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 [14] 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 [51] 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, 54, 55]. Some tools [6, 54] 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 [55] 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 [10]. 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 [9, 12, 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, 55] 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 [53]). 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 [11], 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 [11] 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 [13]. 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. [56]. 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.

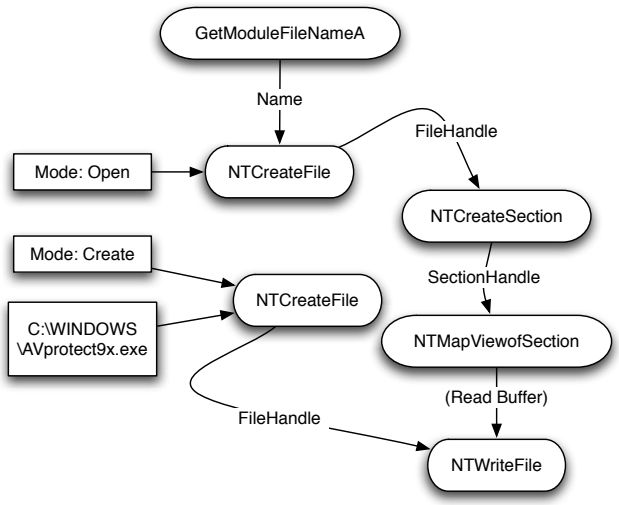

Figure 1: Partial behavior graph for Netsky.

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).

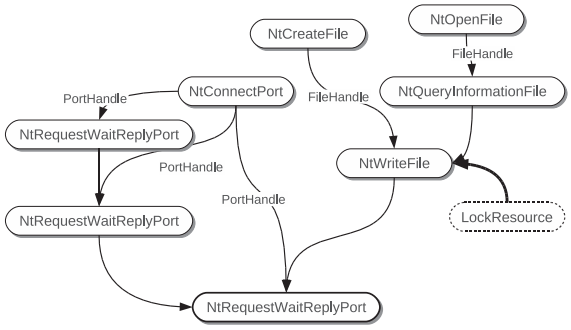

Figure 3: Behavior graph for `Agent.fnn`.

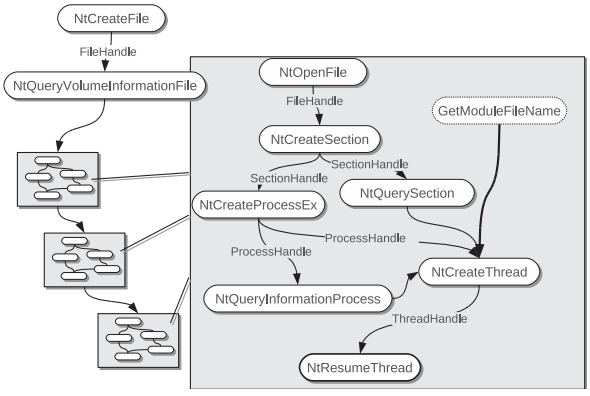

Figure 4: Behavior graph for `Allaple.b`.

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, 54, 55] 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 use content-based signatures that specify tokens that are characteristic for certain malware families. These signatures can either be crafted manually [20, 33] or automatically extracted by analyzing a pool of malicious payloads [32, 41, 49]. Other approaches check for anomalous connections or for network traffic that has suspicious properties. For example, there are systems [19, 38] that detect bots based on similar connections. Other tools [50] analyze network packets for the occurrence of anomalous statistical properties. 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 al-

tering 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 [10, 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 [12] 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 [12]). 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 [52]. 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. Another general, dynamic malware detection technique is based on the analysis of disk access patterns [16]. The basic idea is that malware activity might result in suspicious disk accesses that can be distinguished from normal program usage. The advantage of this approach is that it can be incorporated into the disk controller, and thus, is difficult to bypass. Unfortunately, it can only detect certain types of malware that scan for or modify large numbers of files.

The work that most closely relates to our own is Christodorescu et al. [11]. In [11], 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 [15]. 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 [55], 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 arbi-

trary, 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.

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

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

[11] 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).

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

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

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

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

[16] FELT, A., PAUL, N., EVANS, D., AND GURUMURTHI, S. Disk Level Malware Detection. In *Poster: 15th Usenix Security Symposium* (2006).

[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, H., JHA, S., AND GANAPATHY, V. NetSpy: Automatic Generation of Spyware Signatures for NIDS. In *Annual Computer Security Applications Conference (ACSAC)* (2006).

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

[51] 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).

[52] 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).

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

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

[55] 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).

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

Protecting Confidential Data on Personal Computers with Storage Capsules

Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash
University of Michigan
Ann Arbor, MI, 48109
{kborders, ericvw, billylau, aprakash}@umich.edu

Abstract

Protecting confidential information is a major concern for organizations and individuals alike, who stand to suffer huge losses if private data falls into the wrong hands. One of the primary threats to confidentiality is malicious software on personal computers, which is estimated to already reside on 100 to 150 million machines. Current security controls, such as firewalls, anti-virus software, and intrusion detection systems, are inadequate at preventing malware infection. This paper introduces Storages Capsules, a new approach for protecting confidential files on a personal computer. Storage Capsules are encrypted file containers that allow a compromised machine to securely view and edit sensitive files without malware being able to steal confidential data. The system achieves this goal by taking a checkpoint of the current system state and disabling device output before allowing access a Storage Capsule. Writes to the Storage Capsule are then sent to a trusted module. When the user is done editing files in the Storage Capsule, the system is restored to its original state and device output resumes normally. Finally, the trusted module declassifies the Storage Capsule by re-encrypting its contents, and exports it for storage in a low-integrity environment. This work presents the design, implementation, and evaluation of Storage Capsules, with a focus on exploring covert channels.

1. Introduction

Traditional methods for protecting confidential information rely on upholding system integrity. If a computer is safe from hackers and malicious software (malware), then so is its data. Ensuring integrity in today's interconnected world, however, is exceedingly difficult. Trusted computing platforms such as Terra [8] and trusted boot [26] try to provide this integrity by verifying software. Unfortunately, these platforms are rarely deployed in practice and most software continues to be unverified. More widely-applicable security tools, such as firewalls, intrusion detection systems, and anti-virus software, have been unable to combat malware, with 100 to 150 million infected machines running on the Internet today according to a recent estimate [34]. Security mechanisms for personal computers simply cannot rely on keeping high integrity. Storage Capsules address the need for access to confidential data from compromised personal computers.

There are some existing solutions for preserving confidentiality that do not rely on high integrity. One example is mandatory access control (MAC), which is used by Security-Enhanced Linux [23]. MAC can control the flow of sensitive data with policies that prevent entities that read confidential information from communicating over the network. This policy set achieves the goal of preventing leaks in the presence of malware. However, defining correct policies can be difficult, and they would prevent most useful applications from running properly. For example, documents saved by a word processor that has ever read secret data could not be sent as e-mail attachments. Another embodiment of the same principle can be seen in an "air gap" separated network where computers are physically disconnected from the outside world. Unplugging a compromised computer from the Internet will stop it from leaking information, but doing so greatly limits its utility. Both mandatory access control with strict outbound flow policies and air gap networks are rarely used outside of protecting classified information due to their severe impact on usability.

This paper introduces Storage Capsules, a new mechanism for protecting sensitive information on a local computer. The goal of Storage Capsules is to deliver the same level of security as a mandatory access control system for standard applications running on a commodity operating system. Storage Capsules meet this requirement by enforcing policies at a system-wide level using virtual machines. The user's system can also downgrade from high-secrecy to low-secrecy by reverting to a prior state using virtual machine snapshots. Finally, the system can obtain updated Storage Capsules from a declassification component after returning to low secrecy.

Storage Capsules are analogous to encrypted file containers from the user's perspective. When the user opens a Storage Capsule, a snapshot is taken of the current system state and device output is disabled. At this point, the system is considered to be in *secure mode*. When the user is finished editing files in a Storage Capsule, the system is reverted to its original state – dis-

carding all changes except those made to the Storage Capsule – and device output is re-enabled. The storage capsule is finally re-encrypted by a trusted component.

Storage Capsules guarantee protection against a compromised operating system or applications. Sensitive files are safe when they are encrypted *and* when being accessed by the user in plain text. The Capsule system prevents the OS from leaking information by erasing its entire state after it sees sensitive data. It also stops covert communication by fixing the Storage Capsule size and completely re-encrypting the data every time it is accessed by the OS. Our threat model assumes that the primary operating system can do anything at all to undermine the system. The threat model also assumes that the user, hardware, the virtual machine monitor (VMM), and an isolated secure virtual machine are trustworthy. The Capsule system protects against covert channels in the primary OS and Storage Capsules, as well as many (though not all) covert channels at lower layers (disk, CPU, etc.). One of the contributions of this paper is identifying and suggesting mitigation strategies for numerous covert channels that could potentially leak data from a high-secrecy VM to a low-secrecy VM that runs after it has terminated.

We evaluated the impact that Storage Capsules have on the user's workflow by measuring the latency of security level transitions and system performance during secure mode. We found that for a primary operating system with 512 MB of RAM, transitions to secure mode took about 4.5 seconds, while transitions out of secure mode took approximately 20 seconds. We also compared the performance of the Apache build benchmark in secure mode to that of a native machine, a plain virtual machine, and a virtual machine running an encryption utility. Overall, Storage Capsules added 38% overhead compared to a native machine, and only 5% compared to a VM with encryption software. The common workload for a Storage Capsule is expected to be much lighter than an Apache build. In many cases, it will add only a negligible overhead.

The main contribution of this work is a system that allows safe access to sensitive files from a normal operating system with standard applications. The Capsule system is able to switch modes within one OS rather than requiring separate operating systems or processes for different modes. This paper also makes contributions in the understanding of covert channels in such a system. In particular, it looks at how virtualization technology can create new covert channels and how previously explored covert channels behave differently when the threat model is a low-security virtual machine running after a high-security virtual machine.

It is important to keep in mind that Storage Capsules do not protect integrity. There are a number of attacks that they cannot prevent. If malicious software stops the user from ever entering secure mode by crashing, then the user might be coerced into accessing sensitive files without Storage Capsules. Furthermore, malware can manipulate data to present false information that tricks the user into doing something erroneously, such as placing a stock transaction. These attacks are beyond the scope of this paper.

The remainder of this paper is laid out as follows. Section 2 discusses related work. Section 3 gives an overview of the usage model, the threat model, and design alternatives. Section 4 outlines the system architecture. Section 5 describes the operation of Storage Capsules. Section 6 examines the effect of covert channels on Storage Capsules. Section 7 presents evaluation results. Finally, section 8 concludes and discusses future work.

2. Related Work

The Terra system [8] provides multiple security levels for virtual machines using trusted computing technology. Terra verifies each system component at startup using a trusted platform model (TPM) [29], similar to trusted boot [26]. However, Terra allows unverified code to run in low-security virtual machines. One could imagine a configuration of Terra in which the user's primary OS runs inside of a low-integrity machine, just like in the Capsule system. The user could have a separate secure VM for decrypting, editing, and encrypting files. Assuming that the secure VM always has high integrity, this approach would provide comparable security and usability benefits to Storage Capsules. However, Terra only ensures a secure VM's integrity at startup; it does not protect running software from exploitation. If this secure VM ever loads an encrypted file from an untrusted location, it is exposed to attack. All sources of sensitive data (e-mail contacts, web servers, etc.) would have to be verified and added to the trusted computing base (TCB), bloating its size and impacting both management overhead and security. Furthermore, the user would be unable to safely include data from untrusted sources, such as the internet, in sensitive files. The Capsule system imposes no such headaches; it can include low-integrity data in protected files, and only requires trust in local system components to guarantee confidentiality.

There has been extensive research on controlling the flow of sensitive information inside of a computer. Intra-process flow control techniques aim to verify that individual applications do not inadvertently leak confidential data [6, 22]. However, this does not stop mali-

cious software that has compromised a computer from stealing data at the operating system or file system level. Another approach for controlling information flow is at the process level with a mandatory access control (MAC) system like SELinux [23]. MAC involves enforcing access control policies on high-level objects (typically files, processes, etc.). However, defining correct policies can be quite difficult [15] even for a fixed set of applications. MAC would have a hard time protecting personal computers that download and install programs from the internet. Very few computers use mandatory access control currently, and it is not supported by Microsoft Windows, a popular operating system for personal computers. Storage Capsules employ a similar approach to MAC systems, but do so at a higher level of granularity (system-wide) using virtual machine technology. This allows Storage Capsules to provide more practical security for commodity operating systems without requiring modification.

There are a number of security products available for encrypting and protecting files on a local computer, including compression utilities [25, 35] and full disk encryption software [1, 7, 20, 31]. The goal of file encryption is to facilitate file transmission over an untrusted medium (e.g., an e-mail attachment), or protect against adversarial access to the storage device (e.g., a lost or stolen laptop). File encryption software does safeguard sensitive information while it is decrypted on the end host. Malicious software that has control of the end host can steal confidential data or encryption keys. Capsule also uses file encryption to allow storage in an untrusted location, but it maintains confidentiality while sensitive data is decrypted on the end host.

Storage Capsules rely on the virtual machine monitor as part of the trusted computing base. VMMs are commonly accepted as less complex and more secure than standard operating systems, with the Xen VMM having under 50,000 lines of code [36], compared to 5.7 million lines in the Linux 2.6 kernel [5]. These numbers are reinforced by actual vulnerability reports, with Xen 3.x only having 9 reports up to January 2009 [27], and the Linux 2.6.x kernel having 165 reports [28] in that same time period. VMMs are not invulnerable, but they have proven to be more robust than standard kernels.

Virtualization technology has many useful properties and features that make it a well-suited platform for Storage Capsules. Despite these advantages, Garfinkel et al. warn that virtualization has some shortcomings, especially when it comes to security [9]. Most importantly, have many branches and saved states makes patching and configuration much more difficult. A user might load an old snapshot that is vulnerable to infection by an Internet worm. The Capsule system does not suffer from these limitations because it is designed to have one primary VM with a fairly straight execution path. Transitions too and from secure mode are short-lived, and should have a minimal impact on patching and management tasks.

3. Overview

3.1 Storage Capsules from a User's Perspective

From the user's perspective, Storage Capsules are analogous to encrypted file containers provided by a program like TrueCrypt [31]. Basing the Capsule system off of an existing and popular program's usage model makes it easier to gain acceptance. The primary difference between Storage Capsules and traditional encryption software is that the system enters a secure mode before opening the Storage Capsule's contents. In this secure mode, network output is disabled and any changes that the user makes outside of the Storage Capsule will be lost. The user may still edit the Storage Capsule contents with his or her standard applications. When the user closes the Storage Capsule and exits secure mode, the system reverts to the state it was in before accessing sensitive data.

One motivating example for Storage Capsules is providing a secure journal. A person, call him Bob, may want to write a diary in which he expresses controversial political beliefs. Bob might regularly write in this journal, possibly pasting in news stories or contributions from others on the internet. Being a diligent user, Bob might store this document in an encrypted file container. Unfortunately, Bob is still completely vulnerable to spyware when he enters the decryption password and edits the document. Storage Capsules support the same usage model as normal encrypted file containers, but also deliver protection against spyware while the user is accessing sensitive data.

