Effective *and* Efficient Malware Detection at the End Host

Clemens Kolbitsch[*]*, Paolo Milani Comparetti*[*]*, Christopher Kruegel*[‡]*, Engin Kirda*[§]*,*
Xiaoyong Zhou[†]*, and XiaoFeng Wang*[†]

[*] *Secure Systems Lab, TU Vienna*
{ck,pmilani}@seclab.tuwien.ac.at

[‡] *UC Santa Barbara*
chris@cs.ucsb.edu

[§] *Institute Eurecom, Sophia Antipolis*
kirda@eurecom.fr

[†] *Indiana University at Bloomington*
{zhou,xw7}@indiana.edu

Abstract

Malware is one of the most serious security threats on the Internet today. In fact, most Internet problems such as spam e-mails and denial of service attacks have malware as their underlying cause. That is, computers that are compromised with malware are often networked together to form botnets, and many attacks are launched using these malicious, attacker-controlled networks.

With the increasing significance of malware in Internet attacks, much research has concentrated on developing techniques to collect, study, and mitigate malicious code. Without doubt, it is important to collect and study malware found on the Internet. However, it is even more important to develop mitigation and detection techniques based on the insights gained from the analysis work. Unfortunately, current host-based detection approaches (i.e., anti-virus software) suffer from *ineffective* detection models. These models concentrate on the features of a specific malware instance, and are often easily evadable by obfuscation or polymorphism. Also, detectors that check for the presence of a sequence of system calls exhibited by a malware instance are often evadable by system call reordering. In order to address the shortcomings of ineffective models, several dynamic detection approaches have been proposed that aim to identify the behavior exhibited by a malware family. Although promising, these approaches are unfortunately *too slow* to be used as real-time detectors on the end host, and they often require cumbersome virtual machine technology.

In this paper, we propose a novel malware detection approach that is both *effective* and *efficient*, and thus, can be used to replace or complement traditional anti-virus software at the end host. Our approach first analyzes a malware program in a controlled environment to build a model that characterizes its behavior. Such models describe the information flows between the system calls essential to the malware's mission, and therefore, cannot be easily evaded by simple obfuscation or polymorphic techniques. Then, we extract the program slices responsible for such information flows. For detection, we execute these slices to match our models against the runtime behavior of an unknown program. Our experiments show that our approach can effectively detect running malicious code on an end user's host with a small overhead.

1 Introduction

Malicious code, or malware, is one of the most pressing security problems on the Internet. Today, millions of compromised web sites launch drive-by download exploits against vulnerable hosts [35]. As part of the exploit, the victim machine is typically used to download and execute malware programs. These programs are often bots that join forces and turn into a botnet. Botnets [15] are then used by miscreants to launch denial of service attacks, send spam mails, or host scam pages.

Given the malware threat and its prevalence, it is not surprising that a significant amount of previous research has focused on developing techniques to collect, study, and mitigate malicious code. For example, there have been studies that measure the size of botnets [37], the prevalence of malicious web sites [35], and the infestation of executables with spyware [31]. Also, a number of server-side [4, 43] and client-side honeypots [50] were introduced that allow analysts and researchers to gather malware samples in the wild. In addition, there exist tools that can execute unknown samples and monitor their behavior [6, 28, 53, 54]. Some tools [6, 53] provide reports that summarize the activities of unknown programs at the level of Windows API or system calls. Such reports can be evaluated to find clusters of samples that behave similarly [5, 7] or to classify the type of observed, malicious activity [39]. Other tools [54] incorporate data flow into the analysis, which results in a more comprehensive view of a program's activity in the form of taint graphs.

While it is important to collect and study malware, this is only a means to an end. In fact, it is crucial that

the insight obtained through malware analysis is translated into detection and mitigation capabilities that allow one to eliminate malicious code running on infected machines. Considerable research effort was dedicated to the extraction of network-based detection models. Such models are often manually-crafted signatures loaded into intrusion detection systems [33] or bot detectors [20]. Other models are generated automatically by finding common tokens in network streams produced by malware programs (typically, worms) [32, 41]. Finally, malware activity can be detected by spotting anomalous traffic. For example, several systems try to identify bots by looking for similar connection patterns [19, 38]. While network-based detectors are useful in practice, they suffer from a number of limitations. First, a malware program has many options to render network-based detection very difficult. The reason is that such detectors cannot observe the activity of a malicious program directly but have to rely on artifacts (the traffic) that this program produces. For example, encryption can be used to thwart content-based techniques, and blending attacks [17] can change the properties of network traffic to make it appear legitimate. Second, network-based detectors cannot identify malicious code that does not send or receive any traffic.

Host-based malware detectors have the advantage that they can observe the complete set of actions that a malware program performs. It is even possible to identify malicious code before it is executed at all. Unfortunately, current host-based detection approaches have significant shortcomings. An important problem is that many techniques rely on *ineffective* models. Ineffective models are models that do not capture intrinsic properties of a malicious program and its actions but merely pick up artifacts of a specific malware instance. As a result, they can be easily evaded. For example, traditional anti-virus (AV) programs mostly rely on file hashes and byte (or instruction) signatures [46]. Unfortunately, obfuscation techniques and code polymorphism make it straightforward to modify these features without changing the actual semantics (the behavior) of the program [11]. Another example are models that capture the sequence of system calls that a specific malware program executes. When these system calls are independent, it is easy to change their order or add irrelevant calls, thus invalidating the captured sequence.

In an effort to overcome the limitations of ineffective models, researchers have sought ways to capture the malicious activity that is characteristic of a malware program (or a family). On one hand, this has led to detectors [10, 13, 25] that use sophisticated static analysis to identify code that is semantically equivalent to a malware template. Since these techniques focus on the actual semantics of a program, it is not enough for a malware sample to use obfuscation and polymorphic techniques to alter its appearance. The problem with static techniques is that static binary analysis is difficult [30]. This difficulty is further exacerbated by runtime packing and self-modifying code. Moreover, the analysis is costly, and thus, not suitable for replacing AV scanners that need to quickly scan large numbers of files. Dynamic analysis is an alternative approach to model malware behavior. In particular, several systems [22, 54] rely on the tracking of dynamic data flows (tainting) to characterize malicious activity in a generic fashion. While detection results are promising, these systems incur a significant performance overhead. Also, a special infrastructure (virtual machine with shadow memory) is required to keep track of the taint information. As a result, static and dynamic analysis approaches are often employed in automated malware analysis environments (for example, at anti-virus companies or by security researchers), but they are too *inefficient* to be deployed as detectors on end hosts.

In this paper, we propose a malware detection approach that is both *effective* and *efficient*, and thus, can be used to replace or complement traditional AV software at the end host. For this, we first generate effective models that cannot be easily evaded by simple obfuscation or polymorphic techniques. More precisely, we execute a malware program in a controlled environment and observe its interactions with the operating system. Based on these observations, we generate fine-grained models that capture the characteristic, malicious behavior of this program. This analysis can be expensive, as it needs to be run only once for a group of similar (or related) malware executables. The key of the proposed approach is that our models can be efficiently matched against the runtime behavior of an unknown program. This allows us to detect malicious code that exhibits behavior that has been previously associated with the activity of a certain malware strain.
The main contributions of this paper are as follows:

- We automatically generate fine-grained (effective) models that capture detailed information about the behavior exhibited by instances of a malware family. These models are built by observing a malware sample in a controlled environment.
- We have developed a scanner that can efficiently match the activity of an unknown program against our behavior models. To achieve this, we track dependencies between system calls without requiring expensive taint analysis or special support at the end host.
- We present experimental evidence that demonstrates that our approach is feasible and usable in practice.

