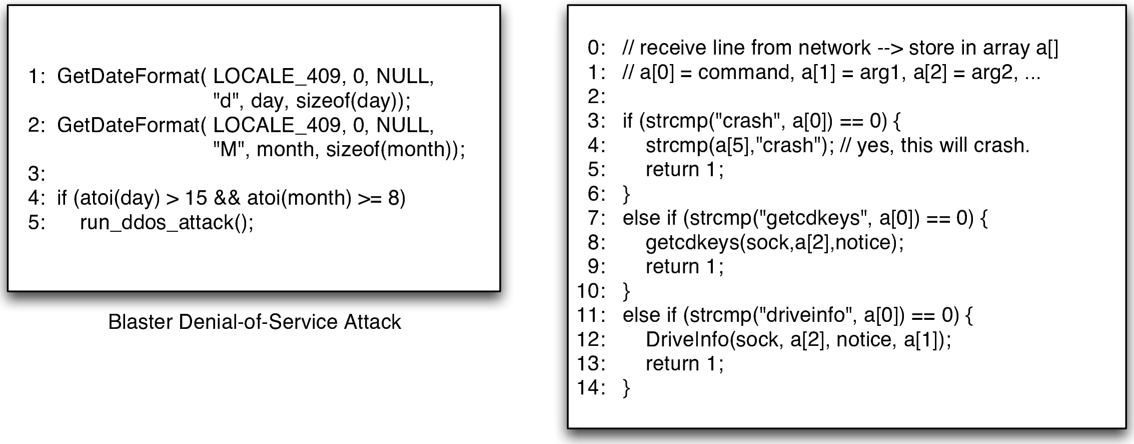
Behavioral analysis results. One interesting class of malicious behavior that can be detected effectively by our system is code that is only executed on a certain date (or in a time interval). As an example for this class, consider the `Blaster` code shown one the left side of Figure 4. This code launches a denial-of-service attack, but only after the 15^{th} of August. Suppose that `Blaster` is executed on the 1^{st} of January. In that case, a single execution trace would yield no indication of an attack. Using our system, however, a snapshot for the first check of the if-condition is created. After resetting the process, the day is rewritten to be larger than 15. Later, the system also passes the month check, updating the month variable to a value of 8 or larger. Hence, the multiple execution path exploration allows us to identify the fact that `Blaster` launches a denial-of-service attack, as well as the dates that it is launched.

Another interesting case in which our analysis can provide a more complete behavioral picture is when malware checks for the existence of a file to determine whether it was already installed. For example, the `Kriz` virus first checks for the existence of the file `KRIZED.TT6` in the system folder. When this file is not present, the virus simply copies itself into the system folder and terminates. Only when the file is already present, malicious behavior can be observed. In such cases, an analysis system that performs a single execution run would only be able to monitor the installation.

Finally, our system is well-suited to identify actions that are triggered by commands that are received over the network or read from a file. An important class of malware that can be controlled by remote commands are IRC (Internet Relay Chat) bots. When started, these programs usually connect to an IRC server, join a channel, and listen to the chat traffic for keywords that trigger certain actions. Modern IRC bots can typically understand more than 100 commands, making a manual analysis slow and tedious. Using our system, we can automate the process and determine, for each command, which behavior is triggered. In contrast, when running a bot in existing analysis tools, it is likely that no malicious actions will be seen, simply because the bot never receives any commands. The code on the right side of Figure 4 shows a fragment of the command loop of the bot `rxBot`. This code implements a series of if-statements that check a line received from the IRC server for the presence of certain keywords. When this code is analyzed, the result of the read from the network (that is, the content of array `a`) is labeled. Therefore, all calls to the `strcmp` function are treated as branching points, and we can extract the actions for one command on each different path.

Performance. The goal of our system is to provide a malware analyst with a detailed report on the behavior of an unknown sample. Thus, performance is not a primary requirement. Nevertheless, for some programs, a significant number of paths needs to be explored. Thus, the time and space requirements for saving and restoring states cannot be completely neglected.

Whenever our system creates a snapshot, it saves the complete active memory content of the process. In addition, the state contains information from the shadow mem-



```
GetDateFormat( LOCALE_409, 0, NULL,
               "d", day, sizeof(day));
GetDateFormat( LOCALE_409, 0, NULL,
               "M", month, sizeof(month));

if (atoi(day) > 15 && atoi(month) >= 8)
    run_ddos_attack();
```

Blaster Denial-of-Service Attack

```
// receive line from network --> store in array a[]
// a[0] = command, a[1] = arg1, a[2] = arg2, ...

if (strcmp("crash", a[0]) == 0) {
    strcmp(a[5],"crash"); // yes, this will crash.
    return 1;
}
else if (strcmp("getcdkeys", a[0]) == 0) {
    getcdkeys(sock,a[2],notice);
    return 1;
}
else if (strcmp("driveinfo", a[0]) == 0) {
    DriveInfo(sock, a[2], notice, a[1]);
    return 1;
}
```

rxBot Command Loop

Figure 4. `Blaster` and `rxBot` source code snippets.