Storage Capsules have some limitations compared to encrypted file containers. These limitations are necessary to gain additional security. First, changes that the user makes outside of the encrypted Storage Capsule while it is open will not persist. This benefits security and privacy by eliminating all traces of activity while the container was open. Storage Capsules guarantee that the OS does not inadvertently hold information about sensitive files as described by Czeskis et al. for the case of TrueCrypt [4]. Unfortunately, any work from computational or network processes that may be running in the background will be lost. One way to remove this limitation would be to fork the primary virtual machine and

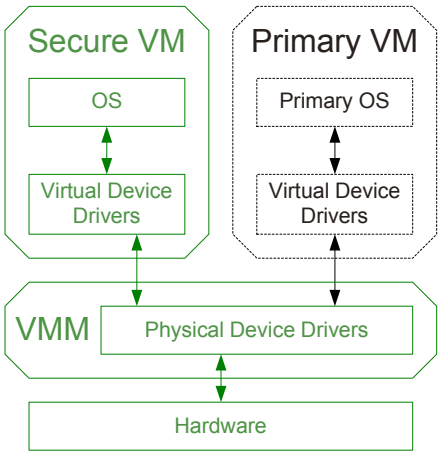

Figure 1. In the Storage Capsule architecture, the user's primary operating system runs in a virtual machine. The secure VM handles encryption and declassification. The dotted black line surrounding the primary VM indicates that it is *not* trusted. The other system components are trusted.

allow a copy of it to run in the background. Allowing low- and high-secrecy VMs to run at the same time, however, reduces security by opening up the door for a variety of covert channels.

3.2 Threat Model

Storage Capsules are designed to allow a compromised operating system to safely edit confidential information. However, some trusted components are necessary to provide security. Figure 1 shows the architecture of the Capsule system, with trusted components having solid lines and untrusted components having dotted lines. The user's primary operating system runs inside of a *primary VM*. Neither the applications, the drivers, nor the operating system are trusted in the primary VM; it can behave in any arbitrary manner. A *virtual machine monitor (VMM)* runs beneath the primary VM, and is responsible for mediating access to physical devices. The VMM is considered part of the trusted computing base (TCB). The Capsule system also relies on a *Secure VM* to save changes and re-encrypt Storage Capsules. This secure VM has only a minimal set of applications to service Storage Capsule requests, and has all other services blocked off with a firewall. The secure VM is also part of the TCB.

The user is also considered trustworthy in his or her intent. Presumably, the user has a password to decrypt each Storage Capsule and could do so using rogue software without going into secure mode and leak sensitive data. The user does not require full access to any trusted components, however. The main user interface is the primary VM, and the user should only interact with the Secure VM or VMM briefly using a limited UI. This prevents the user from inadvertently compromising a trusted component with bad input.

The threat model assumes that malicious software may try to communicate covertly within the primary VM. Storage Capsules are designed to prevent a compromised primary OS from saving data anywhere that will persist through a snapshot restoration. However, Storage Capsules do not guarantee that a malicious primary VM cannot store data somewhere in a trusted component, such as hardware or the VMM, in such a way that it can recover information after leaving secure mode. We discuss several of these covert channels in more depth later in the paper.

3.3 Designs that do not Satisfy Storage Capsule Goals

The first system design that would not meet the security goals laid out in our threat model is conventional file encryption software [1, 7, 20, 31]. Any information stored in an encrypted file would be safe from malicious software, or even a compromised operating system, while it is encrypted. However, as soon as the user decrypts a file, the operating system can do whatever it wants with the decrypted data.

Another design that would not meet the goals of Storage Capsules is the NetTop architecture [19]. With NetTop, a user has virtual machines with multiple security levels. One is for accessing high-secrecy information, and another for low-secrecy information, which may be connected to the internet. Depending on how policies are defined, NetTop either suffers from usability limitations or would have security problems. First assume that the high-secrecy VM must be able to read data from the low-secrecy VM to load files from external locations that are not part of the trusted computing base. Now, if the high-secrecy VM is prevented from writing anything back to the low-secrecy VM, then confidentiality is maintained. However, this prevents the user from making changes to a sensitive document, encrypting it, then sending it back out over a low-secrecy medium. This effectively makes everything read-only from the high-secrecy VM to the low-secrecy VM. The other alternative – letting the high-secrecy VM encrypt and declassify data – opens up a major security hole. Data that comes from the low-secrecy VM also might be malicious in nature. If the high-secrecy VM reads that information, its integrity – and the integrity of its encryption operations – may be compromised.

4. System Architecture

The Capsule system has two primary modes of operation: *normal mode* and *secure mode*. In normal mode, the computer behaves the same as it would without the Capsule system. The primary operating system has access to all devices and can communicate freely over the network. In secure mode, the primary OS is blocked from sending output to the external network or to devices that can store data. Furthermore, the primary operating system's state is saved prior to entering secure mode, and then restored when transitioning back to normal mode. This prevents malicious software running on the primary OS from leaking data from secure mode to normal mode.

The Capsule system utilizes virtual machine technology to isolate the primary OS in secure mode. Virtual machines also make it easy to save and restore system state when transitioning to or from secure mode. Figure 1 illustrates the architecture of the Capsule system. The first virtual machine, labeled Primary VM, contains the primary operating system. This VM is the equivalent of the user's original computer. It contains all of the user's applications, settings, and documents. This virtual machine may be infected with malicious software and is not considered trustworthy. The other virtual machine, labeled Secure VM, is responsible for managing access to Storage Capsules. The secure VM is trusted. The final component of the Capsule system shown in Figure 1 is the Virtual Machine Monitor (VMM). The VMM is responsible for translating each virtual device I/O request into a physical device request, and for governing virtual networks. As such, it can also block device I/O from virtual machines. The VMM has the power to start, stop, save, and restore entire virtual machines. Because it has full control of the computer, the VMM is part of the trusted computing base.

The Capsule system adds three components to the above architecture to facilitate secure access to Storage Capsules. The first is the *Capsule VMM module*, which runs as service inside of the VMM. The Capsule VMM module performs the following basic functions:

- Saves and restores snapshots of the primary VM
- Enables and disables device access by the primary VM
- Catches key escape sequences from the user
- Switches the UI between the primary VM and the secure VM

The Capsule VMM module executes operations as requested by the second component, the *Capsule server*, which runs inside of the secure VM. The Capsule server manages transitions between normal mode and secure mode. During secure mode, it also acts as a disk server, handling block-level read and write requests from the *Capsule viewer*, which runs in the primary VM. The Capsule server has dedicated interfaces for communicating with the Capsule viewer and with the Capsule VMM module. These interfaces are attached to separate virtual networks so that the viewer and VMM module cannot impersonate or communicate directly with each other.

The third component, the Capsule viewer, is an application that accesses Storage Capsules on the primary VM. When the user first loads or creates a new Storage Capsule, the viewer will import the file by sending it to the Capsule server. The user can subsequently open the Storage Capsule, at which point the viewer will ask the Capsule server to transition the system to secure mode. During secure mode, the viewer presents the contents of the Storage Capsule to the user as a new mounted partition. Block-level read and write requests made by the file system are forwarded by the viewer to the Capsule server, which encrypts and saves changes to the Storage Capsule. Finally, the Capsule viewer can retrieve the encrypted Storage Capsule by requesting an export from the Capsule server. The Capsule viewer is not trusted and may cause a denial-of-service at any time. However, the Capsule system is designed to prevent even a compromised viewer from leaking data from secure mode to normal mode.

5. Storage Capsule Operation

5.1 Storage Capsule File Format

A Storage Capsule is actually an encrypted partition that is mounted during secure mode. The Storage Capsule model is based on TrueCrypt [31] – a popular encrypted storage program. Like TrueCrypt, each new Storage Capsule is created with a fixed size. Storage Capsules employ XTS-AES – the same encryption scheme as TrueCrypt – which is the IEEE standard for data encryption [13]. In our implementation, the encryption key for each file is created by taking the SHA-512 hash of a user-supplied password. In a production system, it would be beneficial to employ other methods, such as hashing the password many times and adding a salt, to make attacks more difficult. The key could also come from a biometric reader (fingerprint reader, retina scanner, etc.), or be stored on a key storage device like a smart card. Storage Capsules operation does not depend on a particular key source.

With XTS-AES, a different tweak value is used during encryption for each data unit. A data unit can be one or more AES blocks. The Storage Capsule implementation

uses a single AES block for each data unit. In accordance with the IEEE 1619 standard [13], Storage Capsules use a random 128-bit starting tweak value that is incremented for each data unit. This starting tweak value is needed for decryption, so it is stored at the beginning of the file. Because knowledge of the tweak value does not weaken the encryption [18], it is stored in the clear.

5.2 Creating and Importing a Storage Capsule

The first step in securing data is creating a new Storage Capsule. The following tasks take place during the creation process:

1. The Capsule viewer solicits a Storage Capsule file name and size from the user.
2. The viewer makes a request to the Capsule server on the secure VM to create a new Storage Capsule.
3. The viewer asks the user to enter the secure key escape sequence that will be caught by a keyboard filter driver in the VMM. This deters spoofing by a compromised primary VM.
4. After receiving the escape sequence, the VMM module will give the secure VM control of the user interface.
    a. If the escape sequence is received unexpectedly (i.e. when the VMM has not received a request to wait for an escape sequence from the Capsule server), then the VMM module will still give control of the UI to the secure VM, but the secure VM will display a warning message saying that the user is *not* at a secure password entry page.
5. The Capsule server will ask the user to select a password, choose a random starting tweak value for encryption, and then format the encapsulated partition.
6. The Capsule server asks the VMM module to switch UI focus back to the primary VM.

After the creation process is complete, the Capsule server will send the viewer a file ID that it can store locally to link to the Storage Capsule on the server.

Loading a Storage Capsule from an external location requires fewer steps than creating a new Storage Capsule. If the viewer opens a Storage Capsule file that has been created elsewhere, it imports the file by sending it to the Capsule server. In exchange, the Capsule server sends the viewer a file ID that it can use as a link to the newly imported Storage Capsule. After a Storage Capsule has been loaded, the link on the primary VM looks the same regardless of whether the Capsule was created locally or imported from an external location.

5.3 Opening a Storage Capsule in Secure Mode

At this point, one or more Storage Capsules reside on the Capsule server, and have links to them on the primary VM. When the user opens a link with the Capsule viewer, it will begin the transition to secure mode, which consists of the following steps:

1. The Capsule viewer sends the Capsule server a message saying that the user wants to open a Storage Capsule, which includes the file ID from the link in the primary VM.
2. The Capsule viewer asks the user to enter the escape sequence that will be caught by the VMM module.
3. The VMM module receives the escape sequence and switches the UI focus to the secure VM. This prevents malware on the primary VM from spoofing a transition and stealing the file password.
    a. If the escape sequence is received unexpectedly, the secure VM still receives UI focus, but displays a warning message stating the system is *not* in secure mode.
4. The VMM module begins saving a snapshot of the primary VM in the background. Execution continues, but memory and disk data is copied to the snapshot on write.
5. The VMM module disables network and other device output.
6. The Capsule server asks the user to enter the file password.
7. The VMM module returns UI focus to the primary VM.
8. The Capsule server tells the viewer that the transition is complete and begins servicing disk I/O requests to the Storage Capsule.
9. The Capsule viewer mounts a local partition that uses the Capsule server for back-end disk block storage.

The above process ensures that the primary VM gains access to the Storage Capsule contents only after its initial state has been saved and the VMM has blocked device output. The exact set of devices blocked during secure mode is discussed more in the section on covert channels.

Depending on the source of the Storage Capsule encryption key, step 6 could be eliminated entirely. If the key was obtained from a smart card or other device, then the primary VM would retain UI focus throughout the entire transition, except in the case of an unexpected

escape sequence from the user. In this case, the secure VM must always take over the screen and warn the user that he or she is not in secure mode.

5.4 Storage Capsule Access in Secure Mode

When the Capsule system is running in secure mode, all reads and writes to the Storage Capsule are sent to the Capsule server. The server will encrypt and decrypt the data for each request as it is received, without performing any caching itself. The Capsule server instead relies on the caches within the primary VM and its own operating system to minimize unnecessary encryption and disk I/O. The disk cache in the primary VM sits above the driver that sends requests through to the Capsule server. On the secure VM, disk read and write requests from the Capsule server go through the local file system cache before they are sent to the disk. Later, we measure Storage Capsule disk performance during secure mode and demonstrate that is comparable to current encryption and virtualization software.

During secure mode, the VMM stores all writes to the primary VM's virtual disk in a file. This file contains differences with respect to the disk state at the time of the last snapshot operation (during the transition to secure mode). In the Capsule system, this difference file is stored on a partition that resides in main memory, commonly referred to as a RAM disk. Our implementation uses the ImDisk Virtual Disk Driver [14]. Storing the files on a RAM disk prevents potentially sensitive data that the primary VM writes to its virtual disk from reaching the physical disk. Although malicious access to the physical disk is not the focus of this paper's threat model, allowing confidential data to touch the disk increases the overall risk of leakage and the cost of hardware destruction.

The main downside to storing virtual disk modifications in memory is that the system requires more memory in secure mode. However, the size of primary disk modifications in secure mode should be small under normal circumstances because they are all temporary. Under normal circumstances, the user has no reason to store big files in a location where they will soon be deleted. If the change buffer does fill up due to a denial-of-service by malware or non-standard usage, then writes to the main virtual disk will fail until the system reverts to normal mode. If there turns out to be a legitimate need for large amounts of temporary storage in secure mode, then the change buffer could be stored on the physical disk and the VMM could use an encrypted file system for added protection.

5.5 Reverting to Normal Mode

Transitioning the Capsule system from secure mode back to normal mode is the most security-sensitive operation. Care must be taken to prevent leaks from secure mode back to normal mode insofar as practical for the desired level of security. A full discussion of the channels through which information might flow during this transition and countermeasures can be found in the next section. The Capsule system begins reverting to normal mode when the user enters a key escape sequence. Here, the escape sequence is not to prevent spoofing, but instead to reduce the primary VM's ability to leak data through a timing channel. After the user hits the escape sequence, the following steps take place:

1. The VMM module notifies the Capsule server of the pending transition, which in turn notifies the Capsule viewer.
2. The Capsule server waits up to 30 seconds for the primary VM to flush disk writes to the Storage Capsule. In our experiments, flushing always took less than one second, but uncommon workloads could make it take longer. We chose 30 seconds because it is the default maximum write-back delay for linux.
3. The secure VM reboots in order to flush any state that was affected by the primary VM. (This blocks some covert channels that are discussed in the next section.)
4. The VMM module halts the primary VM, and then reverts its state to the snapshot taken before entering secure mode and resumes execution.
5. The VMM module re-enables network and other device output for the primary VM.