2 System Overview

The goal of our system is to effectively and efficiently detect malicious code at the end host. Moreover, the system should be general and not incorporate *a priori* knowledge about a particular malware class. Given the freedom that malware authors have when crafting malicious code, this is a challenging problem. To attack this problem, our system operates by generating detection models based on the observation of the execution of malware programs. That is, the system executes and monitors a malware program in a controlled analysis environment. Based on this observation, it extracts the behavior that characterizes the execution of this program. The behavior is then automatically translated into detection models that operate at the host level.

Our approach allows the system to quickly detect and eliminate novel malware variants. However, it is reactive in the sense that it must observe a certain, malicious behavior before it can properly respond. This introduces a small delay between the appearance of a new malware family and the availability of appropriate detection models. We believe that this is a trade-off that is necessary for a general system that aims to detect and mitigate malicious code with *a priori* unknown behavior. In some sense, the system can be compared to the human immune system, which also reacts to threats by first detecting intruders and then building appropriate antibodies. Also, it is important to recognize that it is *not* required to observe every malware instance before it can be detected. Instead, the proposed system abstracts (to some extent) program behavior from a single, observed execution trace. This allows the detection of all malware instances that implement similar functionality.

Modeling program behavior. To model the behavior of a program and its security-relevant activity, we rely on system calls. Since system calls capture the interactions of a program with its environment, we assume that any relevant security violation is visible as one or more unintended interactions.

Of course, a significant amount of research has focused on modeling legitimate program behavior by specifying permissible sequences of system calls [18, 48]. Unfortunately, these techniques cannot be directly applied to our problem. The reason is that malware authors have a large degree of freedom in rearranging the code to achieve their goals. For example, it is very easy to reorder independent system calls or to add irrelevant calls. Thus, we cannot represent suspicious activity as system call sequences that we have observed. Instead, a more flexible representation is needed. This representation must capture true relationships between system calls but allow independent calls to appear in any order. For this, we represent program behavior as a *behavior graph* where nodes are (interesting) system calls. An edge is introduced from a node x to node y when the system call associated with y uses as argument some output that is produced by system call x. That is, an edge represents a data dependency between system calls x and y. Moreover, we only focus on a subset of interesting system calls that can be used to carry out malicious activity.

At a conceptual level, the idea of monitoring a piece of malware and extracting a model for it bears some resemblance to previous signature generation systems [32, 41]. In both cases, malicious activity is recorded, and this activity is then used to generate detection models. In the case of signature generation systems, network packets sent by worms are compared to traffic from benign applications. The goal is to extract tokens that are unique to worm flows and, thus, can be used for network-based detection. At a closer look, however, the differences between previous work and our approach are significant. While signature generation systems extract specific, byte-level descriptions of malicious traffic (similar to virus scanners), the proposed approach targets the semantics of program executions. This requires different means to observe and model program behavior. Moreover, our techniques to identify malicious activity and then perform detection differ as well.

Making detection efficient. In principle, we can directly use the behavior graph to detect malicious activity at the end host. For this, we monitor the system calls that an unknown program issues and match these calls with nodes in the graph. When enough of the graph has been matched, we conclude that the running program exhibits behavior that is similar to previously-observed, malicious activity. At this point, the running process can be terminated and its previous, persistent modifications to the system can be undone.

Unfortunately, there is a problem with the previously sketched approach. The reason is that, for matching system calls with nodes in the behavior graph, we need to have information about data dependencies between the arguments and return values of these systems calls. Recall that an edge from node x to y indicates that there is a data flow from system call x to y. As a result, when observing x and y, it is not possible to declare a match with the behavior graph $x \rightarrow y$. Instead, we need to know whether y uses values that x has produced. Otherwise, independent system calls might trigger matches in the behavior graph, leading to an unacceptable high number of false positives.

Previous systems have proposed dynamic data flow tracking (tainting) to determine dependencies between system calls. However, tainting incurs a significant performance overhead and requires a special environ-

ment (typically, a virtual machine with shadow memory). Hence, taint-based systems are usually only deployed in analysis environments but not at end hosts. In this paper, we propose an approach that allows us to detect previously-seen data dependencies by monitoring only system calls and their arguments. This allows efficient identification of data flows without requiring expensive tainting and special environments (virtual machines).

Our key idea to determine whether there is a data flow between a pair of system calls x and y that is similar to a previously-observed data flow is as follows: Using the observed data flow, we extract those parts of the program (the instructions) that are responsible for reading the input and transforming it into the corresponding output (a kind of program slice [52]). Based on this program slice, we derive a symbolic expression that represents the semantics of the slice. In other words, we extract an expression that can essentially pre-compute the expected output, based on some input. In the simplest case, when the input is copied to the output, the symbolic expression captures the fact that the input value is identical to the output value. Of course, more complicated expressions are possible. In cases where it is not possible to determine a closed symbolic expression, we can use the program slice itself (i.e., the sequence of program instructions that transforms an input value into its corresponding output, according to the functionality of the program).

Given a program slice or the corresponding symbolic expression, an unknown program can be monitored. Whenever this program invokes a system call x, we extract the relevant arguments and return value. This value is then used as input to the slice or symbolic expression, computing the expected output. Later, whenever a system call y is invoked, we check its arguments. When the value of the system call argument is equal to the previously-computed, expected output, then the system has detected the data flow.

Using data flow information that is computed in the previously described fashion, we can increase the precision of matching observed system calls against the behavior graph. That is, we can make sure that a graph with a relationship $x \rightarrow y$ is matched only when we observe x and y, **and** there is a data flow between x and y that corresponds to the semantics of the malware program that is captured by this graph. As a result, we can perform more accurate detection and reduce the false positive rate.

3 System Details

In this section, we provide more details on the components of our system. In particular, we first discuss how we characterize program activity via behavior graphs. Then, we introduce our techniques to automatically extract such graphs from observing binaries. Finally, we present our approach to match the actions of an unknown binary to previously-generated behavior graphs.

3.1 Behavior Graphs: Specifying Program Activity

As a first step, we require a mechanism to describe the activity of programs. According to previous work [12], such a specification language for malicious behaviors has to satisfy three requirements: First, a specification must not constrain independent operations. The second requirement is that a specification must relate dependent operations. Third, the specification must only contain security-relevant operations.

The authors in [12] propose *malspecs* as a means to capture program behavior. A malicious specification (malspec) is a directed acyclic graph (DAG) with nodes labeled using system calls from an alphabet Σ and edges labeled using logic formulas in a logic $\mathcal{L}_{dep}$. Clearly, malspecs satisfy the first two requirements. That is, independent nodes (system calls) are not connected, while related operations are connected via a series of edges. The paper also mentions a function *IsTrivialComponent* that can identify and remove parts of the graph that are not security-relevant (to meet the third requirement).

For this work, we use a formalism called *behavior graphs*. Behavior graphs share similarities with malspecs. In particular, we also express program behavior as directed acyclic graphs where nodes represent system calls. However, we do not have unconstrained system call arguments, and the semantics of edges is somewhat different.

We define a system call $s \in \Sigma$ as a function that maps a set of input arguments $a_1, \ldots, a_n$ into a set of output values $o_1, \ldots, o_k$ For each input argument of a system call a_i, the behavior graph captures where the value of this argument is derived from. For this, we use a function $f_{a_i} \in F$. Before we discuss the nature of the functions in F in more detail, we first describe where a value for a system call can be derived from. A system call value can come from three possible sources (or a mix thereof): First, it can be derived from the output argument(s) of previous system calls. Second, it can be read from the process address space (typically, the initialized data section – the `bss` segment). Third, it can be produced by the immediate argument of a machine instruction.