ory and the constraint system. During our experiments, we determined that the size of a state was equal to about three times the amount of memory that a process has allocated. On average, the size of a state was about 3.5 MB, and it never exceeded 10 MB. The time needed to create or restore a snapshot was 4 milliseconds on average, with a small variance (on an Intel Pentium IV with 3.4 GHz and 2 GB RAM). As mentioned in Section 4.2, a timeout of 20 seconds was set for the exploration of each individual program path. In addition, we set a timeout of 100 seconds for the complete analysis run of each sample. This tight, additional time limit was introduced to be able to handle a large number of samples in case certain malware instances would create many paths. In our experiments, we observed that 58% of the malware programs finished before the timeout expired. The remaining 42% of the samples had unexplored paths left when the analysis process was terminated. As a result, by increasing the total timeout, we would expect to achieve an even larger increase in code coverage than that reported in the previous paragraphs. The trade-off is that it would take longer until results are available.

The size of a state could be significantly reduced if we exploited the fact that the majority of memory locations and entries in the shadow memory are 0. Also, we could attempt to create incremental snapshots that only store the difference between the current and previous states. In theory, the number of concurrently active states can be as high as the number of branching points encountered. However, we observed that this is typically not the case, and the number of concurrently active states during the experiments was lower. More precisely, our system used on average 31 concurrent states (the maximum was 469). Note that these numbers also represent the average and maximum depths of the search trees that we observed, as we use a depth-first search strategy. The *total* number of states were on average 32, with a maximum of 1,210. Given the number of concurrently active states, we deemed it not necessary to develop more sophisticated algorithms to create program snapshots. Moreover, in a synthetic benchmark, we verified that our system can handle more than thousand active states.

6 Related Work

Malicious code analysis. Analyzing malicious executables is not a new problem; consequently, a number of solutions already exist. These solutions can be divided into two groups: *static analysis* and *dynamic analysis* techniques. Static analysis is the process of analyzing a program's code without actually executing it. This approach has the advantage that one can cover the entire code and thus, possibly capture the complete program behavior, independent of any single path executed during run-time. In [8], a technique was introduced that uses model checking to identify parts of a program that implement a previously specified, malicious code template. This technique was later extended in [9], allowing more general code templates and using advanced static analysis techniques. In [21], a system was presented that uses static analysis to identify malicious behavior in kernel modules that indicate a rootkit. Finally, in [20], a behavioral-based approach was presented that relies heavily on static code analysis to detect Internet Explorer plug-ins that exhibit spyware-like behavior. The main weakness of static analysis is that the code analyzed may not necessarily be the code that is actually run. In particular, this is true for self-modifying programs that use polymorphic or metamorphic techniques [27]. Also, malware can draw from a wide range of obfuscation mechanisms [22, 30] that may make static analysis very difficult.

Because of the many ways in which code can be obfuscated and the fundamental limits in what can be decided statically, we firmly believe that dynamic analysis is a necessary complement to static detection techniques. In [4], a behavior-based approach was presented that aims to dynamically detect evasive malware by injecting user input into the system and monitoring the resulting actions. In addition, a number of approaches exist that directly analyze the code dynamically. Unfortunately, the support for dynamic code analysis is limited; often, it only consists of debuggers or disassemblers that aid a human analyst. Tools such as CWSandbox [29], the Norman SandBox [25], TTAnalyze [2], or Cobra [28] automatically record the actions performed by a code sample, but they only consider a single execution path and thus, might miss relevant behavior. To address this limitation and to capture a more comprehensive view of a program's behavior, we developed our approach to explore multiple execution paths.

A very recent work that addresses the detection of hidden, time-based triggers in malware is described in [13]. In their work, the authors attempt to automatically discover time-dependent behavior by setting the system time to different values. The problem is that time-based triggers can be missed when the system time is not set to the exact time that the malware expects. In our approach, we do not attempt to provide an environment such that trigger conditions are met, but explore multiple code paths independent of the environment. Thus, we have a better chance of finding hidden triggers. In addition, our approach is more comprehensive, as we can detect arbitrary triggers.

Finally, in their technical report [5], the authors present a system that is similar to ours in its goal to detect trigger-based malware behavior. The main differences are the system design, which is based on mixed execution of binary code using elements of symbolic execution, and a less comprehensive evaluation (on four malware samples).

Software testing. The goal of our work is to obtain a more complete picture of the behavior of a malicious code sample, together with the conditions under which certain actions are performed. This is analogous to software testing where one attempts to find inputs that trigger bugs.