After the Capsule system has reverted to normal mode, all changes that were made in the primary VM during secure mode, except those to the Storage Capsule, are lost. Also, when the Capsule viewer resumes executing in normal mode, it queries the Capsule to see what mode it is in (if the connection fails due to the reboot, normal mode is assumed). This is a similar mechanism to the return value from a fork operation. Without it, the Capsule viewer cannot tell whether secure mode is just beginning or the system has just reverted to normal mode, because both modes start from the same state.

5.6 Exporting Storage Capsules

After modifying a storage capsule, the user will probably want to back it up or transfer it to another person or computer at some point. Storage Capsules support this use case by providing an export operation. The Capsule viewer may request an export from the Capsule server at any time during normal mode. When the Capsule server

exports an encrypted Storage Capsule back to the primary VM, it is essential that malicious software can glean no information from the difference between the Storage Capsule at export compared to its contents at import. This should be the case even if malware has full control of the primary VM during secure mode and can manipulate the Storage Capsule contents in a chosen-plaintext attack.

For the Storage Capsule encryption scheme to be secure, the difference between the exported cipher-text and the original imported cipher-text must appear completely random. If the primary VM can change specific parts of the exported Storage Capsule, then it could leak data from secure mode. To combat this attack, the Capsule server re-encrypts the entire Storage Capsule using a new random 128-bit starting tweak value before each export. There is a small chance of two exports colliding. For any two Storage Capsules, each of size 2 GB (2^{27} encryption blocks), the chance of random 128-bit tweak values partially colliding would be approximately 1 in $2 * 2^{27} / 2^{128}$ or 1 in 2^{100}. Because of the birthday paradox, however, there will be a reasonable chance of a collision between a pair of exports after only 2^{50} exports. This number decreases further with the size of Storage Capsules. Running that many exports would still take an extremely long time (36 million years running 1 export / second). We believe that such an attack is unlikely to be an issue in reality, but could be mitigated if future tweaked encryption schemes support 256-bit tweak values.

5.7 Key Escape Sequences

During all Capsule operations, the primary VM and the Capsule viewer are not trusted. Some steps in the Capsule system's operation involve the viewer asking the user to enter a key escape sequence. If the primary VM becomes compromised, however, it could just skip asking the user to enter escape sequences and display a spoofed UI that looks like what would show up if the user did hit the escape sequence. This attack would steal the file decryption password while the system is still in normal mode. The defense against this attack is that the user should be accustomed to entering the escape sequence and therefore hit it anyway or notice anomalous behavior.

It is unclear how susceptible real users would be to spoofing attack that omits asking for an escape sequence. The success of such an attack is likely to depend on user education. Formally evaluating the usability of escape sequences in the Capsule system is future work. Another design alternative that may help if spoofing attacks are found to be a problem is reserving a secure area on the display. This area would always tell the user whether the system is in secure mode or the secure VM has control of the UI.

6. Covert Channel Analysis

The Storage Capsule system is designed to prevent any direct flow of information from secure mode to normal mode. However, there are a number of covert channels through which information may be able to persist during the transition from secure to normal mode. This section tries to answer the following questions about covert channels in the Capsule system as best as possible:

- Where can the primary virtual machine store information that it can retrieve after reverting to normal mode?
- What defenses might fully or partially mitigate these covert information channels?

We do not claim to expose all covert channels here, but list many channels that we have uncovered during our research. Likewise, the proposed mitigation strategies are not necessarily optimal, but represent possible approaches for reducing the bandwidth of covert channels. Measuring the maximum bandwidth of each covert channel requires extensive analysis and is beyond the scope of this paper. There has been a great deal of research on measuring the bandwidth of covert channels [2, 16, 21, 24, 30, 33], which could be applied to calculate the severity of covert channels in the Capsule system in future work.

The covert channels discussed in this section can be divided into five categories:

1. Primary OS and Capsule – Specific to Storage Capsule design
2. External Devices – Includes floppy, CD-ROM, USB, SCSI, etc.
3. External Network – Changes during secure mode that affect responsiveness to external connections
4. VMM – Arising from virtual machine monitor implementation, includes memory mapping and virtual devices
5. Core Hardware – Includes CPU and disk drives.

The Capsule system prevents most covert channels in the first three categories. It can use the VMM to mediate the primary virtual machine's device access and completely erase the primary VM's state when reverting to normal mode. The Capsule system also works to prevent timing channels when switching between modes of operation, and does respond to external network traffic while in secure mode.

Storage Capsules do not necessarily protect against covert channels in the last two categories. There has been a lot of work on identifying, measuring, and mitigating covert channels in core hardware for traditional MLS systems [16, 17, 21, 30]. Similar methods for measuring and mitigating those core channels could be applied to Storage Capsules. Covert channels arising from virtualization technology have not received much attention. This research hopes to highlight some of the key mechanisms in a VMM that can facilitate covert communication. The remainder of this section explores covert channels in each of these categories, including mitigation strategies and their required trade-offs.

6.1 Primary OS and Storage Capsule Channels

In addition to covert channels that one might find in other systems, the design of Storage Capsules creates opportunities for covert communication. This section focuses on the interactions between the primary VM and the secure VM. The primary VM should not be able to affect the secure VM's state during secure mode in a way that it can detect during normal mode. Here we examine three attack vectors that a compromised primary VM might use to steal data, and corresponding counter-measures.

Transition Timing

When the VMM restores the primary virtual machine from a snapshot, the restored state contains the exact same information as prior to entering secure mode. Following the restoration, the primary VM can obtain additional information by fetching the system time, which tells it when the system returned to normal mode. If the primary VM can manipulate the transition timing from secure mode, then it can leak data.

The Capsule system prevents direct leaks through transition timing by requiring a secure key escape sequence to switch back to normal mode. However, the primary VM may still be able to affect transition timing depending on how long the Capsule server waits for the primary VM to flush disk writes to the Storage Capsule upon leaving secure mode. One option that would preclude manipulation of transition timing would be for the Capsule server to wait a fixed time-period for disk writes to complete. This would be the most secure approach. Unfortunately, this strategy would either impact usability or threaten correctness. If the server were to stall 30 seconds – the default write-back delay for Linux – then the transition time would be unnecessarily long in most cases. On the other hand, if it only waited a few seconds, then some delayed writes might fail and corrupt the Storage Capsule.

For our implementation of the Capsule system, we chose a transition timing policy that sacrifices a few bits of potential information leakage per transition, but is more usable. The Capsule server will wait for the primary VM to flush Storage Capsule writes for up to 30 seconds. If the primary VM finishes the disk flush in less than 30 seconds, the server will immediately proceed with restoration. This opens up the possibility of the primary VM stalling a precise length of time after flushing writes so that it can read back timing information after returning to normal mode. The maximum bandwidth of this timing channel is somewhat limited, however, because the snapshot restoration adds noise to the transition time. Depending on how much memory the primary VM has, the amount of time between exiting secure mode and resuming in normal mode can vary by several seconds. By notifying the server at a specific offset from the most recent half-minute, the primary VM will only be able to communicate a few bits of data. One possible counter-measure to this type of timing channel would be to deliberately skew the clock during secure mode so the primary VM cannot accurately judge system time.

Storage Capsule Contents

The ideal place for a compromised primary VM to store data would be in the Storage Capsule itself. It is essential that the Capsule server cut off this channel. First, Storage Capsules must have a fixed size. This imposes a small usability limitation, which we believe is reasonable given that other popular systems like TrueCrypt [31] fix the size of encrypted file containers. Enforcing the next constraint required to cut off storage channels is slightly more complicated. No matter what changes the primary VM makes to the Storage Capsule in secure mode, it must not be able to deduce what has been changed after the Capsule server exports the Storage Capsule. As discussed earlier, XTS-AES encryption with a different tweak value for each export satisfies this requirement. Whether the primary VM changes every single byte or does not touch anything, the resulting exported Storage Capsule will be random with respect to its original contents.

Social Engineering Attacks

If the primary virtual machine cannot find a way to leak data directly, then it can resort to influencing user behavior. The most straightforward example of a social engineering attack would be for the primary VM to deny service to the user by crashing at a specific time, and then measuring transition time back to normal mode. There is a pretty good chance that the user would respond to a crash by switching back to normal mode immediately, especially if the system is prone to crash-

ing under normal circumstances. In this case, the user may not even realize that an attack is taking place. Another attack that is higher-bandwidth, but perhaps more suspicious, would be for the primary VM to display a message in secure mode that asks the user to perform a task that leaks information. For example, a message could read "Automatic update failed, please open the update dialog and enter last scan time '4:52 PM' when internet connectivity is restored." Users who do not understand covert channels could easily fall victim to this attack. In general, social engineering is difficult to prevent. The Capsule system currently does not include any counter-measures to social engineering. In a real deployment, the best method of fighting covert channels would be to properly educate the users.

6.2 External Device Channels

Any device that is connected to a computer could potentially store information. Fortunately, most devices in a virtual machine are virtual devices, including the keyboard, mouse, network card, display, and disk. In a traditional system, two processes that have access to the keyboard could leak data through the caps-, num-, and scroll-lock state. The VMware VMM resets this device state when reverting to a snapshot, so a virtual machine cannot use it for leaking data. We did not test virtualization software other than VMware to see how it resets virtual device state.

Some optional devices may be available to virtual machines. These include floppy drives, CD-ROM drives, sound adapters, parallel ports, serial ports, SCSI devices, and USB devices. In general, there is no way of stopping a VM that is allowed to access these devices from leaking data. Even devices that appear to be read-only, such as a CD-ROM drive, may be able to store information. A VM could eject the drive or position the laser lens in a particular spot right before switching back to normal mode. While these channels would be easy to mitigate by adding noise, the problem worsens when considering a generic bus like USB. A USB device could store anything or be anything, including a disk drive. One could allow access to truly read-only devices, but each device would have to be examined on an individual basis to uncover covert channels. The Capsule system prevents these covert channels because the primary VM is not given access to external devices. If the primary VM needs access to external devices, then they would have to be disabled during secure mode.

6.3 External Network Channels

In addition to channels from the Primary VM in secure mode to normal mode, it is also important to consider channels between the Storage Capsule system and external machines during secure mode. If malware can utilize so many resources that it affects how responsive the VMM is to external queries (such as pings), then it can leak data to a colluding external computer.

The best way to mitigate external network channels is for the VMM to immediately drop all incoming packets with a firewall, not even responding with reset packets for failed connections. If the VMM does not require any connections during secure mode, which it did not for our implementation, then this is the easiest and most effective approach.

6.4 Virtual Machine Monitor Channels

In a virtualization system, everything is governed by the virtual machine monitor, including memory mapping, device I/O, networking, and snapshot saving/restoration. The VMM's behavior can potentially open up new covert channels that are not present in a standard operating system. These covert channels are implementation-dependent and may or may not be present in different VMMs. This section serves as a starting point for thinking about covert channels in virtual machine monitors.

Memory Paging

Virtual machines are presented with a virtual view of their physical memory. From a VM's perspective, it has access to a contiguous "physical" memory segment with a fixed size. When a VM references its memory, the VMM takes care of mapping that reference to a real physical page, which is commonly called a machine page. There are a few different ways that a VMM can implement this mapping. First, it could directly pin all of the virtual machine's physical pages to machine pages. If the VMM uses this strategy, and it keeps the page mapping constant during secure mode and after restoration, then there is no way for a virtual machine to affect physical memory layout. However, this fixed mapping strategy is not always the most efficient way to manage memory.

Prior research describes resource management strategies in which the VMM may over-commit memory to virtual machines and page some of the VM's "physical" memory out to disk [11, 32]. If the VMM employs this strategy, then a virtual machine can affect the VMM's page table by touching different pages within its address space. The residual effects of page table manipulation may be visible to a VM after a snapshot restoration, unless the VMM first pages in all of the VM's memory.

A snapshot restoration should page in all of a VM's memory in most cases. But, if it is a "background" restoration, then accessing a memory location that has not been loaded from the snapshot yet *and* is paged out to disk might incur two page faults instead of one, which can be measured and may leak information.

VMware ESX server employs a number of tricks that allow it to over-commit memory to virtual machines [32]. Each of the mechanisms described by Waldspurger allow efficient server consolidation, but also create an opportunity for covert channels. The first mechanism he describes is *ballooning*. Guest virtual machines contain balloon drivers that allocate extra memory and then tell the VMM it can page out the corresponding "physical" pages. With ballooning, a misbehaving guest OS can touch the balloon driver's pages and cause the VMM to load them in from disk. When the VM is subsequently reverted to a snapshot, the compromised OS can again touch all of the balloon driver pages to see whether they have already been faulted in prior to the snapshot restoration. The VMM could mitigate this ballooning channel by restoring the state of its page table to match the original set of ballooned pages when restoring a snapshot. Of course, ballooning may be used as a covert channel between simultaneously executing VMs in systems other than Capsule.

There has also been research on sharing memory pages with the same content in virtual machines [3, 32]. If the VMM maps multiple pages to one machine page and marks them as copy-on-write, then the virtual machine will encounter a page fault when writing to one of them. If a VM sets a large number of pages to have the same value during secure mode, then a much larger number of page faults will take place when restoring a snapshot. However, these page faults will only be noticeable to the VM if execution resumes in normal mode before all of the VM's memory pages are loaded from the snapshot file. In this case, the VM can measure the total restoration time or write to pages before they have loaded and test for page faults to recover information. If the VM resumes execution after its memory has been fully restored and pages have been re-scanned for duplication, then this covert channel will not work.

The Capsule system does not over-commit memory for virtual machines, so the memory saving techniques mentioned above do not take effect. Our implementation of the Capsule system does not employ any counter-measures to covert channels based on memory paging.

Virtual Networks

The Capsule system blocks external network access during secure mode, but it relies on a virtual network for communication between the secure VM and the primary VM. While the virtual network itself is stateless (to the best of our knowledge), anything connected to the network could potentially be a target for relaying information from secure mode to normal mode. The DHCP and NAT services in the VMM are of particular interest. A compromised virtual machine may send arbitrary packets to these services in an attempt to affect their state. For example, a VM might be able to claim several IP addresses with different spoofed MAC addresses. It could then send ARP requests to the DHCP service following snapshot restoration to retrieve the spoofed MAC addresses, which contain arbitrary data. The Capsule system restarts both the DHCP and NAT services when switching back to normal mode to avert this covert channel.

Any system that allows both a high-security and low-security VM to talk to a third trusted VM (the secure VM in Capsule) exposes itself another covert channel. Naturally, all bets are off if the primary VM can compromise the secure VM. Even assuming the secure VM is not vulnerable, the primary VM may still be able to convince it to relay data from secure mode to normal mode. Like the DHCP service on the host, the secure VM's network stack stores information. For example, the primary VM could send out TCP SYN packets with specific source port numbers that contain several bits of data right before reverting to normal mode. When the primary VM resumes execution, it could see the source ports in SYN/ACK packets from the secure VM.

It is unclear exactly how much data can be stashed in the network stack on an unsuspecting machine and how long that information will persist. The only way to guarantee that a machine will not inadvertently relay state over the network is to reboot it. This is the approach we take to flush the secure VM's network stack state when switching back to normal mode in Capsule.

Guest Additions

The VMware VMM supports additional software that can run inside of virtual machines to enhance the virtualization experience. The features of guest additions include drag-and-drop between VMs and a shared clipboard. These additional features would undermine the security of any virtual machine system with multiple confidentiality levels and are disabled in the Capsule system.