As mentioned previously, a function is used to capture the input to a system call argument a_i. More precisely, the function f_{a_i} for an argument a_i is defined as $f_{a_i} : x_1, x_2, \ldots, x_n \rightarrow y$, where each x_i represents the output o_j of a previous system call. The values that are read from memory are part of the function body, represented by $l(addr)$. When the function is evaluated,

$l(\text{addr})$ returns the value at memory location *addr*. This technique is needed to ensure that values that are loaded from memory (for example, keys) are not constant in the specification, but read from the process under analysis. Of course, our approach implies that the memory addresses of key data structures do not change between (polymorphic) variants of a certain malware family. In fact, this premise is confirmed by a recent observation that data structures are stable between different samples that belong to the same malware class [14]. Finally, constant values produced by instructions (through immediate operands) are implicitly encoded in the function body. Consider the case in which a system call argument a_i is the constant value 0, for example, produced by a `push $0` instruction. Here, the corresponding function is a constant function with no arguments $f_{a_i} :\rightarrow 0$. Note that a function $f \in F$ can be expressed in two different forms: As a (symbolic) formula or as an algorithm (more precisely, as a sequence of machine instructions – this representation is used in case the relation is too complex for a mathematical expression).

Whenever an input argument a_i for system call y depends on the some output o_j produced by system call x, we introduce an edge from the node that corresponds to x, to the node that corresponds to y. Thus, edges encode dependencies (i.e., temporal relationships) between system calls.

Given the previous discussion, we can define behavior graphs G more formally as: $G = (V, E, F, \delta)$, where:

- V is the set of vertices, each representing a system call $s \in \Sigma$
- E is the set of edges, $E \subseteq V \times V$
- F is the set of functions $\bigcup f : x_1, x_2, \ldots, x_n \rightarrow y$, where each x_i is an output arguments o_j of system call $s \in \Sigma$
- δ, which assigns a function f_i to each system call argument a_i

Intuitively, a behavior graph encodes relationships between system calls. That is, the functions f_i for the arguments a_i of a system call s determine how these arguments depend on the outputs of previous calls, as well as program constants and memory values. Note that these functions allow one to *pre-compute* the expected arguments of a system call. Consider a behavior graph G where an input argument a of a system call s_t depends on the outputs of two previous calls s_p and s_q. Thus, there is a function f_a associated with a that has two inputs. Once we observe s_p and s_q, we can use the outputs o_p and o_q of these system calls and plug them into f_a. At this point, we know the expected value of a, assuming that the program execution follows the semantics encoded in the behavior graph. Thus, when we observe at a later point the invocation of s_t, we can check whether its actual argument value for a matches our precomputed value $f_a(o_p, o_q)$. If this is the case, we have high confidence that the program executes a system call whose input is related (depends on) the outputs of previous calls. This is the key idea of our proposed approach: We can identify relationships between system calls without tracking any information at the instruction-level during runtime. Instead, we rely solely on the analysis of system call arguments and the functions in the behavior graph that capture the semantics of the program.

3.2 Extracting Behavior Graphs

As mentioned in the previous section, we express program activity as behavior graphs. In this section, we describe how these behavior graphs can be automatically constructed by observing the execution of a program in a controlled environment.

Initial Behavior Graph

As a first step, an unknown malware program is executed in an extended version of Anubis [6, 7], our dynamic malware analysis environment. Anubis records all the disassembled instructions (and the system calls) that the binary under analysis executes. We call this sequence of instructions an *instruction log*. In addition, Anubis also extracts data dependencies using taint analysis. That is, the system taints (marks) each byte that is returned by a system call with a unique label. Then, we keep track of each labeled byte as the program execution progresses. This allows us to detect that the output (result) of one system call is used as an input argument for another, later system call.

While the instruction log and the taint labels provide rich information about the execution of the malware program, this information is not sufficient. Consider the case in which an instruction performs an indirect memory access. That is, the instruction *reads* a memory value from a location $\mathcal{L}$ whose address is given in a register or another memory location. In our later analysis, we need to know which instruction was the last one to *write* to this location $\mathcal{L}$. Unfortunately, looking at the disassembled instruction alone, this is not possible. Thus, to make the analysis easier in subsequent steps, we also maintain a *memory log*. This log stores, for each instruction that accesses memory, the locations that this instruction reads from and writes to.

Another problem is that the previously-sketched taint tracking approach only captures data dependencies. For

example, when data is written to a file that is previously read as part of a copy operation, our system would detect such a dependency. However, it does not consider control dependencies. To see why this might be relevant, consider that the amount of data written as part of the copy operation is determined by the result of a system call that returns the size of the file that is read. The file size returned by the system call might be used in a loop that controls how often a new block of data needs to be copied. While this file size has an indirect influence on the (number of) write operation, there is no data dependency. To capture indirect dependencies, our system needs to identify the scope of code blocks that are controlled by tainted data. The start of such code blocks is identified by checking for branch operations that use tainted data as arguments. To identify the end of the scope, we leverage a technique proposed by Zhang et al. [55]. More precisely, we employ their *no preprocessing without caching* algorithm to find convergence points in the instruction log that indicate that the different paths of a conditional statement or a loop have met, indicating the end of the (dynamic) scope. Within a tainted scope, the results of all instructions are marked with the label(s) used in the branch operation, similar to the approach presented in [22].

At this point, our analysis has gathered the complete log of all executed instructions. Moreover, operands of all instructions are marked with taint labels that indicate whether these operands have data or control dependencies on the output of previous system calls. Based on this information, we can construct an initial behavior graph. To this end, every system call is mapped into a node in the graph, labeled with the name of this system call. Then, an edge is introduced from node x to y when the output of call x produces a taint label that is used in any input argument for call y.

Figure 1 depicts a part of the behavior graph of the Netsky worm. In this graph, one can see the system calls that are invoked and the dependencies between them when Netsky creates a copy of itself. The worm first obtains the name of its executable by invoking the *GetModuleFileNameA* function. Then, it opens the file of its executable by using the *NtCreateFile* call. At the same time, it creates a new file in the Windows system directory (i.e., in C:\Windows) that it calls *AVProtect9x.exe*. Obviously, the aim is to fool the user into believing that this file is benign and to improve the chances of survival of the worm. Hence, if the file is discovered by chance, a typical user will probably think that it belongs to some anti-virus software. In the last step, the worm uses the *NtCreateSection* system call to create a virtual memory block with a handle to itself and starts reading its own code and making a copy of it into the *AVProtect9x.exe* file.

In this example, the behavior graph that we generate specifically contains the string *AVProtect9x.exe*. However, obviously, a virus writer might choose to use random names when creating a new file. In this case, our behavior graph would contain the system calls that are used to create this random name. Hence, the randomization routines that are used (e.g., checking the current time and appending a constant string to it) would be a part of the behavior specification.

Figure 2 shows an excerpt of the trace that we recorded for Netsky. This is part of the input that the behavior graph is built from. On Line 1, one can see that the worm obtains the name of executable of the current process (i.e., the name of its own file). Using this name, it opens the file on Line 3 and obtains a handle to it. On Line 5, a new file called *AVProtect9x.exe* is created, where the virus will copy its code to. On Lines 8 to 10, the worm reads its own program code, copying itself into the newly created file.

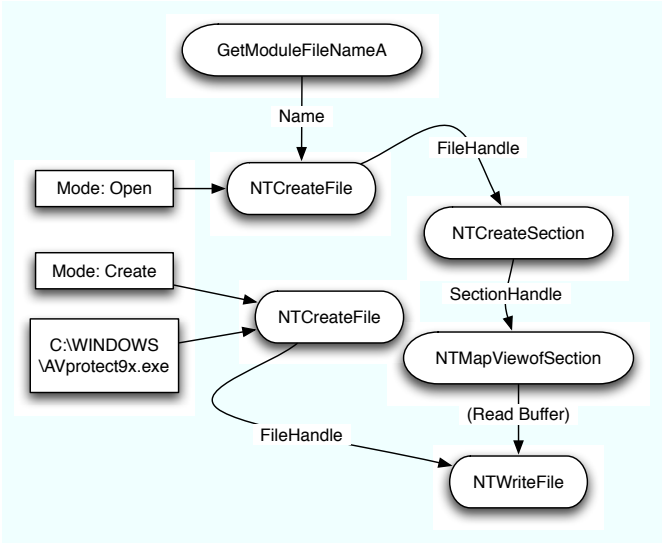

Figure 1: Partial behavior graph for Netsky.