A number of test input generation systems [6, 15, 16] were presented that analyze a program and attempt to find input that drives execution to a certain program point. The difference to our approach is that the emphasis of these systems is to reach a certain point, and not to explore the complete program behavior. Other tools were proposed that explore multiple paths of a program to detect implementation errors. For example, model checking tools [10, 17, 18] translate programs into finite state machines and then reason whether certain properties hold on these automata. The systems that are closest to our work are DART [14] and EXE [7]. Both systems use symbolic execution [19]. That is, certain inputs are expressed as symbolic variables, and the system explores in parallel both alternative execution paths when a conditional operation is found that uses this symbolic input. Similar to our approach, these systems can explore multiple execution paths that depend on interesting input. Also, the conditions under which certain paths are selected can be calculated (and are subsequently used to generate test cases). The main differences to our technique are the following. First, the goal of these systems is to explore programs for program bugs while our intent is to create comprehensive behavioral profiles of malicious code. Second, we do not have the possibility of using source code and operate directly on hostile (obfuscated) binaries. This leads to a significantly different implementation in which interesting inputs are dynamically tracked by taint propagation. Also, the problem we are addressing is complicated by the fact that we are not able to utilize built-in operating system mechanisms (e.g., fork) to explore alternative program paths. Hence, we require an infrastructure to save and restore snapshots of the program execution.

Speculative execution. In [24], a system was presented that uses process snapshots to implement speculative execution. In distributed files systems, processes typically have to wait until remote file system operations are completed before they can resume execution. With speculative execution, processes continue without waiting for remote responses, based on locally available data only. When it later turns out that the remote operation returns data that is different from the local one, the process is reset to its previously stored snapshot. The concept of snapshots used in speculative execution is similar to the one in our work. The difference is that we use snapshots as a means to explore alternative execution paths, which requires consistent memory updates.

7 Conclusions

In this paper, we presented a system to explore multiple execution paths of Windows executables. The goal is to obtain a more comprehensive overview of the actions that an unknown sample can perform. In addition, the tool automatically provides the information under which circumstances a malicious action is triggered.

Our system works by tracking how a program processes interesting input (e.g., the local time, file checks, reads from the network). In particular, we dynamically check for conditional branch instructions whose outcome depend on certain input values. When such an instruction is encountered, a snapshot of the current execution state is created. When the program later finishes along the first branch, we reset it to the previously saved state and modify the argument of the condition such that the other branch is taken. When performing this rewrite operation, it is important to consistently update all memory locations that are related to the argument value. This is necessary to prevent the program from executing invalid or impossible paths.

Our experiments demonstrate that, for a significant fraction of malware samples in our evaluation set, the system is indeed exploring multiple paths. In these cases, our knowledge about a program's behavior is extended compared to a system that observes a single run. We also show for a number of real-world malware samples that the actions that were discovered by our technique reveal important and relevant information about the behavior of the malicious code.

Acknowledgments

This work was supported by the Austrian Science Foundation (FWF) under grant P18157, the FIT-IT project Pathfinder, and the Secure Business Austria competence center. We would like to thank our shepherd Andrew Myers and the anonymous referees for their valuable comments.

References

[1] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach To Collect Malware. In *Recent Advances in Intrusion Detection (RAID)*, 2006.

[2] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[3] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference*, 2005.

[4] K. Borders, X. Zhao, and A. Prakash. Siren: Catching Evasive Malware (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[5] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards Automatically Identifying Trigger-based Behavior in Malware using Symbolic Execution and Binary Analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University, 2007.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Conference on Computer and Communication Security*, 2006.

[8] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium*, 2003.

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

[10] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *International Conference on Software Engineering (ICSE)*, 2000.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[12] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[13] J. Crandall, G. Wassermann, D. Oliveira, Z. Su, F. Wu, and F. Chong. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Conference on Architectural Support for Programming Languages and OS*, 2006.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, 2005.

[15] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ACM Symposium on Software Testing and Analysis*, 1998.

[16] N. Gupta, A. Mathur, and M. Soffa. Automated test data generation using an iterative relaxation method. In *Symposium on Foundations of Software Engineering (FSE)*, 1998.

[17] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *10th SPIN Workshop*, 2003.

[18] G. Holzmann. The model checker spin. *Software Engineering*, 23(5), 1997.

[19] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[20] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Usenix Security Symposium*, 2006.

[21] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Application Conference (ACSAC)*, 2004.

[22] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security*, 2003.

[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[24] E. Nightingale, P. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *20th Symposium on Operating Systems Principles (SOSP)*, 2005.

[25] Norman. Normal Sandbox. `http://sandbox.norman.no/`, 2006.

[26] Symantec. Internet Security Threat Report: Volume X. `http://www.symantec.com/enterprise/threatreport/index.jsp`, 2006.

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

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

[29] C. Willems. CWSandbox: Automatic Behaviour Analysis of Malware. `http://www.cwsandbox.org/`, 2006.

[30] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.