6.5 Core Hardware Channels

Core hardware channels allow covert communication via one of the required primary devices: CPU or disk. Memory is a core device, but memory mapping is handled by the VMM, and is discussed in the previous section. Core hardware channels might exist in any multi-level secure system and are not specific to Storage Capsules or virtual machines. One difference between prior research and this work is that prior research focuses on a threat model of two processes that are executing simultaneously on the same hardware. In the Capsule system, the concern is not with simultaneous processes, but with a low-security process (normal-mode VM) executing on the same hardware after a high-security process (secure-mode VM) has terminated. This constraint rules out some traditional covert channels that rely on resource contention, such as a CPU utilization channel.

CPU State

Restoring a virtual machine's state from a snapshot will overwrite all of the CPU register values. However, modern processors are complex and store information in a variety of persistent locations other than architecture registers. Many of these storage areas, such as branch prediction tables, are not well-documented or exposed directly to the operating system. The primary method for extracting this state is to execute instructions that take a variable number of clock cycles depending on the state and measure their execution time, or exploit speculative execution feedback. Prior research describes how one can use these methods to leak information through cache misses [24, 33].

There are a number of counter-measures to covert communication through CPU state on modern processors. In general, the more instructions that execute in between secure mode and normal mode, the less state will persist. Because the internal state of a microprocessor is not completely documented, it is unclear exactly how much code would need to run to eliminate all CPU state. One guaranteed method of wiping out all CPU state is to power off the processor. However, recent research on cold boot attacks [12] shows that it may take several minutes for memory to fully discharge. This strategy would lead to an unreasonably long delay when switching from secure mode to normal mode.

The ideal solution for eliminating covert CPU state channels in Capsule and other virtualization systems would be with hardware support. The latest CPUs already support hardware virtualization, which allows them to fully emulate instruction sets for virtual machines. An additional mechanism is needed when switching between virtual machines that not only restores register and memory mappings, but also restores all state that could affect VM execution. This operation would load all of the cache data (L1, L2, and instruction), the branch prediction table, and any other intermediate state. It would also be sufficient to provide an instruction that would erase all of this data.

Although the Capsule system does not take counter-measures to prevent CPU state covert channels, the VMM restores a significant portion of the primary VM's original memory state before the VM can resume execution. This restoration will likely eliminate any residual cache lines from secure mode. It will also add a great deal of noise to any other CPU state, such as branch predictor tables, due to the number of instructions that execute during restoration. Computing the maximum bandwidth of CPU state channels in the Capsule system is future work.

Disk State

The caching and layout of files on disk, and in the file system, can be used to transmit information. Another covert channel exists in the positioning of the disk arm [10]. If the primary VM can manipulate disk state in a way that is readable after transitioning back to normal mode, then it can leak data. There are a few aspects of the Capsule system's design that make such an attack much more difficult. Any writes to the virtual disk during secure mode actually go to flat difference file that stores changes since the last snapshot operation. The primary VM has no control over where the data goes on disk, only how much is written, which would make exploitation of such a channel difficult. The VMM deletes this file when transitioning back to normal mode. This frees up any blocks that were allocated for the difference file. Finally, the VMM reads in the primary VM's original memory state from disk during snapshot restoration. This moves the position of the disk arm and adds considerable noise to the cache.

Despite the noise from transitioning back to normal mode, the disk cache is the most likely candidate for a covert channel. All writes are sent to a difference file and deleted, but reads touch the same virtual disk file during secure mode and normal mode. Consider the primary VM sequentially reading every block on its disk during secure mode to transmit a 1, or reading very little to transmit a 0. The primary VM might be able to tell what happened in secure mode by reading back disk blocks and measuring if they hit the disk cache. One would need to flush all of the disk caches to eliminate this channel. The Capsule system does not take any steps to mitigate disk state channels.

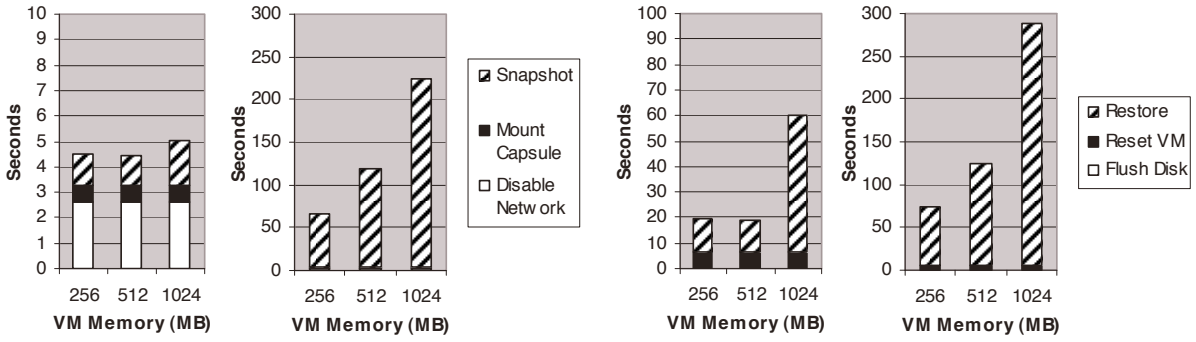

Figure 2. Transition times for different amounts of primary VM memory. (a) to secure mode with background snapshot, (b) to secure mode with full snapshot (c) to normal mode with background restore, and (d) to normal mode with full restore.

6.6 Mitigating VMM and Core Hardware Covert Channels

The design of Storage Capsules centers around improving local file encryption with a minimal impact on existing behavior. The user only has to take a few additional steps, and no new hardware is required. The current implementation is designed to guard against many covert channels, but does not stop leakage through all of them, such as the CPU state, through which data may leak from secure to normal mode. If the cost of small leaks outweighs usability and the cost of extra hardware, then there is an alternative design that can provide additional security.

One way of cutting off almost all covert channels would be to migrate the primary VM to a new isolated computer upon entering secure mode. This way, the virtual machine would be running on different core hardware and a different VMM while in secure mode, thus cutting off covert channels at those layers. VMware ESX server already supports live migration, whereby a virtual machine can switch from one physical computer to another without stopping execution. The user would have two computers at his or her desk, and use one for running the primary VM in secure mode, and the other for normal mode. When the user is done accessing a Storage Capsule, the secure mode computer would reboot and then make the Storage Capsule available for export over the network. This extension of the Capsule system's design would drastically reduce the overall threat of covert channels, but would requires additional hardware and could add usability impediments that would not be suitable in many environments.

7. Performance Evaluation

There are three aspects of performance that are important for Storage Capsules: (1) transition time to secure mode, (2) system performance in secure mode, and (3) transition time to normal mode. It is important for transitions to impose only minimal wait time on the user and for performance during secure mode to be comparable to that of a standard computer for common tasks. This section evaluates Storage Capsule performance for transitions and during secure mode. The experiments were conducted on a personal laptop with a 2 Ghz Intel T2500 processor, 2 GB of RAM, and a 5200 RPM hard drive. Both the host and guest operating systems (for the secure VM and primary VM) were Windows XP Service Pack 3, and the VMM software was VMware Workstation ACE Edition 6.0.4. The secure VM and the primary VM were both configured with 512 MB of RAM and to utilize two processors, except where indicated otherwise.

The actual size of the Storage Capsule does not affect any of the performance numbers in this section. It does, however, influence how long it takes to run an import or export. Both import and export operations are expected to be relatively rare in most cases – import only occurs when loading a Storage Capsule from an external location, and export is required only when sending a Storage Capsule to another user or machine. Importing and exporting consist of a disk read, encryption (for export only), a local network transfer, and a disk write. On our test system, the primary VM could import a 256 MB Storage Capsule in approximately 45 seconds and export it in approximately 65 seconds. Storage Capsules that are imported and exported more often, such as e-mail attachments, are likely to be much smaller and should take only a few seconds.

7.1 Transitioning to and from Secure Mode

The transitions to and from secure mode consist of several tasks. These include disabling/enabling device output, mounting/dismounting the Storage Capsule, saving/restoring snapshots, waiting for an escape sequence, and obtaining the encryption key. Fortunately, some operations can happen in parallel. During the transition to secure mode, the system can do other things while waiting for user input. The evaluation does not count this time, but it will reduce the delay experienced by the user in a real deployment. VMware also supports both background snapshots (copy-on-write) and background restores (copy-on-read). This means that execution may resume in the primary VM before memory has been fully saved or restored from the snapshot file. The system will run slightly slower at first due to page faults, but will speed up as the snapshot or restore operation nears completion. A background snapshot or restore must complete before another snapshot or restore operation can begin. This means that even if the primary VM is immediately usable in secure mode, the system cannot revert to normal mode until the snapshot is finished.

Figure 2 shows the amount of time required for transitioning to and from secure mode with different amounts of RAM in the primary VM. Background snapshots and restorations make a huge difference. Transitioning to secure mode takes 4 to 5 seconds with a background snapshot, and 60 to 230 seconds without. The time required for background snapshots, mounting the Storage Capsule, and disabling network output also stays fairly constant with respect to primary VM memory size. However, the full snapshot time scales linearly with the amount of memory. Note that the user must wait for the full snapshot time before reverting to normal mode.

The experiments show that reverting to normal mode is a more costly operation than switching to secure mode, especially when comparing the background restore to the background snapshot operation. This is because VMware allows a virtual machine to resume immediately during a background snapshot, but waits until a certain percentage of memory has been loaded in a background restore. Presumably, memory reads are more common than memory writes, so copy-on-read for the restore has worse performance than copy-on-write for the snapshot. VMware also appears to employ a non-linear strategy for deciding what portion of a background restore must complete before the VM may resume execution. It waited approximately the same amount of time when a VM had 256 MB or 512 MB of RAM, but delayed significantly longer for the 1 GB case.

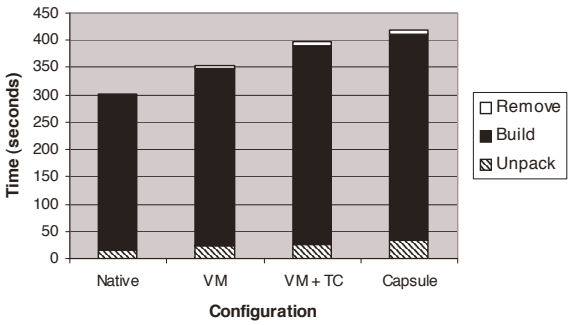

Figure 3. Results from building Apache with a native OS, a virtual machine, a virtual machine running TrueCrypt, and Capsule. Storage Capsules add only a 5% overhead compared to a VM with TrueCrypt, 18% slower than a plain VM, and 38% overhead compared to a native OS.

The total transition times to secure mode are all reasonable. Many applications will take 4 or 5 seconds to load a document anyway, so this wait time imposes little burden on the user. The transition times back to normal mode for 256 MB and 512 MB are also reasonable. Waiting less than 20 seconds does not significantly disrupt the flow of work. However, 60 seconds may be long wait time for some users. It may be possible to optimize snapshot restoration by using copy-on-write memory while the primary VM is in secure mode. This way, the original memory would stay in tact and the VMM would only need to discard changes when transitioning to normal mode. Optimizing transition times in this manner is future work.

7.2 Performance in Secure Mode

Accessing a Storage Capsule imposes some overhead compared to a normal disk. A Storage Capsule read or write request traverses the file system in the primary VM, and is then sent to the secure VM over the virtual network. The request then travels through a layer of encryption on the secure VM, out to its virtual disk, and then to the physical drive. We compared the disk and processing performance of Storage Capsules to three other configurations. These configurations consisted of a native operating system, a virtual machine, and a virtual machine with a TrueCrypt encrypted file container. For the evaluation, we ran an Apache build benchmark. This benchmark involves decompressing and extracting the Apache web server source code, building the code, and then removing all of the files. The Apache build benchmark probably represents the worst case scenario for Storage Capsule usage. We expect that the primary use of Storage Capsules will be for less disk-intensive activities like editing documents or images, for which the overhead should be unnoticeable.

Figure 3 shows the results of the Apache build benchmark. Storage Capsules performed well overall, only running 38% slower than a native system. Compared to a single virtual machine running similar encryption software (TrueCrypt), Storage Capsules add an overhead of only 5.1% in the overall benchmark and 31% in the unpack phase. This shows that transferring reads and writes over the virtual network to another VM has a reasonably small performance penalty. The most significant difference can be seen in the remove phase of the benchmark. It executes in 1.9 seconds on a native system, while taking 5.5 seconds on a VM, 6.5 seconds on a VM with TrueCrypt, and 7.1 seconds with Storage Capsules. The results from the VM and VM with TrueCrypt tests show, however, that the slowdown during the remove phase is due primarily to disk performance limitations in virtual machines rather than the Capsule system itself.

8. Conclusion and Future Work

This paper introduced Storage Capsules, a new mechanism for securing files on a personal computer. Storage Capsules are similar to existing encrypted file containers, but protect sensitive data from malicious software during decryption and editing. The Capsule system provides this protection by isolating the user's primary operating system in a virtual machine. The Capsule system turns off the primary OS's device output while it is accessing confidential files, and reverts its state to a snapshot taken prior to editing when it is finished. One major benefit of Storage Capsules is that they work with current applications running on commodity operating systems.

Covert channels are a serious concern for Storage Capsules. This research explores covert channels at the hardware layer, at the VMM layer, in external devices, and in the Capsule system itself. It looks at both new and previously examined covert channels from a novel perspective, because Storage Capsules have different properties than side-by-side processes in a traditional multi-level secure system. The research also suggests ways of mitigating covert channels and highlights their usability and performance trade-offs. Finally, we evaluated the overhead of Storage Capsules compared to both a native system and standard virtual machines. We found that transitions to and from secure mode were reasonably fast, taking 5 seconds and 20 seconds, respectively. Storage Capsules also performed well in an Apache build benchmark, adding 38% overhead compared to a native OS, but only a 5% penalty when compared to running current encryption software inside of a virtual machine.

In the future, we plan to further explore covert channels discussed in this work. This includes measuring their severity and quantifying the effectiveness of mitigation strategies. We also hope to conduct a study on usability of keyboard escape sequences for security applications. Storage Capsules rely on escape sequences to prevent spoofing attacks by malicious software, and it would be beneficial to know how many users of the Capsule system would still be vulnerable to such attacks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0705672. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

References

[1] M. Blaze. A Cryptographic File System for UNIX. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, Nov. 1993.

[2] R. Browne. An Entropy Conservation Law for Testing the Completeness of Covert Channel Analysis. In *Proc. of the 2nd ACM Conference on Computer and Communication Security (CCS)*, Nov. 1994.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.

[4] A. Czeskis, D. St. Hilair, K. Koscher, S. Gribble, and T. Kohno. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. In *Proc. of the 3rd USENIX Workshop on Hot Topics in Security (HOTSEC '08)*, Aug. 2008.

[5] M. Delio. Linux: Fewer Bugs than Rivals. *Wired Magazine,* `http://www.wired.com/ software/coolapps/news/2004/12/66022`, Dec. 2004.