Computing Argument Functions

In the next step, we have to compute the functions $f \in F$ that are associated with the arguments of system call nodes. That is, for each system call argument, we first have to identify the *sources* that can influence the value of this argument. Also, we need to determine how the values from the sources are manipulated to derive the argument value. For this, we make use of binary *program slicing*. Finally, we need to translate the sequence of instructions that compute the value of an argument (based on the values of the sources) into a function.

```
1  GetModuleFileNameA([out] lpFilename -> "C:\
       netsky.exe")
2  ...
3  NtCreateFile(Attr->ObjectName:"C:\netsky.exe",
       mode: open, [out] FileHandle -> A)
4  ...
5  NtCreateFile(Attr->ObjectName:"C:\WINDOWS\
       AVprotect9x.exe", mode: create, [out]
       FileHandle -> B)
6  ...
7  NtCreateSection(FileHandle: A, [out]
       SectionHandle -> C)
8  NtMapViewOfSection(SectionHandle: C,
       BaseAddress: 0x3b0000)
9  ...
10 NtWriteFile(FileHandle: B, Buffer: "MZ\90\00...
       ", Length: 16896)
11 ...
```

Figure 2: Excerpt of the observed trace for Netsky.

Program slicing. The goal of the program slicing process is to find all sources that directly or indirectly influence the value of an argument a of system call s, which is also called a *sink*. To this end, we first use the function signature for s to determine the type and the size of argument a. This allows us to determine the bytes that correspond to the sink a. Starting from these bytes, we use a standard dynamic slicing approach [2] to go backwards, looking for instructions that *define* one of these bytes. For each instruction found in this manner, we look at its operands and determine which values the instruction *uses*. For each value that is used, we locate the instruction that defines this value. This process is continued recursively. As mentioned previously, it is sometimes not sufficient to look at the instruction log alone to determine the instruction that has defined the value in a certain memory location. To handle these cases, we make use of the memory log, which helps us to find the previous write to a certain memory location.

Following def-use chains would only include instructions that are related to the sink via data dependencies. However, we also wish to include control flow dependencies into a slice. Recall from the previous subsection that our analysis computes tainted scopes (code the has a control flow dependency on a certain tainted value). Thus, when instructions are included into a slice that are within a tainted scope, the instructions that create this scope are also included, as well as the code that those instructions depend upon.

The recursive analysis chains increasingly add instructions to a slice. A chain terminates at one of two possible endpoints. One endpoint is the system call that produces a (tainted) value as output. For example, consider that we trace back the bytes that are written to a file (the argument that represents the write buffer). The analysis might determine that these bytes originate from a system call that reads the data from the network. That is, the values come from the "outside," and we cannot go back any further. Of course, we expect that there are edges from all sources to the sink that eventually uses the values produced by the sources. Another endpoint is reached when a value is produced as an immediate operand of an instruction or read from the statically initialized data segment. In the previous example, the bytes that are written to the file need not have been read previously. Instead, they might be originating from a string embedded in the program binary, and thus, coming from "within."

When the program slicer finishes for a system call argument a, it has marked all instructions that are involved in computing the value of a. That is, we have a subset (a slice) of the instruction log that "explains" *(1)* how the value for a was computed, and *(2)*, which sources were involved. As mentioned before, these sources can be constants produced by the immediate operands of instructions, values read from memory location *addr* (without any other instruction previously writing to this address), and the output of previous system calls.

Translating slices into functions. A program slice contains all the instructions that were involved in computing a specific value for a system call argument. However, this slice is not a program (a function) that can be directly run to compute the outputs for different inputs. A slice can (and typically does) contain a single machine instruction of the binary program more than once, often with different operands. For example, consider a loop that is executed multiple times. In this case, the instructions of the binary that make up the loop body appear multiple times in the slice. However, for our function, we would like to have code that represents the loop itself, not the unrolled version. This is because when a different input is given to the loop, it might execute a different number of times. Thus, it is important to represent the function as the actual loop code, not as an unrolled sequence of instruction.

To translate a slice into a self-contained program, we first mark all instructions in the binary that appear at least once in the slice. Note that our system handles packed binaries. That is, when a malware program is packed, we consider the instructions that it executes *after* the unpacking routine as the relevant binary code. All instructions that do not appear in the slice are replaced with no operation statements (nops). The input to this code depends on the sources of the slice. When a source is a constant, immediate operand, then this constant is directly included into the function. When the source is a read operation from a memory address *addr* that was not previously written by the program, we replace it with a special function that reads the value at *addr* when a program is analyzed. Finally, outputs of previous system calls are replaced with variables.

In principle, we could now run the code as a function, simply providing as input the output values that we observe from previous system calls. This would compute a result, which is the pre-computed (expected) input argument for the sink. Unfortunately, this is not that easy. The reason is that the instructions that make up the function are taken from a binary program. This binary is made up of procedures, and these procedures set up stack frames that allow them to access local variables via offsets to the base pointer (register *%ebp*) or the stack pointer (x86 register *%esp*). The problem is that operations that manipulate the base pointer or the stack pointer are often not part of the slice. As a result, they are also not part of the function code. Unfortunately, this means that local variable accesses do not behave as expected. To compensate for that, we have to go through the instruction log (and the program binary) and *fix the stack*. More precisely, we analyze the code and add appropriate instructions that manipulate the stack and, if needed, the frame pointer appropriately so that local variable accesses succeed. For this, some knowledge about compiler-specific mechanisms for handling procedures and stack frames is required. Currently, our prototype slicer is able to handle machine code generated from standard C and C++ code, as well as several human-written/optimized assembler code idioms that we encountered (for example, code that is compiled without the frame pointer).

Once the necessary code is added to fix the stack, we have a function (program) at our disposal that captures the *semantics* of that part of the program that computes a particular system call argument based on the results of previous calls. As mentioned before, this is useful, because it allows us to pre-compute the argument of a system call that we would expect to see when an unknown program exhibits behavior that conforms to our behavior graph.

Optimizing Functions

Once we have extracted a slice for a system call argument and translated it into a corresponding function (program), we could stop there. However, many functions implement a very simple behavior; they copy a value that is produced as output of a system call into the input argument of a subsequent call. For example, when a system call such as `NtOpenFile` produces an opaque handle, this handle is used as input by all subsequent system calls that operate on this file. Unfortunately, the chain of copy operations can grow quite long, involving memory accesses and stack manipulation. Thus, it would be beneficial to identify and simplify instruction sequences. Optimally, the complete sequence can be translated into a formula that allows us to directly compute the expected output based on the formula's inputs.

To simplify functions, we make use of symbolic execution. More precisely, we assign symbolic values to the input parameters of a function and use a symbolic execution engine developed previously [23]. Once the symbolic execution of the function has finished, we obtain a symbolic expression for the output. When the symbolic execution engine does not need to perform any approximations (e.g., widening in the case of loops), then we can replace the algorithmic representation of the slice with this symbolic expression. This allows us to significantly shorten the time it takes to evaluate functions, especially those that only move values around. For complex functions, we fall back to the explicit machine code representation.

3.3 Matching Behavior Graphs

For every malware program that we analyze in our controlled environment, we automatically generate a behavior graph. These graphs can then be used for detection at the end host. More precisely, for detection, we have developed a scanner that monitors the system call invocations (and arguments) of a program under analysis. The goal of the scanner is to efficiently determine whether this program exhibits activity that matches one of the behavior graphs. If such a match occurs, the program is considered malicious, and the process is terminated. We could also imagine a system that unrolls the persistent modifications that the program has performed. For this, we could leverage previous work [45] on safe execution environments.