[6] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7), Jul. 1977.

[7] C. Fruhwirth. LUKS – Linux Unified Key Setup. `http://code.google.com/p/ cryptsetup/`, Jan. 2009.

[8] T. Garfinkel, B. Pfaff, J, Chow, M. Rosenblum, D. Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proc. of the 19th*

ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003.

[9] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Jun. 2005.

[10] B. Gold, R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward. A Security Retrofit of VM/370. In *AFIPS Proc., 1979 National Computer Conference*, 1979.

[11] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. In *Proc. of the Symposium on Operating System Principles*, Dec. 1999.

[12] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. of 17th USENIX Security Symposium*, Jul. 2008.

[13] IEEE Computer Society. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, Apr. 2008.

[14] O. Lagerkvist. ImDisk Virtual Disk Driver. `http://www.ltr-data.se/opencode.html#ImDisk`, Dec. 2008.

[15] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proc. of the 12th USENIX Security Symposium*, Aug. 2003.

[16] M. Kang and I. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *Proc. of the 1st ACM Conference on Computer and Communication Security (CCS)*, Nov. 1993.

[17] R. Kemmerer. An Approach to Identifying Storage and Timing Channels. In *ACM Transactions on Computer Systems*, 1(3), Aug. 1983.

[18] M. Liskov, R. Rivest, and D. Wagner. Tweakable Block Ciphers. In *Advances in Cryptology – CRYPTO '02*, 2002.

[19] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. `http://www.vmware.com/pdf/Tech-TrendNotes.pdf`, 2000.

[20] Microsoft Corporation. BitLocker Drive Encryption: Technical Overview. `http://technet.microsoft.com/en-us/library/cc732774.aspx`, Jan. 2009.

[21] I. Moskowitz and A. Miller. Simple Timing Channels. In *Proc. of the IEEE Symposium on Security and Privacy*, May 1994.

[22] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1999.

[23] National Security Agency. Security-enhanced Linux. `http://www.nsa.gov/selinux`, Jan. 2008.

[24] C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan 2005*, May 2005.

[25] A. Roshal. WinRAR Archiver, a Powerful Tool to Process RAR and ZIP Files. `http://www.rarlab.com/`, Jan. 2009.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proc. of the 13th USENIX Security Symposium,* Aug. 2004.

[27] Secunia. Xen 3.x – Vulnerability Report. `http://secunia.com/product/15863/?task=statistics`, Jan. 2009.

[28] Secunia. Linux Kernel 2.6.x – Vulnerability Report. `http://secunia.com/product/2719/?task=statistics`, Jan. 2009.

[29] Trusted Computing Group. Trusted Platform Module Main Specification. `http://www.trustedcomputinggroup.org`, Ver. 1.2, Rev. 94, June 2006.

[30] J. Trostle. Multiple Trojan Horse Systems and Covert Channel Analysis. In *Proc. of Computer Security Foundations Workshop IV*, Jun. 1991.

[31] TrueCrypt Foundation. TrueCrypt – Free Open-Source On-the-fly Encryption. `www.truecrypt.org`, Jan. 2009.

[32] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[33] Z. Wang and R. Lee. Covert and Side Channels Due to Processor Architecture. In *Proc. of the 22nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.

[34] T. Weber. Criminals 'May Overwhelm the Web'. *BBC News*, `http://news.bbc.co.uk/1/hi/business/6298641.stm`, Jan. 2007.

[35] WinZip International LLC. WinZip – The Zip File Utility for Windows. `http://www.winzip.com/`, Jan. 2009.

[36] XenSource, Inc. Xen Community. `http://xen.xensource.com/`, Apr. 2008.

Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms

Ralf Hund Thorsten Holz Felix C. Freiling

Laboratory for Dependable Distributed Systems
University of Mannheim, Germany
hund@uni-mannheim.de, {holz,freiling}@informatik.uni-mannheim.de

Abstract

Protecting the kernel of an operating system against attacks, especially injection of malicious code, is an important factor for implementing secure operating systems. Several kernel integrity protection mechanism were proposed recently that all have a particular shortcoming: They cannot protect against attacks in which the attacker re-uses existing code within the kernel to perform malicious computations. In this paper, we present the design and implementation of a system that fully automates the process of constructing instruction sequences that can be used by an attacker for malicious computations. We evaluate the system on different commodity operating systems and show the portability and universality of our approach. Finally, we describe the implementation of a practical attack that can bypass existing kernel integrity protection mechanisms.

1 Introduction

Motivation. Since it is hard to prevent users from running arbitrary programs within their own account, all modern operating systems implement protection concepts that protect the realm of one user from another. Furthermore, it is necessary to protect the kernel itself from attacks. The basis for such mechanisms is usually called *reference monitor* [2]. A reference monitor controls all accesses to system resources and only grants them if they are allowed. While reference monitors are an integral part of any of today's mainstream operating systems, they are of limited use: because of the sheer size of a mainstream kernel, the probability that some system call, kernel driver or kernel module contains a vulnerability rises. Such vulnerabilities can be exploited to subvert the operating system in arbitrary ways, giving rise to so called *rootkits*, malicious software running without the user's notice.

In recent years, several mechanism to protect the integrity of the kernel were introduced [6, 9, 15, 19, 22], as we now explain. The main idea behind all of these approaches is that the memory of the kernel should be protected against unauthorized injection of code, such as rootkits. Note that we focus in this work on kernel integrity protection mechanisms and not on control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms, which are orthogonal to the techniques we describe in the following.

1.1 Kernel Integrity Protection Mechanisms

Kernel Module Signing. Kernel module signing is a simple approach to achieve kernel code integrity. When kernel module signing is enabled, every kernel module should contain an embedded, valid digital signature that can be checked against a trusted root certification authority (CA). If this check fails, loading of the code fails, too. This technique has been implemented most notably for Windows operating systems since XP [15] and is used in every new Windows system.

Kernel module signing allows to establish basic security guidelines that have to be followed by kernel code software developers. But the security of the approach rests on the assumption that the already loaded kernel code, i.e., the kernel and *all* of its modules, does not have a vulnerability which allows for execution of unsigned kernel code. It is thus insufficient to check for kernel code integrity only upon loading.

W⊕X. W⊕X is a general approach which aims at preventing the exploitation of software vulnerabilities at runtime. The idea is to prevent execution of injected code by enforcing the W⊕X property on all, or certain, page tables of the virtual address space: A memory page must never be writable *and* executable at the same time. Since injected code execution always implies previous

instruction writes in memory, the integrity of the code can be guaranteed. The W⊕X technique first appeared in OpenBSD 3.3; similar implementations are available for other operating systems, including the PaX [28] and Exec Shield patches for Linux, and PaX for NetBSD. Data Execution Prevention (DEP) [16] is a technology from Microsoft that relies on W⊕X for preventing exploitation of software vulnerabilities and has been implemented since Windows XP Service Pack 2 and Windows Server 2003.

The effectiveness of W⊕X relies on the assumption that the attacker wishes to modify and execute code in kernel space. In practice, however, an attacker usually first gains userspace access which implies the possibility to alter page-wise permission in the userspace portion of the virtual address space. Due to the fact that the no-executable bit in the page-table is not fine-grained enough, it is not possible to mark a memory page to be executably *only* in user mode. So an attacker may simply prepare her instructions in userspace and let the vulnerable code jump there.

NICKLE. *NICKLE* [19] is a system which allows for lifetime kernel code integrity, and thus rootkit prevention, by exploiting a technology called *memory shadowing*. NICKLE is implemented as virtual machine monitor (VMM) which maintains a separate so-called *shadow memory*. The shadow memory is not accessible from within the VM guest and contains copies of certain portions of the VM guest's main memory. Newly executing code, i.e., code that is executed for the first time, is authenticated using a simple cryptographic hash value comparison and then copied to the shadow memory transparently by NICKLE. Since the VMM is trusted in this model, it is guaranteed that no unauthenticated modifications to the shadow memory can be applied as executing guest code can never access the privileged shadow memory. Therefore, any attempt to execute unauthenticated code can be foiled in the first place. Another positive aspect of this approach is that it can be implemented in a rather generic fashion, meaning that it is perfectly applicable to both open source and commodity operating systems. Of course, NICKLE itself has to make certain assumptions about underlying file format of executable code, e.g., driver files, since it needs to understand the loading of these files. Currently, NICKLE supports Windows 2000, Windows XP, as well as Linux 2.4 and 2.6 based kernels. So far, NICKLE has been implemented for QEMU, VMware, and VirtualBox hypervisors. The QEMU source code is publicly available [20].

The isolation of the VMM from the VM Guest allows for a comparably unrestrictive threat model. In the given system, an attacker may have gained the highest level of privilege within the VM guest and may access the entire memory space of the VM. In other words, an adversary may compromise arbitrary system entities, e.g., files, processes, etc., as long as the compromise happens only *inside* the VM.

SecVisor. SecVisor [22] is a software solution that consist of a general, operating system independent approach to enforce W⊕X based on a hypervisor and memory virtualization. In the threat model for SecVisor an attacker can control everything but the CPU, the memory controller, and kernel memory. Furthermore, an attacker can have the knowledge of kernel exploits, i.e., she can exploit a vulnerability in kernel mode. In this setting, SecVisor "protects the kernel against code injection attacks such as kernel rootkits" [22]. This is achieved by implementing a hypervisor that restricts what code can be executed by a (modified) Linux kernel. The hypervisor virtualizes the physical memory and the MMU to set page-table-based memory protections. Furthermore, SecVisor verifies certain properties on kernel mode entry and exit, e.g., all kernel mode exits set the privilege level of the CPU to that of user mode or the instruction pointer points to approved code at kernel entry. Franklin et al. showed that these properties are prone to attacks and successfully injected code in a SecVisor-protected Linux kernel [8], but afterwards also corrected the errors found.

1.2 Bypassing Integrity Protection Mechanisms

Based on earlier programming techniques like *return-to-libc* [17, 21, 27], Shacham [23] introduced the technique of *return-oriented programming*. This technique allows to execute arbitrary programs in privileged mode without adding code to the kernel. Roughly speaking, it misuses the system stack to "re-use" existing code fragments (called *gadgets*) of the kernel (we explain this technique in more detail in Section 2). Shacham analyzed the GNU Linux libc of Fedora Core 4 on an Intel x86 machine and showed that executing one malicious instruction in system mode is sufficient to construct arbitrary computations from existing code. No malicious code is needed, so most of the integrity protection mechanisms fail to stop this kind of attack.

Buchanan et al. [4] recently extended the approach to the Sparc architecture. They investigated the Solaris 10 C library, extracted code gadgets and wrote a compiler that can produce Sparc machine programs that are made up entirely of the code from the identified gadgets. They concluded that it is not sufficient to prevent introduction of malicious *code*; we must rather prevent introduction of malicious *computations*.

Attacker Model. Like the mentioned literature, we base our work on the following reasonable attacker model. We assume that the attacker has full access to the user's address space in normal mode (local attacker) and that there exists at least one vulnerability within a system call such that it is possible to point the control flow to an address of the attacker's choice at least once while being in privileged mode. In practice, a vulnerable driver or kernel module is sufficient to satisfy these assumptions. Our attack model also covers the typical "remote compromise" attack scenario in network security where attackers first achieve local user access by guessing a weak password and then escalate privileges.

Contributions. In this paper, we take the obvious next step to show the futility of current kernel integrity protection techniques. We make the following research contributions:

- While previous work [4, 21, 23] was based on manual analysis of machine language code to create gadgets, we present a system that fully automates the process of constructing gadgets from kernel code and translating arbitrary programs into return-oriented programs. Our automatic system can use any kernel code (not only libc, but also drivers for example) even on commodity operating systems.

- Using our automatic system, we construct a portable rootkit for Windows systems that is entirely based on return-oriented-programming. It therefore is able to bypass even the most sophisticated integrity checking mechanism known today (for example NICKLE [19] or SecVisor [22]).

- We evaluate the performance of return-oriented programs and show that the runtime overhead of this programming technique is significant: In our tests we measured a slowdown factor of more than 100 times in sorting algorithms. However, for exploiting a system this slowdown might not be important.

Outline. The paper is structured as follows. In Section 2 we provide a brief introduction to the technique of return-oriented-programming. In Section 3 we introduce in detail our framework for automating the gadget construction and translating arbitrary programs into return-oriented programs. We present evaluation results for our framework in Section 4: Using ten different machines, we confirm the portability and universality of our approach. We present the design and implementation of a return-oriented rootkit in Section 5 and finally conclude the paper in Section 6 with a discussion of future work.

2 Background: Return-Oriented Programming

The idea behind a return-to-libc attack [17, 27] is that the attacker can use a buffer overflow to overwrite the return address on the stack with the address of a legitimate instruction which is located in a library, e.g., within the C runtime *libc* on UNIX-style systems. Furthermore, the attacker places the arguments to this function to another portion of the stack, similar to classical buffer overflow attacks. This approach can circumvent some buffer overflow protection techniques, e.g., non-executable stack.

The technique of return-oriented programming was introduced by Shacham et al. [4, 23]. It generalizes return-to-libc attacks by chaining short new instructions streams ("useful instructions") that then return. Several instructions can be combined to a *gadget*, the basic block within return-oriented programs that for example computes the AND of two operands or performs a comparison. Gadgets are self-contained and perform one well-defined step of a computation. The attacker uses these gadgets to cleverly craft stack frames that can then perform arbitrary computations. Fig. 1 illustrates the process of return-oriented programming. First, the attacker identifies useful instructions that are followed by a `ret` instruction (e.g., instruction sequences A, B and C in Fig. 1). These are then chained to gadgets to perform a certain operation. For example, instruction sequences A and B are chained together to gadget 1 in Fig. 1. On the stack, the attacker places the appropriate return addresses to these instruction sequences. In the example of Fig. reffig:rop the return addresses on the stack will cause the executions of gadget 1 and then gadget 2. The stack pointer ESP determines which instruction to fetch and execute, i.e., within return-oriented programming the stack pointer adopts the role of the instruction pointer (IP): Note that the processor does not automatically increment the stack pointer, but the `ret` instruction at the end of each useful instruction does.

The authors showed that both the libc library of Linux running on the x86 architecture (CISC) as well as the libc library of Solaris running on a SPARC (RISC) contain enough useful instructions to construct meaningful gadgets. They manually analyzed the libc of both environments and constructed a library of gadgets that is Turing-complete. We extend their work by presenting the design and implementation of a fully automated return-oriented framework that can be used on commodity operating systems. Furthermore, we describe an actual attack against kernel integrity protection systems by implementing a return-oriented rootkit.

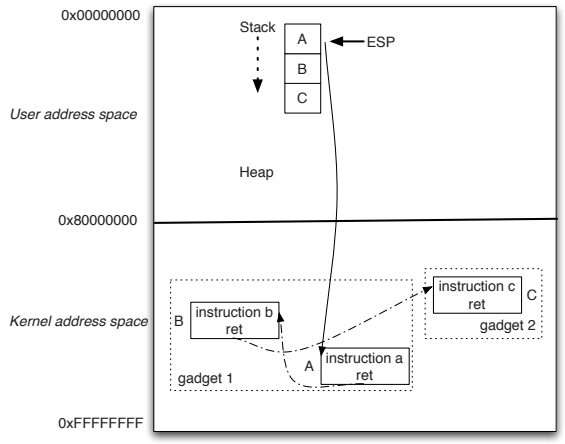

Figure 1: Schematic overview of return-oriented programming on the Windows platform

3 Automating Return-Oriented Programming

In order to be able to create and execute return-oriented programs in a generic way, we created our own, modular toolset which enables one to abstract from the varying concrete conditions one faces in this context. Additionally, our system greatly simplifies the development of return-oriented programs by providing high-level constructs to accomplish certain tasks. Figure 2 provides a schematic overview of our system; it is partitioned into three core components:

- *Constructor*. The Constructor scans a given set of files containing executable code, spots useful instruction sequences and builds return-oriented gadgets in an automatic fashion. These well-defined gadgets serve as a low-level abstraction and interface to the Compiler.
- *Compiler*. The Compiler provides a comparatively high-level language for programming in a return-oriented way. It takes the output of the Constructor along with a source file written in a dedicated language to produce the final memory image of the program.
- *Loader*. As the Compiler's output is position independent, it is the task of the Loader to resolve relative memory addresses to absolute addresses. This component is implemented as library that is supposed to be linked against by an exploit.

All components have been implemented in C++ and currently we support Windows NT-based operating systems running on an IA-32 architecture. In the following paragraphs, we give more details on each component's inner workings.

3.1 Automated Gadget Construction

One of the most essential parts of our system is the *automated construction* of return-oriented gadgets, thus enabling us to abstract from a concrete set of executable code being exploited for our purposes. This is in contrast to previous work [4, 23], which focused on concrete versions of a C library instead. Our system works on an arbitrary set of files containing valid x86 machine code instructions; we will henceforth refer to these files as the *codebase*.

Our framework implements the creation of gadgets in the so-called *Constructor*, which performs three subsequent jobs: First, it scans the codebase to find *useful instruction sequences*, i.e., instructions preceding a return (`ret`) instruction. These instructions can then be used to implement a return-oriented program by concatenating the sequences in a specific way. Our current implementation targets machines running an arbitrary Windows version as operating system and thus we use all driver executables and the kernel as codebase in the scanning phase. In the second step, our algorithm chains the instruction sequences together to form *gadgets* that perform basic operations. We define the term gadget analog to Shacham [23], i.e., gadgets comprise composite useful instruction sequences to accomplish a well-defined task (e.g., perform an AND operation or check a boolean condition). More precisely, when we talk of *concrete gadgets*, we mean the corresponding *stack allocation*, i.e., the contents (return-addresses, constants, etc.) of the memory area the stack register points to. Gadgets represent an intermediate abstraction layer whose elements are the basic units subsequently used by the Compiler for building the final return-oriented program. Gadgets being written to the Constructor's final output file are called *final gadgets*. In the third step, the Constructor searches for exported symbols in the codebase and saves these in the output file for later use by the Compiler.

3.1.1 Finding Useful Instruction Sequences

The first decision that has do be made is describing the basic instruction sequences being the very core of a return-oriented program. As previously mentioned, these instructions occur prior to a `ret` x86 assembler instruction. We have to decide how many instructions preceding a return are considered. For instance, the Constructor might look for sequences such as `mov eax, ecx; add eax, edx; ret`, and incorporate these in the subsequent gadget construction. An easier approach, however, is to consider only a single instruction before a return instruction. Of course, the former attempt has the advantage of being more comprehensive, along with the drawback of requiring additional overhead. This stems from the fact that one has to take every instruction's pos-

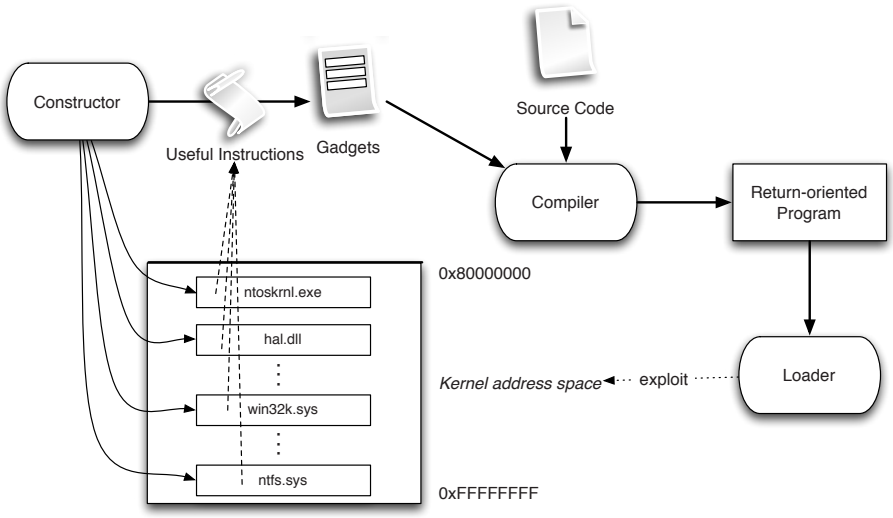

Figure 2: Schematic system overview

sible side-effects on registers and memory into account. In our work, we have thus chosen to implement the latter approach. Rudimentary research has shown that the additional value of using longer instruction sequences hardly justifies the imposed overhead since the effect of the former is not very significant in practice: We have observed that the high density of the x86 instructions encoding does not introduce substantial surplus concerning additional instruction sequences. We would also like to stress that this simplified approach has not turned out to be problematic in our work so far since the codebase of every system we evaluated held sufficient instruction sequences to implement arbitrary return-oriented programs (see Section 4 for details). However, our system might still be extended in the future in order to support more than one instruction preceding a return instruction.

To scan the codebase for useful instruction sequences, the Constructor first enumerates all sections of the PE file that contain executable code and scans these for x86 `ret` opcodes. In addition to the standard `ret` instruction, which has the opcode `0xC3`, we are also interested in return instructions that add an immediate value to the stack, represented by opcode `0xC2` and followed by the 16bit immediate offset. The former are favorable to the latter as they induce less memory consumption in the stack allocation since we need to append effectively unused memory before the next instruction.

Having found all available return instructions, the Constructor then bytewise disassembles the sequence backwards, thereby building a trie. This works analogoulsy to the method already described by Shacham [23]. In order to disassemble encoded x86 instructions, our program uses the *distorm* library [10].

3.1.2 Building Gadgets

The next logical step is chaining together instruction sequences to form structured gadgets that perform basic operations. Gadgets built by the Constructor form the very basic entities that are chained together by the Compiler for building the program stack allocation. Due to the clear separation of the Constructor and the Compiler, final gadgets are independent of each other. Therefore, each final gadget constitutes an autonomous piece of return-oriented code: Final gadgets take a set of source operands, perform a well-defined operation on these, and then write the result into a destination operand. In our model, source and destination operands are always memory variable addresses. For example, an addition gadget takes two source operands, i.e., memory addresses to both input variables, as input, adds both values, and then writes back the result to the memory address pointed to by the destination operand. There are certain exceptions to this rule, namely final gadgets that perform very specific tasks for certain situations, e.g., manipulating the stack register directly. Final gadgets are designed to be fine-grained with respect to the constraints imposed by the operand model. They can be separated into three classes: Arithmetic, logical and bitwise operations; control flow manipulations (static and dynamic); and stack register manipulations.

The crucial point in gadget construction concerns the algorithm that is deployed to spot appropriate useful instructions and the rules in which they are chained together. We consider completeness, memory consumption, and runtime to be the three dominating properties. By completeness, we mean the algorithm's fundamental ability to construct gadgets even in a minimal codebase,

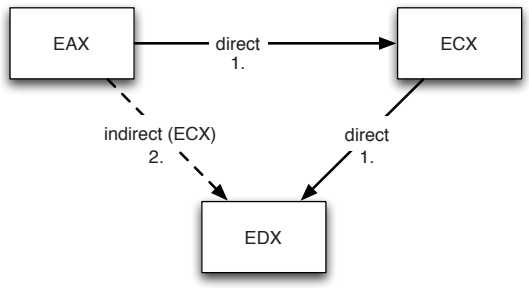

Figure 3: MOV connection graph: Chained instructions can be used to emulate other instructions.

where minimal indicates a codebase with a theoretically minimal set of instruction sequences to allow for corresponding gadget computations. By memory consumption, we denote that the constructed gadgets should be preferably small in size. By runtime, we mean that the algorithm should terminate within a reasonable period of time. Due to the CISC nature of x86 and the corresponding complexity of the machine language, we consider the completeness property to be the most difficult one to achieve. The many subtle details of this platform make it hard to find all possible combinations of useful instruction performing a given operation.

In the following, we provide a deeper look into our gadget construction algorithm. As with every modern CPU, x86 is a register-based architecture. This observations drives the starting point of our algorithm in that its first step is to define a set of general purpose registers that are allowed to be used, i.e., read from or written to, by gadget computations. This also has the positive side-effect that it enables an easy way to control which registers are modified by the return-oriented program. We will henceforth call these registers *working registers*.

Basic Gadgets. Starting from this point, we gradually construct lists of gadgets performing a related task for each working register. More precisely, the first step is to check which register can be loaded with fixed values, an operation that can easily be achieved with a `pop <register>; ret` sequence (*register-based constant load gadgets*). Afterwards, the Constructor searches for unary instructions sequences, e.g., `not` or `neg`, that take working registers as their operands (*register-based unary operation gadgets*). Subsequently, the algorithm checks which working registers are connected by binary instruction sequences, e.g., `mov`, `add`, `and`, and the like (*register-based binary operation gadgets*). In order to find indirectly connected registers, we build a directed graph for each operation whereas a node represents a register and an edge depicts an operation from the source register to the destination register (also always being a source operand on x86). Then, we traverse all paths in the graph for each node. For example, let us assume the following situation: The given codebase allows for the execution of `mov ecx, eax; ret` and `mov edx, ecx; ret` sequences, but does not supply `mov edx, eax` sequences. We can easily find the corresponding path in our graph and hence construct a gadget that moves the content of `eax` to `edx` by chaining together both sequences (see Fig. 3). Since x86 is not a load-store-architecure, i.e., most instructions may take direct memory operands, we also search for memory operand based instructions (*register-based memory load-/operation gadgets*). This also allows us to check which working registers can be loaded with memory contents, for instance, `mov eax, [ecx]; ret` easily allows us to load an arbitrary memory location into `eax` by preparing `ecx` accordingly. The result of this first stage of the algorithm are lists of internal gadgets being bound to working registers and performing certain operations on these.

In the next stage, our algorithm merges working register-based gadgets to form new, final gadgets that perform certain operations, e.g., addition, multiplication, bitwise OR, and so on (*final unary/binary operation gadgets*). Therefore, it generates every possible combination of according register-based load/store and operation gadgets to choose the one being minimal with respect to consumed memory space. In the construction, we have to take into account possibly emerging side-effects when connecting instruction sequences. We say that a gadget has a *side-effect* on a given register when it is modified during execution. For instance, if we wish to build a gadget that loads two memory addresses into `eax` and `ecx` and appends an `and eax, ecx; ret` sequence, we have to make sure that both load gadgets do not have side-effects on each other's working register.

Control Flow Alteration Gadgets. Afterwards, the algorithm constructs final gadgets that allow for static and dynamic control flow alterations in a return-oriented program (*final comparison and dynamic control flow gadgets*). Therefore, we must first compare two operands with either a `cmp` or `sub` instruction, both have the same impact on the `eflags` registers which holds the condition flags. The main problem in this context is gaining access to the CPU's flag registers as this is only possible with a limited set of instructions. As already pointed out by Shacham [23], a straightforward solution is to search for the `lahf` instruction, which stores the lower part of the `eflags` register into `ah`. Another possibility is to search for `setCC` instructions, which store either one or zero depending on whether the condition is true or not. Thereby, `CC` can be any condition known to the CPU, e.g., equal, less than, greater or equal, and so on. Once

we have stored the result of the comparison (where 1 means true and 0 means false) the natural way to proceed is to multiply this value by four and add it to a jump table pointer. Then, we simply move the stack register to the value being pointed at.

Additional Gadgets. Finally, the Constructor builds some special gadgets that enable very specific tasks, such as, e.g., pointer dereferencing (*final dereferencing gadgets*), and direct stack or base register manipulation (*stack register manipulation gadgets*). The latter are required in certain situation as described in the next section. The final output of the Constructor is an XML file that describes the finals gadgets along with a list of exported symbols from the codebase.

Turing Completeness. Gadgets are used within return-oriented programming as the basis blocks of each computation. An interesting question is now which kind of gadgets are needed such that return-oriented programming is *Turing complete*, i.e., it can compute every Turing-computable function [30]. We construct gadgets to load/store variables (including pointer dereferencing), branch instructions, and also gadgets for arithmetic operations (i.e., addition and not). This set of gadgets is minimal in the sense that we can construct from these gadgets any program: Our return-oriented framework can implement so called *GOTO languages*, which are Turing complete [12].

3.2 Compiler

The Compiler is the next building block of our return-oriented framework: This tool takes the final gadgets constructed by the Constructor along with a high-level language source file as input to produce the stack allocation for the return-oriented program. The Compiler acts as an abstraction of the concrete codebase so that developers do not have to mess with the intricacies of the codebase on the lowest layer; moreover, it provides a comparatively easy and abstract way to formulate a complex task to be realized in a return-oriented fashion. The Compiler's output describes the stack allocation as well as additional memory areas serving a specific purpose in a position independent way, i.e., it only contains relative memory addresses. This stems from the fact that the Compiler cannot be aware of the final code locations since drivers may be relocated in kernel memory due to address conflicts. Moreover, the program memory's base location may be unknown at this stage. It is hence the task of the Loader to resolve these relative addresses to absolute addresses (see next section).

3.2.1 Dedicated Language

Naturally, one of the first considerations in compiler development concerns the programming language employed. One possibility is to build the Compiler on top of an already existing language, ideally one that is designed the accomplish low-level tasks, such as C. However, this also introduces a profound overhead as all the language's peculiarities, e.g., the entire type system, must be implemented in a correct manner. Due to our very specific needs, we have found none of the existing language to be suited for our purpose and thus decided to create a dedicated language. It bears certain resemblance to many existing languages, specifically C. Our dedicated language provides the following code constructs:

- subroutines and recursive subroutine calls,
- a basic type-system that consists of two variable types,
- all arithmetic bitwise, logical and pointer operators known to the C language with some minor deviations, and
- nested dynamic control flow decisions and nested conditional looping.

Additionally, we also supports external subroutine calls which enables one to dispatch operations to exported symbols from drivers or the kernel; this gives us more flexibility, greatly simplifies the development of return-oriented programs, and also substantially decreases stack allocation memory consumption.

Two basic variable types are supported: Integers and character arrays, the former being 32bit long while in case of the latter, strings are zero-terminated just as in C. Along with the ability to call external subroutines, this enables us to use standard C library functions exported by the kernel to process strings within the program. We do not need support for short and char integers for now as we do not consider these to be substantially relevant for our needs. Short integer operations thus must be emulated by the return-oriented program when needed.