In the following, we discuss how our scanner matches a stream of system call invocations (received from the program under analysis) against a behavior graph. The scanner is a user-mode process that runs with administrative privileges. It is supported by a small kernel-mode driver that captures system calls and arguments of processes that should be monitored. In the current design, we assume that the malware process is running under the normal account of a user, and thus, cannot subvert the kernel driver or attack the scanner. We believe that this assumption is reasonable because, for recent versions of Windows, Microsoft has made significant effort to have users run without root privileges. Also, processes that run executables downloaded from the Internet can be automatically started in a low-integrity mode. Interestingly, we have seen malware increasingly adapting to this new landscape, and a substantial fraction can now successfully execute as a normal user.

The basic approach of our matching algorithm is the following: First, we partition the nodes of a behavior graph into a set of *active* nodes and a set of *inactive*

nodes. The set of active nodes contains those nodes that have already been matched with system call(s) in the stream. Initially, all nodes are inactive.

When a new system call s arrives, the scanner visits all inactive nodes in the behavior graph that have the correct type. That is, when a system call `NtOpenFile` is seen, we examine all inactive nodes that correspond to an `NtOpenFile` call. For each of these nodes, we check whether all its parent nodes are active. A parent node for node N is a node that has an edge to N. When we find such a node, we further have to ensure that the system call has the "right" arguments. More precisely, we have to check all functions $f_i : 1 \leq i \leq k$ associated with the k input arguments of the system call s. However, for performance reasons, we do not do this immediately. Instead, we only check the *simple functions*. Simple functions are those for which a symbolic expression exists. Most often, these functions check for the equality of handles. The checks for *complex functions*, which are functions that represent dependencies as programs, are deferred and optimistically assumed to hold.

To check whether a (simple) function f_i holds, we use the output arguments of the parent node(s) of N. More precisely, we use the appropriate values associated with the parent node(s) of N as the input to f_i. When the result of f_i matches the input argument to system call s, then we have a match. When all arguments associated with simple functions match, then node N can be activated. Moreover, once s returns, the values of its output parameters are stored with node N. This is necessary because the output of s might be needed later as input for a function that checks the arguments of N's child nodes.

So far, we have only checked dependencies between system calls that are captured by simple functions. As a result, we might activate a node y as the child of x, although there exists a complex dependency between these two system calls that is *not* satisfied by the actual program execution. Of course, at one point, we have to check these complex relationships (functions) as well. This point is reached when an *interesting* node in the behavior graph is activated. Interesting nodes are nodes that are (a) associated with security-relevant system calls and (b) at the "bottom" of the behavior graph. With security-relevant system calls, we refer to all calls that write to the file system, the registry, or the network. In addition, system calls that start new processes or system services are also security-relevant. A node is at the "bottom" of the behavior graph when it has no outgoing edges.

When an interesting node is activated, we go back in the behavior graph and check all complex dependencies. That is, for each active node, we check all complex functions that are associated with its arguments (in a way that is similar to the case for simple functions, as outlined previously). When all complex functions hold, the node is marked as *confirmed*. If any of the complex functions associated with the input arguments of an active node N does not hold, our previous optimistic assumption has been invalidated. Thus, we deactivate N as well as all nodes in the subgraph rooted in N.

Intuitively, we use the concept of interesting nodes to capture the case in which a malware program has demonstrated a chain of activities that involve a series of system calls with non-trivial dependencies between them. Thus, we declare a match as soon as any interesting node has been confirmed. However, to avoid cases of overly generic behavior graphs, we only report a program as malware when the process of confirming an interesting node involves at least one complex dependency.

Since the confirmed activation of a single interesting node is enough to detect a malware sample, typically only a subset of the behavior graph of a malware sample is employed for detection. More precisely, each interesting node, together with all of its ancestor nodes and the dependencies between these nodes, can be used for detection independently. Each of these subgraphs is itself a behavior graph that describes a specific set of actions performed by a malware program (that is, a certain behavioral trait of this malware).

4 Evaluation

We claim that our system delivers effective detection with an acceptable performance overhead. In this section, we first analyze the detection capabilities of our system. Then, we examine the runtime impact of our prototype implementation. In the last section, we describe two examples of behavior graphs in more detail.

Name	Type
Allaple	Exploit-based worm
Bagle	Mass-mailing worm
Mytob	Mass-mailing worm
Agent	Trojan
Netsky	Mass-mailing worm
Mydoom	Mass-mailing worm

Table 1: Malware families used for evaluation.

4.1 Detection Effectiveness

To demonstrate that our system is effective in detecting malicious code, we first generated behavior graphs for six popular malware families. An overview of these families is provided in Table 1. These malware families were selected because they are very popular, both in our own malware data collection (which we obtained from

Name	Samples	Kaspersky variants	Our variants	Samples detected	Effectiveness
Allaple	50	2	1	50	1.00
Bagle	50	20	14	46	0.92
Mytob	50	32	12	47	0.94
Agent	50	20	2	41	0.82
Netsky	50	22	12	46	0.92
Mydoom	50	6	3	49	0.98
Total	300	102	44	279	0.93

Table 2: Training dataset.

Anubis [1]) and according to lists compiled by anti-virus vendors. Moreover, these families provide a good cross section of popular malware classes, such as mail-based worms, exploit-based worms, and a Trojan horse. Some of the families use code polymorphism to make it harder for signature-based scanners to detect them. For each malware family, we randomly selected 100 samples from our database. The selection was based on the labels produced by the Kaspersky anti-virus scanner and included different variants for each family. During the selection process, we discarded samples that, in our test environment, did not exhibit any interesting behavior. Specifically, we discarded samples that did not modify the file system, spawn new processes, or perform network communication. For the `Netsky` family, only 63 different samples were available in our dataset.

Detection capabilities. For each of our six malware families, we randomly selected 50 samples. These samples were then used for the extraction of behavior graphs. Table 2 provides some details on the training dataset. The "Kaspersky variants" column shows the number of different variants (labels) identified by the Kaspersky anti-virus scanner (these are variants such as `Netsky.k` or `Netsky.aa`). The "Our variants" column shows the number of different samples from which (different) behavior graphs had to be extracted before the training dataset was covered. Interestingly, as shown by the "Samples detected" column, it was not possible to extract behavior graphs for the entire training set. The reasons for this are twofold: First, some samples did not perform any interesting activity during behavior graph extraction (despite the fact that they did show relevant behavior during the initial selection process). Second, for some malware programs, our system was not able to extract valid behavior graphs. This is due to limitations of the current prototype that produced invalid slices (i.e., functions that simply crashed when executed).

To evaluate the detection effectiveness of our system, we used the behavior graphs extracted from the training dataset to perform detection on the remaining 263 samples (the test dataset). The results are shown in Table 3. It can be seen that some malware families, such as `Allaple` and `Mydoom`, can be detected very accurately. For others, the results appear worse. However, we have to consider that different malware variants may exhibit different behavior, so it may be unrealistic to expect that a behavior graph for one variant always matches samples belonging to another variant. This is further exacerbated by the fact that anti-virus software is not particularly good at classifying malware (a problem that has also been discussed in previous work [5]). As a result, the dataset likely contains mislabeled programs that belong to different malware families altogether. This was confirmed by manual inspection, which revealed that certain malware families (in particular, the `Agent` family) contain a large number of variants with widely varying behavior.

To confirm that different malware variants are indeed the root cause of the lower detection effectiveness, we then restricted our analysis to the 155 samples in the test dataset that belong to "known" variants. That is, we only considered those samples that belong to malware variants that are also present in the training dataset (according to Kaspersky labels). For this dataset, we obtain a detection effectiveness of 0.92. This is very similar to the result of 0.93 obtained on the training dataset. Conversely, if we restrict our analysis to the 108 samples that do *not* belong to a known variant, we obtain a detection effectiveness of only 0.23. While this value is significantly lower, it still demonstrates that our system is sometimes capable of detecting malware belonging to previously unknown variants. Together with the number of variants shown in Table 2, this indicates that our tool produces a behavior-based malware classification that is more general than that produced by an anti-virus scanner, and therefore, requires a smaller number of behavior graphs than signatures.