The Compiler has been implemented in C++ using the ANTLR compiler generation framework [29]. Source code examples for our dedicated programming language are introduced in Section 5.4 and in Appendix B.

3.2.2 Memory Layout

Just as the Constructor chains together instruction sequences, the Compiler chains together gadgets to build a program performing the semantics imposed by the source code. Apart from that, it also defines the memory layout and assigns code and data to memory locations. By *code*, we henceforth mean the stack allocation of the

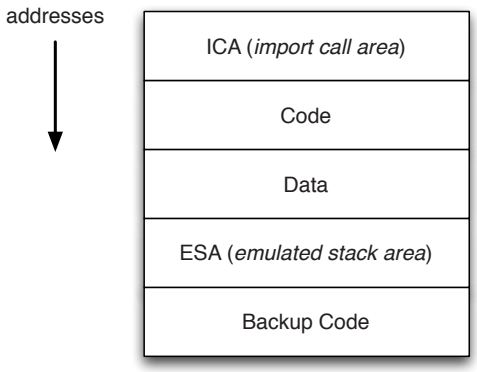

Figure 4: Memory layout of program image within our return-oriented framework

sum of all gadgets of a program (mostly return addresses to instruction sequences); this must not be confused with real CPU code, i.e., the code as we defined does not need *any* executable memory, but appears like usual data to the processor. This is the key concept in bypassing kernel integrity protection mechanisms: We do not need to inject code since we re-use existing code within the kernel during an exploit.

When we use the term *data*, we henceforth mean the memory area composed by the program's variables and temporary internal data required by computations. We then constitute the memory layout to consist of a linear memory space we hereafter call the *program image*, which is shown in Fig. 4. Furthermore, some regions of this space serve special purposes we describe later on. In total, we separate the program image into five sections: Code, data, ICA, ESA and backup code.

The so-called *import call area* (ICA) resides at the very beginning, i.e., the lowest address, of the address space. When executing external function calls, the program prepares the call to be dispatched with the stack pointer `esp` pointing at the end (the highest address) of the ICA. Therefore, it first prepares this region by copying the arguments and return addresses to point to specific stack manipulation gadgets. Special care has to be taken concerning the imposed calling convention of the callee. We support both relevant conventions, namely `stdcall`, i.e., the callee cleans up the stack, and `cdecl`, i.e., the caller cleans up the stack. The need for such a dedicated section stems from the fact that, upon entry, the callee considers all memory regions below `esp` to be available to hold its local variables, hence overwriting return-oriented code that might still be needed at a later stage, i.e., when a jump back occurs.

Following the ICA, the Compiler places the code, i.e., return addresses and constant values to be popped into registers, followed by the data section which holds the actual explicit variables as well as some implicit temporary variables that are mandatory during computation. After that, the *emulated stack area* (ESA) resides, which is used to emulate a "stack in the stack" to allow for recursive subroutine calls in the return-oriented program. The program image is terminated by an optional backup of the code section, a necessity that arises from a peculiarity of the Windows operating system we discuss later on in Section 5.2.

3.2.3 Miscellaneous

We also provide special language constructs enabling one to retransfer the CPU control flow to a non-return oriented source. For instance, in the typical case of an exploit and subsequent execution of return-oriented code, we might wish to return to the vulnerable code to allow for a continuation of execution. Therefore, we must restore the `esp` register to point to its original value. Our language hence provides appropriate primitives to tamper with the stack.

3.3 Loader

The final building block of our system consists of the Loader whose main task is to resolve the program image's relative addresses to absolute addresses. Therefore, it must first enumerate all loaded drivers in the system and retrieve their base addresses. Luckily, Windows provides a function by the name of `EnumDeviceDrivers` that lets us accomplish this task even in usermode.

For the sake of flexibility, the Loader is implemented as a dynamic link library (DLL). The actual exploit transfers the task of building the final program image to the Loader and then adjusts the exploit to modify the instruction pointer `eip` to a gadget that modifies the stack (e.g., `pop esp; ret`) to start the execution of the return-oriented program. It is therefore sufficient for the exploit to be able to modify eight subsequent bytes in the stack frame: The first four bytes are a return address (of the sequence `pop esp; ret`) that is executed upon the next `ret` in the current control flow; the last four bytes point to the entry point of the program image to which control will flow after the execution of the next `ret`.

4 Evaluation Results

We implemented the system we described in the previous section in the C++ programming language. The Constructor consists of about 3,400 lines of code (LOC), whereas the Compiler is implemented in about 3,200 LOC. The loader only needs 700 LOC.

In the following, we present evaluation results for the individual components of our framework. We first show measurement results for the Constructor and Compiler and then provide several examples of the gadgets constructed by our tools. Finally we also measure the runtime overhead of return-oriented programs.

4.1 Constructor and Compiler

4.1.1 Evaluation of Useful Instructions and Gadget Construction

One goal of our work is to fully automate the process of constructing gadgets from kernel code on different platforms without the need of manual analysis of machine language code. We thus tested the Constructor on ten different machines running different versions of Windows as operating system: Windows 2003 Server, Windows XP, and Windows Vista were considered in different service pack versions to assess a wide variety of platforms. On each tested platform the Constructor was able to find enough useful instructions to construct all important gadgets that are needed by the Compiler, i.e., on each platform we are able to compile arbitrary return-oriented programs. This substantiates our claim that our framework is general and portable.

Table 1 provides an overview of the results for the gadget construction algorithm for six of the ten test configurations. We omitted the remaining four test results for the sake of brevity; the results for these machines are very similar to the listed ones. The table contains test results for two scenarios: On the one hand, we list the number of return instructions and trie leaves when using *any* kernel code, e.g., all drivers and kernel components. On the other hand, we list in the restricted column (res.) the results when using *only* the main kernel component (`ntoskrnl.exe`) and the Win32-subsystem (`win32k.sys`) for extracting useful instructions. These two components are available in any Windows environment and thus constitute a memory region an attacker can always use to build gadgets.

The number of return instructions found varies with the platform and is influenced by many factors, mainly OS version/service pack and hardware configuration. Especially the hardware configuration can significantly enlarge the number of available return instructions since the installed drivers add a large codebase to the system: We found that often graphic card drivers add thousands of instructions that can be used by an attacker. For the complete codebase we found that on average every 162nd instruction is a return. Therefore an attacker typically finds tens of thousands of instructions she can use.

If the attacker restricts herself to using only the core kernel components, she is still able to find enough return instructions to be able to construct all necessary gadgets: We found that on average every 153rd instruction is a return, indicating a more dense structure within the core kernel components. These returns and the preceding instructions could be used to construct the gadgets in all tested environments. This result indicates that on Window-based systems an attacker can implement an arbitrary return-oriented program since all important gadgets can be built.

The most common instruction preceding a return is `pop ebp`: On average across all tested systems, this instruction was found in about 72% of the cases. This is no surprise since the sequence `pop ebp; ret` is the standard exit sequence for C code. Other common instructions the Constructor finds are `add esp, <const>` (12.2%), `pop (eax|ecx|edx)` (4.2%), and `xor eax, eax` (3.7%). Other instructions can be found rather seldom, but if a given instruction occurs at least once the attacker can use it. For example, the instruction `lahf`, which is used to access the CPU's flag registers, was commonly found less than 10 times, but nevertheless the attacker can take advantage of it.

4.1.2 Gadget Examples

In order to illustrate the gadgets constructed by our framework, we present a few examples of gadgets in this section. A full listing of all gadgets constructed during the evaluation on ten different machines is available online [13] such that our results can be verified.

Figure 5 shows the AND gadget constructed on two different machines both running Windows XP SP2. In each of the sub-figures, the left part displays the instructions that are actually used for the computation: Remember that our current implementation considers one instruction preceding a return instruction, i.e., after each of the displayed instructions one implicit `ret` instruction is executed. The right part shows the memory locations where the instruction is found within kernel memory (R), or indicates the label name (L). Labels are memory variable addresses.

The two gadgets each perform a logical AND of two values. This is achieved by loading the two operands into the appropriate registers (`pop, mov` sequence), then performing the `and` instruction on the registers, and finally writing the result back to the destination address. Although both programs are executed on Windows XP SP2 machines, the resulting return-oriented code looks completely different since useful instructions in different kernel components are used by the Constructor.

Another example of a gadget constructed by our framework is shown in Figure 6. The left example shows a gadget for a machine running Windows Vista, while the example on the right hand side is constructed on a ma-

Machine configuration	# `ret` inst.	# trie leaves	# `ret` inst. (res)	# trie leaves (res)
Native / XP SP2	118,154	148,916	22,398	25,968
Native / XP SP3	95,809	119,533	22,076	25,768
VMware / XP SP3	58,933	67,837	22,076	25,768
VMware / 2003 Server SP2	61,080	70,957	23,181	26,399
Native / Vista SP1	181,138	234,685	30,922	36,308
Bootcamp / Vista SP1	177,778	225,551	30,922	36,308

Table 1: Overview of return instructions found and generated trie leaves on different machines

```
pop ecx             | R: ntkrnlpa.exe:0006373C
                    | L: <RightSourceAddress>+4
mov edx, [ecx-0x4]  | R: vmx_fb.dll:00017CBD
pop eax             | R: ntkrnlpa.exe:000436AE
                    | L: <LeftSourceAddress>
mov eax, [eax]      | R: win32k.sys:000065D1
and eax, edx        | R: win32k.sys:000ADAE6
pop ecx             | R: ntkrnlpa.exe:0006373C
                    | L: <DestinationAddress>
mov [ecx], eax      | R: win32k.sys:0000F0AC
```

```
pop ecx             | R: nv4_mini.sys:00005A15
                    | L: <RightSourceAddress>-4
pop eax             | R: nv4_mini.sys:00074EF2
                    | L: <LeftSourceAddress>
mov eax, [eax]      | R: nv4_disp.dll:00125F30
and eax, [ecx+0x4]  | R: sthda.sys:000024ED
pop ecx             | R: nv4_mini.sys:00005A15
                    | L: <DestinationAddress>
mov [ecx], eax      | R: nv4_disp.dll:000DE9DA
```

Figure 5: Example of two AND gadgets constructed on different machines running Windows XP SP2. The implicit `ret` instruction after each instruction is omitted for the sake of brevity.

chine running Windows 2003 Server. Again, the memory locations of the gadget instructions are completely different since the Constructor found different useful instruction sequences that are then used to build the gadget.

4.2 Runtime Overhead

The average runtime of the Constructor for the restricted set of drivers that should be analyzed is 2,009 ms, thus the time for finding and constructing the final gadgets is rather small.

To assess the overhead of return-oriented programming in real-world settings, we also measured the overhead of an example program written within our framework compared to a "native" implementation in C. Therefore, we implemented two identical versions of QuickSort, one in C and one in our dedicated return-oriented language. The source code of the latter can be seen in Appendix B.

Both algorithms sort an integer array of 500,000 randomly selected elements and the evaluations were carried out on an Intel Core 2 Duo T7500 based notebook running Windows XP SP3. The C code was compiled with Microsoft Visual Studio 2008; in order to improve the fundamental expressiveness of the comparison, all compiler optimizations were disabled. Each algorithm was executed three times and we calculated the average of the runtimes.

The return-oriented QuickSort took *21,752 ms* on average compared to *161 ms* for C QuickSort. The results clearly show that the overhead imposed by return-oriented programs is significant; on average, they were 135 times slower than their C counterparts. We would like to stress that we did not build our system with speed optimizations in mind. Additionally, in our domain, return-oriented rootkits usually do not involve time-intensive computations, thus the slowness might not be a problem in practice. On the other hand, the overhead might well be exploited by detection mechanisms that try to find return-oriented programs.

5 Return-Oriented Rootkit

In order to evaluate our system in the presence of a kernel vulnerability, we have implemented a dedicated driver containing insecure code. Remember that our attack model includes this situation. By this example, we show that our systems allows us to implement a return-oriented rootkit in an efficient and portable manner. This rootkits bypasses kernel code integrity mechanisms like NICKLE and SecVisor since we do not inject new code into the kernel, but only execute code that is already available. While the authors of NICKLE and SecVisor acknowledge that such a vulnerability could exist [19, 22], we are the first to actually show an implementation of an attack against these systems. In the following, we first introduce the different stages of the infection process and afterwards describe the internals of our rootkit example.

```
'LoadEspPointer' gadget:
pop ecx         | R: nvlddmkm.sys:000156F5
                | L: <Address>
mov eax, [ecx]  | R: ntkrnlpa.exe:002D15C3
mov eax, [eax]  | R: win32k.sys:000011AE
pop ecx         | R: nvlddmkm.sys:000156F5
                | L: &<LocalVar>
mov [ecx], eax  | R: ntkrnlpa.exe:0002039B
pop esp         | R: nvlddmkm.sys:00036A54
                | L: <LocalVar>
```

```
'LoadEspPointer' gadget:
pop eax         | R: ntkrnlpa.exe:0001CD4F
                | L: <Address>
mov eax, [eax]  | R: win32k.sys:00087E17
mov eax, [eax]  | R: win32k.sys:00087E17
pop ecx         | R: ntkrnlpa.exe:00080A8D
                | L: &<LocalVar>
mov [ecx], eax  | R: win32k.sys:000A8DDB
pop esp         | R: ntkrnlpa.exe:00081A67
                | L: <LocalVar>
```

Figure 6: Example of gadget constructed on a machine running Windows Vista SP1 (left) and Windows 2003 Server (right). Again, the implicit `ret` instruction after each instruction is omitted.

5.1 Experimental Setup

Vulnerability. As already stated, we assume the presence of a vulnerability in kernel code that enables an exploit to corrupt the program flow in kernel mode. More precisely, our dedicated driver contains a specially crafted buffer overflow vulnerability that allows an attacker to tamper with the kernel stack. The usual way to implement driver-to-process communication is to provide a device file name being accessible from userspace. The process hence opens this device file and may send data to the driver by writing to it. Write requests trigger so-called *I/O request packets* (IRP) at the driver's callback routine. The driver then takes the input data from userspace and copies it into its own local buffer without validating its length. This leads to a classical buffer overflow attack and enables us to write stack values of arbitrary length.

Exploit. We exploit this vulnerability by writing an oversized buffer to the device file, thereby replacing the return value on the stack to point to a `pop esp; ret` sequence, and the next stack value to point to the entrypoint of the return-oriented program. By overwriting these eight bytes, we manage to modify the stack register to point to the beginning of our return-oriented program. Of course, the vulnerability itself may vary in its concrete nature, however, any similar insecure code allows us to mount our attack: A single vulnerability within the kernel or any third-party driver is enough to attack a system and start the return-oriented program.

The only question that remains is where to put the program image. We basically have two options: First, the exploit could overwrite the entire kernel stack with our return-oriented program; in case of the above vulnerability, this would be possible as there is no upper limit. In case of Windows, the kernel stack size has a fixed limit of 3 pages which heavily constrains this option. Second, the exploit could, at least initially, keep the program image in userspace memory. We prefer the latter approach to implement our *rootkit loader*, although it has some implications that need to be addressed as we now explain.