Name	Samples	Known variant samples	Samples detected	Effectiveness
Allaple	50	50	45	0.90
Bagle	50	26	30	0.60
Mytob	50	26	36	0.72
Agent	50	4	5	0.10
Netsky	13	5	7	0.54
Mydoom	50	44	45	0.90
Total	263	155	168	0.64

Table 3: Detection effectiveness.

False positives. In the next step, we attempted to evaluate the amount of false positives that our system would produce. For this, we installed a number of popular applications on our test machine, which runs Microsoft Windows XP and our scanner. More precisely, we used Internet Explorer, Firefox, Thunderbird, putty, and Notepad. For each of these applications, we went through a series of common use cases. For example, we surfed the web with IE and Firefox, sent a mail with Thunderbird (*including* an attachment), performed a remote ssh login with putty, and used notepad for writing and saving text. No false positives were raised in these tests. This was expected, since our models typically capture quite tightly the behavior of the individual malware families. However, if we omitted the checks for *complex functions* and assumed all complex dependencies in the behavior graph to hold, *all* of the above applications raised false positives. This shows that our tool's ability to capture arbitrary data-flow dependencies and verify them at runtime is essential for effective detection. It also indicates that, in general, system call information alone (without considering complex relationships between their arguments) might not be sufficient to distinguish between legitimate and malicious behavior.

In addition to the Windows applications mentioned previously, we also installed a number of tools for performance measurement, as discussed in the following section. While running the performance tests, we also did not experience any false positives.

4.2 System Efficiency

As every malware scanner, our detection mechanism stands and falls with the performance degradation it causes on a running system. To evaluate the performance impact of our detection mechanism, we used 7-zip, a well-known compression utility, Microsoft Internet Explorer, and Microsoft Visual Studio. We performed the tests on a single-core, 1.8 GHz Pentium 4 running Windows XP with 1 GB of RAM.

For the first test, we used a command line option for 7-zip that makes it run a simple benchmark. This reflects the case in which an application is mostly performing CPU-bound computation. In another test, 7-zip was used to compress a folder that contains 215 MB of data (6,859 files in 808 subfolders). This test represents a more mixed workload. The third test consisted of using 7-zip to archive three copies of this same folder, performing no compression. This is a purely IO-bound workload. The next test measures the number of pages per second that could be rendered in Internet Explorer. For this test, we used a local copy of a large (1.5MB) web page [3]. For the final test, we measured the time required to compile and build our scanner tool using Microsoft Visual Studio. The source code of this tool consists of 67 files and over 17,000 lines of code. For all tests, we first ran the benchmark on the unmodified operating system (to obtain a baseline). Then, we enabled the kernel driver that logs system call parameters, but did not enable any user-mode detection processing of this output. Finally, we also enabled our malware detector with the full set of 44 behavior graphs.

The results are summarized in Table 4. As can be seen, our tool has a very low overhead (below 5%) for CPU-bound benchmarks. Also, it performs well in the I/O-bound experiment (with less than 10% overhead). The worst performance occurs in the compilation benchmark, where the system incurs an overhead of 39.8%. It may seem surprising at first that our tool performs worse in this benchmark than in the IO-bound archive benchmark. However, during compilation, the scanned application is performing almost 5,000 system calls per second, while in the archive benchmark, this value is around 700. Since the amount of computation performed in user-mode by our scanner increases with the number of system calls, compilation is a worst-case scenario for our tool. Furthermore, the more varied workload in the compile benchmark causes more complex functions to be evaluated. The 39.8% overhead of the compile benchmark can further be broken down into 12.2% for the

Test	Baseline	Driver		Scanner	
		Score	Overhead	Score	Overhead
7-zip (benchmark)	114 sec	117 sec	2.3%	118 sec	2.4%
7-zip (compress)	318 sec	328 sec	3.1%	333 sec	4.7%
7-zip (archive)	213 sec	225 sec	6.2%	231 sec	8.4%
IE - Rendering	0.41 page/s	0.39 pages/s	4.4%	0.39 page/s	4.4%
Compile	104 sec	117 sec	12.2%	146 sec	39.8%

Table 4: Performance evaluation.

kernel driver, 16.7% for the evaluation of *complex functions*, and 10.9% for the remaining user-mode processing. Note that the high cost of complex function evaluation could be reduced by improving our symbolic execution engine, so that less complex functions need to be evaluated. Furthermore, our prototype implementation spawns a new process every time that the verification of complex dependencies is triggered, causing unnecessary overhead. Nevertheless, we feel that our prototype performs well for common tasks, and the current overhead allows the system to be used on (most) end user's hosts. Moreover, even in the worst case, the tool incurs significantly less overhead than systems that perform dynamic taint propagation (where the overhead is typically several times the baseline).

4.3 Examples of Behavior Graphs

To provide a better understanding of the type of behavior that is modeled by our system, we provide a short description of two behavior graphs extracted from variants of the Agent and Allaple malware families.

Agent.ffn.StartService. The `Agent.ffn` variant contains a resource section that stores chunks of binary data. During execution, the binary queries for one of these stored resources and processes its content with a simple, custom decryption routine. This routine uses a variant of XOR decryption with a key that changes as the decryption proceeds. In a later step, the decrypted data is used to overwrite the Windows system file `C:\WINDOWS\System32\drivers\ip6fw.sys`. Interestingly, rather than directly writing to the file, the malware opens the `\\.\C:` logical partition at the offset where the `ip6fw.sys` file is stored, and directly writes to that location. Finally, the malware restarts Windows XP's integrated IPv6 firewall service, effectively executing the previously decrypted code.

Figure 3 shows a simplified behavior graph that captures this behavior. The graph contains nine nodes, connected through ten dependencies: six simple dependencies representing the reuse of previously obtained object handles (annotated with the parameter name), and four complex dependencies. The complex dependency that captures the previously described decryption routine is indicated by a bold arrow in Figure 3. Here, the `LockResource` function provides the body of the encrypted resource section. The `NtQueryInformationFile` call provides information about the `ip6fw.sys` file. The `\\.\C:` logical partition is opened in the `NtCreateFile` node. Finally, the `NtWriteFile` system call overwrites the firewall service program with malicious code. The check of the complex dependency is triggered by the activation of the last node (bold in the figure).

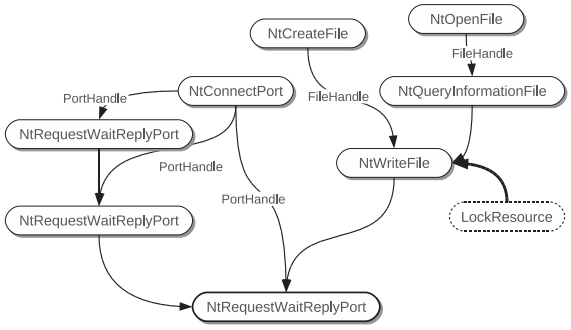

Figure 3: Behavior graph for `Agent.fnn`.

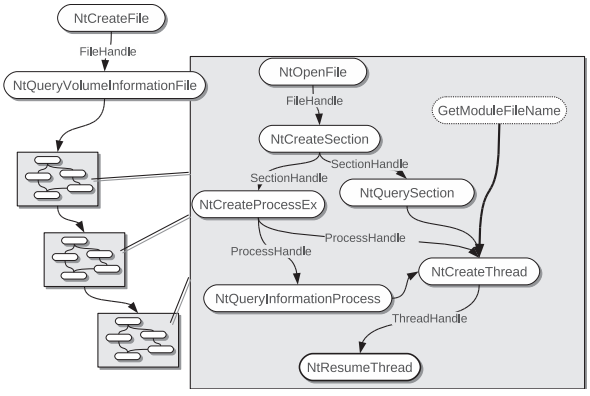

Figure 4: Behavior graph for `Allaple.b`.

Allaple.b.CreateProcess. Once started, the `Allaple.b` variant copies itself to the file `c:\WINDOWS\system32\urdvxc.exe`. Then, it invokes this executable various times with different command-line arguments. First, `urdvxc.exe /installservice` and `urdvxc.exe /start` are used to execute stealthily as a system service. In a second step, the malware tries to remove its traces by eliminating the original binary. This is done by calling `urdvxc.exe /uninstallservice patch:<binary>` (where `<binary>` is the name of the originally started program).

The graph shown in Figure 4 models part of this behavior. In the `NtCreateFile` node, the `urdvxc.exe` file is created. This file is then invoked three times with different arguments, resulting in three almost identical subgraphs. The box on the right-hand side of Figure 4 is an enlargement of one of these subgraphs. Here, the `NtCreateProcessEx` node represents the invocation of the `urdvxc.exe` program. The argument to the uninstall command (i.e., the name of the original binary) is supplied by the `GetModuleFileName` function to the `NtCreateThread` call. The last `NtResumeThread` system call triggers the verification of the complex dependencies.

5 Limitations

In this section, we discuss the limitations and possible attacks against our current system. Furthermore, we discuss possible solutions to address these limitations.

Evading signature generation. A main premise of our system is that we can observe a sample's malicious activities inside our system emulator. Furthermore, we require to find taint dependencies between data sources and the corresponding sinks. If a malware accomplishes to circumvent any of these two required steps, our system cannot generate system call signatures or find a starting point for the slicing process.

Note that our system is based on an unaccelerated version of Qemu. Since this is a system emulator (i.e., not a virtual machine), it implies that certain trivial means of detecting the virtual environment (e.g., such as Red Pill as described in [36]) are not applicable. Detecting a system emulator is an arms race against the accuracy of the emulator itself. Malware authors could also use delays, time-triggered behavior, or command and control mechanisms to try to prevent the malware from performing any malicious actions during our analysis. This is indeed the fundamental limitation of all dynamic approaches to the analysis of malicious code.

In maintaining taint label propagation, we implemented data and control dependent taint propagation and pursue a conservative approach to circumvent the loss of taint information as much as possible. Our results show that we are able to deal well with current malware. However, as soon as we observe threats in the wild targeting this feature of our system, we would need to adapt our approach.

Modifying the algorithm (input-output) behavior. Our system's main focus lies on the detection of data input-output relations and the malicious algorithm that the malware author has created (e.g., propagation technique). As soon as a malware writer decides to implement a new algorithm (e.g., using a different propagation approach), our slices would not be usable for the this new malware type. However, note that completely modifying the malicious algorithms contained in a program requires considerable manual work as this process is difficult to automate. As a result, our system raises the bar significantly for the malware author and makes this process more costly.

6 Related Work

There is a large number of previous work that studies the behavior [34, 37, 42] or the prevalence [31, 35] of different types of malware. Moreover, there are several systems [6, 47, 53, 54] that aid an analyst in understanding the actions that a malware program performs. Furthermore, techniques have been proposed to classify malware based on its behavior using a supervised [39] or unsupervised [5, 7] learning approach. In this paper, we propose a novel technique to effectively and efficiently identify malicious code on the end host. Thus, we focus on related work in the area of malware detection.

Network detection. One line of research focuses on the development of systems that detect malicious code at the network level. Most of these systems [20, 32, 33] use content-based signatures that specify tokens that are characteristic for certain malware families. Other approaches check for anomalous connections [19] or for network traffic that has suspicious properties [49]. While network-based detection has the advantage that a single sensor can monitor the traffic to multiple machines, there are a number of drawbacks. First, malware has significant freedom in altering network traffic, and thus, evade detection [17, 46]. Second, not all malware programs use the network to carry out their nefarious tasks. Third, even when an infected host is identified, additional action is necessary to terminate the malware program.

Static analysis. The traditional approach to detecting malware on the end host (which is implemented by antivirus software) is based on statically scanning executables for strings or instruction sequences that are characteristic for a malware sample [46]. These strings are typically extracted from the analysis of individual programs. The problem is that such strings are typically specific to the syntactic appearance of a certain malware instance. Using code polymorphism and obfuscation, malware programs can alter their appearance while keeping their behavior (functionality) unchanged [11, 46]. As a result, they can easily evade signature-based scanners.

As a reaction to the limitations of signature-based detection, researchers have proposed a number of higher-order properties to describe executables. The hope is that such properties capture intrinsic characteristics of a malware program and thus, are more difficult to disguise. One such property is the distribution of character n-grams in a file [26, 27]. This property can help to identify embedded malicious code in other files types, for example, Word documents. Another property is the control flow graph (CFG) of an application, which was used to detect polymorphic variants of malicious code instances that all share the same CFG structure [8, 24]. More sophisticated static analysis approaches rely on code templates or specifications that capture the malicious functionality of certain malware families. Here, symbolic execution [25], model checking [21], or techniques from compiler verification [13] are applied to recognize arbitrary code fragments that implement a specific function. The power of these techniques lies in the fact that a certain functionality can always be identified, independent of the specific machine instructions that express it.

Unfortunately, static analysis for malware detection faces a number of significant problems. One problem is that current malware programs rely heavily on run-time packing and self-modifying code [46]. Thus, the instruction present in the binary on disk are typically different than those executed at runtime. While generic unpackers [40] can sometimes help to obtain the actual instructions, binary analysis of obfuscated code is still very difficult [30]. Moreover, most advanced, static analysis approaches are very slow (in the order of minutes for one sample [13]). This makes them unsuitable for detection in real-world deployment scenarios.

Dynamic analysis. Dynamic analysis techniques detect malicious code by analyzing the execution of a program or the effects that this program has on the platform (operating system). An example of the latter category is Strider GhostBuster [51]. The tool compares the view of the system provided by a possible compromised OS to the view that is gathered when accessing the file system directly. This can detect the presence of certain types of rootkits that attempt to hide from the user by filtering the results of system calls.

The work that most closely relates to our own is Christodorescu et al. [12]. In [12], malware specifications (*malspecs*) are extracted by contrasting the behavior of a malware instance against a corpus of benign behaviors. Similarly to our behavior graphs, *malspecs* are DAGs where each node corresponds to a system call invocation. However, *malspecs* do not encode arbitrary data flow dependencies between system call parameters, and are therefore less specific than the behavior graphs described in this work. As discussed in Section 4, using behavior graphs for detection without verifying that complex dependencies hold would lead to an unacceptably large number of false positives.

In [22], a dynamic spyware detector system is presented that feeds browser events into Internet Explorer Browser Helper Objects (i.e., BHOs – IE plugins) and observes how the BHOs react to these browser events. An improved, tainting-based approach called Tquana is presented in [16]. In this system, memory tainting on a modified Qemu analysis environment is used to track the information that flows through a BHO. If the BHO collects sensitive data, writes this data to the disk, or sends this data over the network, the BHO is considered to be suspicious. In Panorama [54], whole-system taint analysis is performed to detect malicious code. The taint sources are typically devices such as a network card or the keyboard. In [44], bots are detected by using taint propagation to distinguish between behavior that is initiated locally and behavior that is triggered by remote commands over the network. In [29], malware is detected using a hierarchy of manually crafted behavior specifications. To obtain acceptable false positive rates, taint tracking is employed to determine whether a behavior was initiated by user input.

Although such approaches may be promising in terms of detection effectiveness, they require taint tracking on the end host to be able to perform detection. Tracking taint information across the execution of arbitrary, untrusted code typically requires emulation. This causes significant performance overhead, making such approaches unsuitable for deployment on end user's machines. In contrast, our system employs taint tracking when extracting a model of behavior from malicious code, but it does *not* require tainting to perform detection based on that model. Our system can, therefore, efficiently and effectively detect malware on the end user's machine.