5.2 Intricacies in Practice

One of the main practical obstacles that we faced stems from the way how Windows treats its kernel stack. All current Windows operating systems separate kernel space execution into several *interrupt request levels* (IRQL). IRQLs introduce a priority mechanism into kernel-level execution and are similar to user-level thread priorities. Every interrupt is executed in a well-defined IRQL; whenever such an interrupt occurs, it is compared to the IRQL of the currently executing thread. In case the current IRQL is above the requested one, the interrupt is queued for later rescheduling. As a consequence, an interrupt cannot suspend a computation running at a higher IRQL. This has some implications concerning accessing pageable memory in kernel mode since page-access exceptions are being processed in a specific IRQL (`APC_LEVEL`, to be precise) while other interrupts are handled at higher IRQLs. Hence, the kernel and drivers must not access pageable memory areas at certain IRQLs.

Unfortunately, this leads to some problems due to a peculiarity of Windows kernels: Whenever interrupts occur and hence must be handled, the Windows kernel borrows the current kernel stack to perform its interrupt handling. Therefore, the interrupt handler allocates the memory *below* the current value of `esp` as the handler's stack frame. While this is totally acceptable in common situations, it leads to undesirable implications in case of return-oriented programs as the stack values below the current stack pointer may indeed be needed in the subsequent execution. As described in Section 3.1.2, control flow branches are stack register modifications: When the program wants to jump backwards, it may fail at this point since the prior code might have been overwritten by the interrupt handler in the first place. To solve this problem, the Compiler provides an option to dynamically restore affected code areas: Whenever a return-oriented

control flow transition backwards occurs (which hence could have been subject to unsolicited modifications), we first prepare the ICA to perform a `memcpy` call that restores the affected code from the backup code section and subsequently performs the return-oriented jump. This works since the ICA is located below the code section and hence the code section cannot be overwritten during the call. The data and backup section will never be overwritten as they are always on top of every possible value of `esp`.

Furthermore, we will also run into IRQL problems in case the program stack is located in pageable memory: As soon as an interrupt is dispatched above `APC_LEVEL`, a blue-screen-of-death occurs. This problem should be overcome by means of the `VirtualLock` function which allows a process to lock a certain amount of pages into physical memory, thereby eliminating the possibility of paging errors on access. However, due to reasons which are yet not known to us, this does not always work as intended for memory areas larger than one page. We have frequently encountered paging errors in kernelmode although the related memory pages have previously been locked. We therefore introduce a workaround for this issue in the next section.

5.3 Rootkit Loader

To overcome the paging IRQL problem, we have implemented a pre-step in the loading phase. More precisely, in the first stage, we prepare a tiny return-oriented rootkit loader that fits into one memory page and prepares the entry of the actual return-oriented rootkit. It allocates memory from the kernel's non-paged pool, which is definitely never paged out, and copies the rootkit code from userspace before performing a transition to the actual rootkit. This has proven to work reliably in practice and we have not encountered any further IRQL problems. Again, the Rootkit Loader program image resides in userspace, which limits the ability of kernel integrity protection mechanisms to prohibit the loading of our rootkit.

5.4 Rootkit Implementation

To demonstrate our system's capability, we have implemented a return-oriented rootkit that is able to hide certain system processes. This is achieved by an approach similar to the one introduced by Hoglund and Butler [11]: Our rootkit cycles through Windows' internal process block list to search for the process that should be hidden and, if successful, then modifies the pointers in the doubly-linked list accordingly to release the process' block from the list. Since the operating system holds a separate, independent scheduling list, the process will still be running in the system, albeit not being present in the results of process enumeration requests: The process is hidden within Windows and not visible within the Taskmanager. Figure 8 in Appendix A illustrates the rootkit in practice.

```
int ListStartOffset =
  &CurrentProcess->process_list.Flink -
  CurrentProcess;
int ListStart =
  &CurrentProcess->process_list.Flink;
int ListCurrent = *ListStart;
while(ListCurrent != ListStart) {
  struct EPROCESS *NextProcess =
    ListCurrent - ListStartOffset;
  if(RtlCompareMemory(NextProcess->ImageName,
                      "Ghost.exe", 9) == 9) {
    break;
  }
  ListCurrent = *ListCurrent;
}

if(ListCurrent != ListStart) {
  // process found, do some pointer magic
  struct EPROCESS *GhostProcess =
    ListCurrent - ListStartOffset;

  // Current->Blink->Flink = Current->Flink
  GhostProcess->process_list.Blink->Flink =
    GhostProcess->process_list.Flink;

  // Current->Flink->Blink = Current->Blink
  GhostProcess->process_list.Flink->Blink =
    GhostProcess->process_list.Blink;

  // Current->Flink = Current->Blink = Current
  GhostProcess->process_list.Flink =
    ListCurrent;
  GhostProcess->process_list.Blink =
    ListCurrent;
}
```

Figure 7: Rootkit source code snippet in dedicated language for return-oriented programming that can be compiled with our Compiler.

Figure 7 shows an excerpt of the rootkit source code written in our dedicated language. This snippet shows the code for (a) finding the process to be hidden and (b) hiding the process as explained above.

Once the process hiding is finished, the rootkit performs a transition back to the vulnerable code to continue normal execution. This seems to be complicated since we have modified the stack pointer in the first place and must hence restore its original value. However, in practice this turns out to be not problematic since this value is available in the thread environment block that is always located at a fixed memory location. Hence, we reconstruct the stack and jump back to our vulnerable driver. Besides process hiding, arbitrary data-driven attacks can be implemented in the same way: The rootkit needs to

exploit the vulnerability repeatedly in order to gain control and can then execute arbitrary return-oriented programs that perform the desired operation [3].

We would like to mention at this point that more sophisticated rootkit functionality, e.g., file and socket hiding, might demand more powerful constructs, namely *persistent* return-oriented callback routines. Data-only modifications as implemented by our current version of the rootkit hence might not be sufficient in this case. In contrast to Riley et al. [19], we do believe that this is possible in the given environment by the use of specific instruction sequences. However, we have not yet had the time to prove our hypothesis and hence leave this topic up to future work in this area.

The rootkit example works on Windows 2000, Windows Server 2003 and Windows XP (including all service packs). We did not port it to the Vista platform yet as the publicly available information on the Vista kernel is still limited. We also expect problems with the Vista *PatchGuard*, a kernel patch protection system developed by Microsoft to protect x64 editions of Windows against malicious patching of the kernel. However, we would like to stress that PatchGuard runs at the same privilege level as our rootkit and hence could be defeated. In the past, detailed reports showed how to circumvent Vista PatchGuard in different ways [24, 26, 25].

6 Conclusion and Future Work

In this paper we presented the design and implementation of a framework to automate return-oriented programming on commodity operating systems. This system is portable in the sense that the Constructor first enumerates what instruction sequences can be used and then dynamically generates gadgets that perform higher-level operations. The final gadgets are then used by the Compiler to translate the source code of our dedicated programming language into a return-oriented program. The language we implemented resembles the syntax of the C programming language which greatly simplifies developing programs within our framework. We confirmed the portability and universality of our framework by testing the framework on ten different machines, providing deeper insights into the mechanisms and constraints of return-oriented programming. Finally we demonstrated how a return-oriented rootkit can be implemented that circumvents kernel integrity protection systems like NICKLE and SecVisor.

In the future, we want to investigate effective detection techniques for return-oriented rootkits. We also plan to extend the research in two other important directions. First, we plan to examine how the current rootkit can be improved to also support *persistent* kernel modifications. This change is necessary to implement rootkit functions like hiding of files or network connections, which require a persistent return-oriented callback routine. This change would enhance the rootkit beyond the current data-driven attacks. Second, we plan to analyze how the techniques presented in this paper could be used to attack control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms. These mechanisms are orthogonal to the kernel integrity protection mechanisms we covered in this paper.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity – Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[2] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, October 1972. AD-758 206, ESD/AFSC.

[3] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[4] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, October 2008.

[5] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, May 2008.

[7] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[8] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report Cylab Technical Report CMU-CyLab-08-008, CMU, June 2008.

[9] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS)*, February 2003.

[10] Gil Dabah. diStorm64 - The ultimate disassembler library. `http://www.ragestorm.net/distorm`, 2009.

[11] Greg Hoglund and Jamie Butler. *Rootkits : Subverting the Windows Kernel*. Addison-Wesley Professional, July 2005.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[13] Ralf Hund. Listing of gadgets constructed on ten evaluation machines. `http://pi1.informatik.uni-mannheim.de/filepool/projects/return-oriented-rootkit/measurements-ro.tgz`, May 2009.

[14] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[15] Microsoft. Digital Signatures for Kernel Modules on Systems Running Windows Vista. `http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kmsigning.doc`, July 2007.

[16] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. `http://support.microsoft.com/kb/875352`, 2008.

[17] Nergal. The advanced return-into-lib(c) exploits: PaX case study. `http://www.phrack.org/issues.html?issue=58&id=4`, 2001.

[18] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, October 2007.

[19] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[20] Ryan Riley, Xuxian Jiang, and Dongyan Xu. NICKLE: No Instructions Creeping into Kernel Level Executed. `http://friends.cs.purdue.edu/dokuwiki/doku.php?id=nickle`, 2008.

[21] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Techniques. `http://www.suse.de/~krahmer/no-nx.pdf`, September 2005.

[22] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[23] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007.

[24] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, 3, January 2006.

[25] Skywing. PatchGuard Reloaded: A Brief Analysis of PatchGuard 3. *Uninformed*, 8, September 2007.

[26] Skywing. Subverting PatchGuard Version 2. *Uninformed*, 6, January 2007.

[27] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/0063.html`, 1997.

[28] PaX Team. Documentation for the PaX project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

[29] Terence Parr. ANTLR Parser Generator. `http://www.antlr.org`, 2009.

[30] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

A Return-Oriented Rootkit in Practice

Figure 8 depicts the results of an attack using our return-oriented rootkit: The process Ghost.exe (lower left window) is a simple application that periodically prints a status message on the screen. The rootkit (upper left window) first exploits the vulnerability in the driver to start

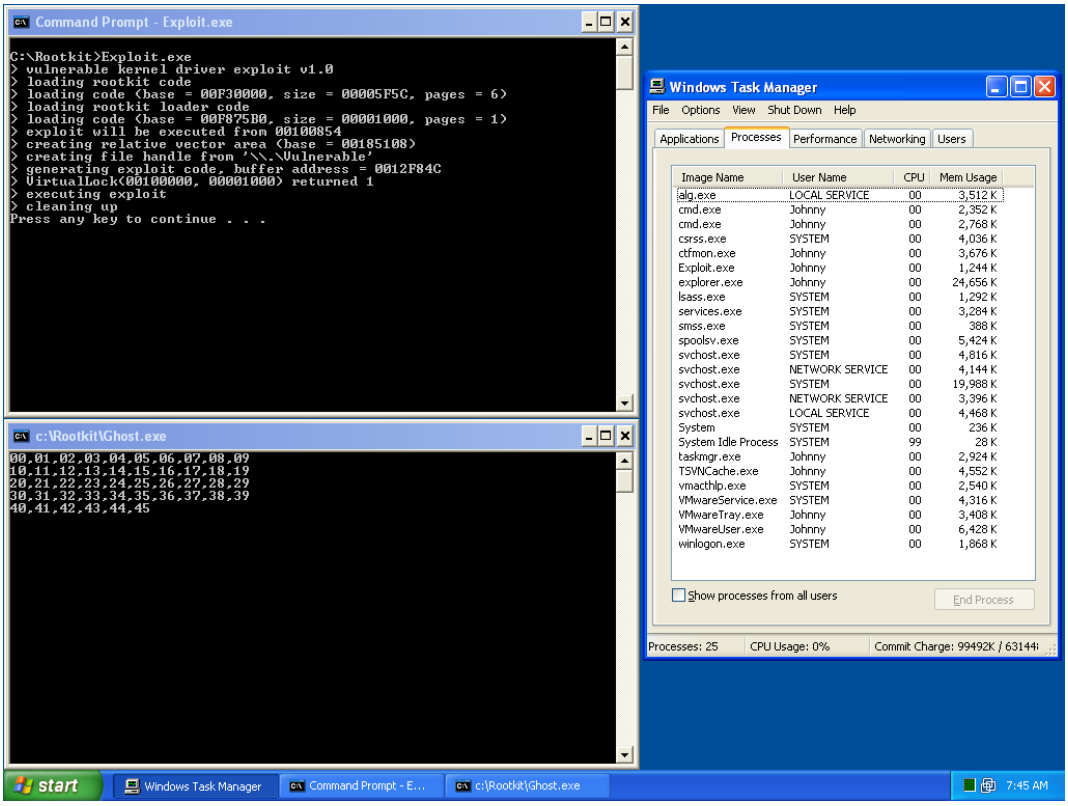

Figure 8: Return-oriented rootkit in practice, hiding the process Ghost.exe.

the return-oriented program. This program then hides the presence of Ghost.exe as explained in Section 5.4: The process Ghost.exe is running, however, the rootkit removed it from the list of running processes and it is not visible in the Taskmanager.

B Return-Oriented QuickSort

The following listing shows an implementation of QuickSort within our dedicated programming language. The syntax is close to the C programming language, allowing a programmer to implement a return-oriented program without too much effort. The most notable exception from C's syntax is related to importing of external functions: Our language can import subroutine calls from other libraries, enabling an easy way to call external functions like `printf()` or `srand()`. However, each function needs to be imported explicitly. Furthermore, the language implements only a basic type-system consisting of integers and character arrays, but this should not pose a limitation.

```
import("msvcrt.dll", printf:cdecl,
                     srand:cdecl,
                     rand:cdecl,
                     malloc:cdecl);

import("kernel32.dll", GetCurrentProcess,
                       TerminateProcess,
                       GetTickCount);

int data;
int size = 500000;

function partition(int left, int right,
                   int pivot_index) {
  int pivot = data[pivot_index];
  int temp = data[pivot_index];
  data[pivot_index] = data[right];
  data[right] = temp;
  int store_index = left;
  int i = left;

  while(i < right) {
    if(data[i] <= pivot) {
      temp = data[i];
      data[i] = data[store_index];
      data[store_index] = temp;
      store_index = store_index + 1;
    }
    i = i + 1;
  }

  temp = data[store_index];
  data[store_index] = data[right];
  data[right] = temp;

  return store_index;
}
```

```
function quicksort(int left, int right) {
  if(left < right) {
    int pivot_index = left;
    pivot_index = partition(left, right,
                            pivot_index);
    quicksort(left, pivot_index - 1);
    quicksort(pivot_index + 1, right);
  }
}

function start() {
  printf("Welcome to ro—QuickSort\n");
  data = malloc(4 * size);
  srand(GetTickCount());
  int i = 0;
  while(i < size) {
    data[i] = rand();
    i = i + 1;
  }

  int time_start = GetTickCount();
  quicksort(0, size - 1);
  int time_end = GetTickCount();
  printf("Sorting completed in %u ms:\n",
        time_end - time_start);
}
```