Dialog rewriting. In their technical report [9], the authors present Rosetta, a system that extracts relationships (transformation functions) between input and output fields of network protocols. These relationships are

important to be able to compute the correct values of dynamic fields when performing protocol replay or NAT rewriting. To extract transformation functions, the authors use binary analysis, dynamic program slicing, and symbolic execution. Their approach resembles the techniques that we use for inferring complex dependencies between system call arguments. Of course, there are also significant differences between their work and ours. First, the problem domain and the goals of the two systems are entirely different. Second, we use symbolic execution as an optimization step and can execute functions (slices) even when no symbolic formula can be found.

7 Conclusion

Although a considerable amount of research effort has gone into malware analysis and detection, malicious code still remains an important threat on the Internet today. Unfortunately, the existing malware detection techniques have serious shortcomings as they are based on ineffective detection models. For example, signature-based techniques that are commonly used by anti-virus software can easily be bypassed using obfuscation or polymorphism, and system call-based approaches can often be evaded by system call reordering attacks. Furthermore, detection techniques that rely on dynamic analysis are often strong, but too slow and hence, inefficient to be used as real-time detectors on end user machines.

In this paper, we proposed a novel malware detection approach. Our approach is both *effective* and *efficient*, and thus, can be used to replace or complement traditional AV software at the end host. Our detection models cannot be easily evaded by simple obfuscation or polymorphic techniques as we try to distill the behavior of malware programs rather than their instance-specific characteristics. We generate these fine-grained models by executing the malware program in a controlled environment, monitoring and observing its interactions with the operating system. The malware detection then operates by matching the automatically-generated behavior models against the runtime behavior of unknown programs.

Acknowledgments

The authors would like to thank Christoph Karlberger for his invaluable programming effort and advice concerning the Windows kernel driver. This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368, and by the European Commission through project FP7-ICT-216026-WOMBAT. Xiaoyong Zhou and XiaoFeng Wang were supported in part by the National Science Foundation Cyber Trust program under Grant No. CNS-0716292.

References

[1] ANUBIS. http://anubis.iseclab.org, 2009.

[2] AGRAWAL, H., AND HORGAN, J. Dynamic Program Slicing. In *Conference on Programming Language Design and Implementation (PLDI)* (1990).

[3] B. COLLINS-SUSSMAN, B. W. FITZPATRICK AND C. M. PILATO. Version Control with Subversion. http://svnbook.red-bean.com/en/1.5/svn-book.html, 2008.

[4] BAECHER, P., KOETTER, M., HOLZ, T., DORNSEIF, M., AND FREILING, F. The Nepenthes Platform: An Efficient Approach To Collect Malware. In *Recent Advances in Intrusion Detection (RAID)* (2006).

[5] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z., JAHANIAN, F., AND NAZARIO, J. Automated Classification and Analysis of Internet Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2007).

[6] BAYER, U., KRUEGEL, C., AND KIRDA, E. TTAnalyze: A Tool for Analyzing Malware. In *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)* (2006).

[7] BAYER, U., MILANI COMPARETTI, P., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)* (2009).

[8] BRUSCHI, D., MARTIGNONI, L., AND MONGA, M. Detecting Self-Mutating Malware Using Control Flow Graph Matching. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2006).

[9] CABALLERO, J., AND SONG, D. Rosetta: Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NAT Rewriting. Tech. Rep. CMU-CyLab-07-014, Cylab, Carnegie Mellon University, 2007.

[10] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium* (2003).

[11] CHRISTODORESCU, M., AND JHA, S. Testing Malware Detectors. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2004).

[12] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining Specifications of Malicious Behavior. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2007).

[13] CHRISTODORESCU, M., JHA, S., SESHIA, S., SONG, D., AND BRYANT, R. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy* (2005).

[14] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. Digging For Data Structures . In *Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[15] DAGON, D., GU, G., LEE, C., AND LEE, W. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)* (2007).

[16] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic Spyware Analysis. In *Usenix Annual Technical Conference* (2007).

[17] FOGLA, P., SHARIF, M., PERDISCI, R., KOLESNIKOV, O., AND LEE, W. Polymorphic Blending Attacks. In *15th Usenix Security Symposium* (2006).

[18] FORREST, S., HOFMEYR, S., SOMAYAJI, A., AND LONGSTAFF, T. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy* (1996).

[19] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *17th Usenix Security Symposium* (2008).

[20] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *16th Usenix Security Symposium* (2007).

[21] KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. Detecting Malicious Code by Model Checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2005).

[22] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-based Spyware Detection. In *15th Usenix Security Symposium* (2006).

[23] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Automating Mimicry Attacks Using Static Binary Analysis. In *14th Usenix Security Symposium* (2005).

[24] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2005).

[25] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)* (2004).

[26] LI, W., STOLFO, S., STAVROU, A., ANDROULAKI, E., AND KEROMYTIS, A. A Study of Malcode-Bearing Documents. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2007).

[27] LI, W., WANG, K., STOLFO, S., AND HERZOG, B. Fileprints: Identifying File Types by N-Gram Analysis. In *IEEE Information Assurance Workshop* (2005).

[28] LI, Z., WANG, X., LIANG, Z., AND REITER, M. AGIS: Automatic Generation of Infection Signatures. In *Conference on Dependable Systems and Networks (DSN)* (2008).

[29] MARTIGNONI, L., STINSON, E., FREDRIKSON, M., JHA, S., AND MITCHELL, J. C. A Layered Architecture for Detecting Malicious Behaviors. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[30] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection . In *23rd Annual Computer Security Applications Conference (ACSAC)* (2007).

[31] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S., AND LEVY, H. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)* (2006).

[32] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy* (2005).

[33] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31* (1999).

[34] POLYCHRONAKIS, M., MAVROMMATIS, P., AND PROVOS, N. Ghost turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).

[35] PROVOS, N., MAVROMMATIS, P., RAJAB, M., AND MONROSE, F. All Your iFrames Point to Us. In *17th Usenix Security Symposium* (2008).

[36] RAFFETSEDER, T., KRUEGEL, C., AND KIRDA, E. Detecting System Emulators. In *Information Security Conference (ISC)* (2007).

[37] RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)* (2006).

[38] REITER, M., AND YEN, T. Traffic aggregation for malware detection. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2008).

[39] RIECK, K., HOLZ, T., WILLEMS, C., DUESSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2008).

[40] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Application Conference (ACSAC)* (2006).

[41] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm Fingerprinting. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[42] SMALL, S., MASON, J., MONROSE, F., PROVOS, N., AND STUBBLEFIELD, A. To Catch A Predator: A Natural Language Approach for Eliciting Malicious Payloads. In *17th Usenix Security Symposium* (2008).

[43] SPITZNER, L. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.

[44] STINSON, E., AND MITCHELL, J. C. Characterizing bots' remote control behavior. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2007).

[45] SUN, W., LIANG, Z., VENKATAKRISHNAN, V., AND SEKAR, R. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Network and Distributed Systems Symposium (NDSS)* (2005).

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

[47] VASUDEVAN, A., AND YERRABALLI, R. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *IEEE Symposium on Security and Privacy* (2006).

[48] WAGNER, D., AND DEAN, D. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy* (2001).

[49] WANG, K., AND STOLFO, S. Anomalous Payload-based Network Intrusion Detection. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2005).

[50] WANG, Y., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (2006).

[51] WANG, Y., BECK, D., VO, B., ROUSSEV, R., AND VERBOWSKI, C. Detecting Stealth Software with Strider Ghostbuster. In *Conference on Dependable Systems and Networks (DSN)* (2005).

[52] WEISER, M. Program Slicing. In *International Conference on Software Engineering (ICSE)* (1981).

[53] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy 2*, 2007 (5).

[54] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM Conference on Computer and Communication Security (CCS)* (2007).

[55] ZHANG, X., GUPTA, R., AND ZHANG, Y. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)* (2003).