A Forced Sampled Execution Approach to Kernel Rootkit Identification

Jeffrey Wilhelm and Tzi-cker Chiueh

Core Research Group
Symantec Research Laboratories

{jeffrey_wilhelm, tzi-cker_chiueh}@symantec.com

Abstract. Kernel rootkits are considered one of the most dangerous forms of malware because they reside inside the kernel and can perform the most privileged operations on the compromised machine. Most existing kernel rootkit detection techniques attempt to detect the existence of kernel rootkits, but cannot do much about removing them, other than booting the victim machine from a clean operating system image and configuration. This paper describes the design, implementation and evaluation of a kernel rootkit identification system for the Windows platform called Limbo, which prevents kernel rootkits from entering the kernel by checking the legitimacy of every kernel driver before it is loaded into the operating system. Limbo determines whether a kernel driver is a kernel rootkit based on its binary contents and run-time behavior. To expose the execution behavior of a kernel driver under test, Limbo features a **forced sampled execution** approach to traverse the driver's control flow graph. Through a comprehensive characterization study of current kernel rootkits, we derive a set of run-time features that can best distinguish between legitimate and malicious kernel drivers. Applying a Naive Bayes classification algorithm on this chosen feature set, the first Limbo prototype is able to achieve 96.2% accuracy for a test set of 754 kernel drivers, 311 of which are kernel rootkits.

Keywords: rootkit detection, X86 ISA emulation, dynamic malware analysis, Bayes classifier, and intrusion prevention

1 Introduction

A kernel rootkit [13, 1, 12, 11, 4] is a piece of binary code that a computer intruder, after breaking into a machine, installs into the victim's operating system to perform various malicious functions, including data gathering, hiding processes and files, sending out spam emails, mounting DoS attacks against a chosen target, etc. The majority of kernel rootkits are installed by a separate user-level process, which creeps into the victim machine by exploiting browser weaknesses or application vulnerabilities. In addition, they are mostly installed

into the kernel through a well-defined application program interface (API) that allows legitimate kernel modules to be loaded into the kernel's address space.

Recent threat trend analysis [9] reports that malware authors are increasingly turning to rootkits to establish a permanent, undetectable presence on the systems they compromise. Although it is possible to detect existence of kernel rootkits through in-depth scanning of the kernel address space, this approach is less than ideal because it is extremely difficult to remove kernel rootkits, which can reside anywhere inside the kernel and run at the same privilege level as the security software and operating system. The goal of the Limbo project is develop a real-time kernel rootkit identification system that can pro-actively prevent kernel rootkits from being loaded into Microsoft's Windows operating system.

In contrast to existing kernel rootkit detection systems and products [30, 2, 7, 18, 6, 17, 28, 22, 19], which mostly detect rootkits after they are loaded into the kernel, Limbo takes a *preventive* approach in that it checks whether a kernel driver is a rootkit or not before it is loaded into the kernel, and prohibits the driver from being loaded if that is the case. Because Limbo is required to make this determination in real time, the time budget for this check is quite limited, which makes the kernel rootkit identification problem even more challenging.

Limbo bases its decision on the static attributes and dynamic behaviors of the kernel driver under test. That is, it needs to actually run the kernel driver and monitor its execution. Applying the run-time monitoring approach to kernel rootkit identification involves three technical challenges. First, how to build an emulation environment that can both run arbitrary kernel modules successfully and provide flexible interfaces for recording interesting events? Second, how to collect as many run-time behaviors as possible from a kernel driver without trying all possible inputs or exploring all possible execution paths through the driver? Third, how to extract effective features from a kernel driver's run-time behaviors and use them as the basis for training a classifier that could distinguish between legitimate kernel drivers and malicious kernel rootkits?

Limbo is built on top of PAM32, which is an instruction set architecture emulator designed to run user-level X86 or IA32 binaries. The original PAM32 does not provide an adequate emulation of the Windows kernel for kernel module execution because it does not support kernel-mode instructions, accesses to kernel data structures, or calls to Windows OS's kernel functions. Limbo adds a sufficient amount of kernel emulation into PAM32 so that it can successfully execute a total of over 270 kernel drivers used in the development of this project. Even if a kernel driver can successfully run on an emulator, there is no guarantee that the emulator can extract all the interesting run-time behaviors from the kernel driver, because each execution run most likely only exercises a particular portion of the driver's binary. Limbo solves this problem by *flood emulation* or *forced sampled execution*, which forces a kernel driver's control towards particular paths and strives to exercise each of the driver's basic blocks at least a certain number of times. To solve the final problem, we manually perform a comprehensive characterization of a set of kernel rootkits and legitimate kernel

drivers, use the analysis results to derive a set of distinguishing features based on their binary attributes and run-time behaviors, and then feed them as inputs to a Naive Bayes classifier training tool. Through these three techniques, Limbo is able to correctly identify kernel rootkits and legitimate kernel drivers 96% of the time for a test suite of 754 kernel drivers, while keeping the additional performance overhead under 500 msec.

The rest of this paper is organized as follows. Section 2 reviews previous works related to kernel rootkit detection and identification. Section 3 summarizes the result of our manual behavioral characterization study of kernel rootkits. Section 4 describes the design and implementation of the Limbo kernel rootkit identification system. Section 5 presents the efficiency and effectiveness result of a quantitative performance evaluation study of the first Limbo prototype. Section 6 concludes this paper with a summary of the main research contributions of this work, and directions for future work.

2 Related Work

The fundamental problem underlying kernel rootkit identification is how to determine if a piece of binary code is malicious or not. A simpler version of this problem is how to determine if a program is a semantically equivalent variant of a known malicious program. There are several approaches to this problem. The simplest approach is string comparison as used in anti-virus file scanning products [7, 8], which aims to detect known malware using byte-level signatures extracted from the malware. A slightly more sophisticated approach is to parse binaries into instructions, then extract instruction-level features such as n-gram or n-set [29, 14] and apply standard training algorithms to derive a classifier that can distinguish between benign and malicious binaries based on these features. The third approach is to apply program analysis techniques to binary programs to compute their control-flow graph representation, and determine if a given binary is a variant of a known malware by computing their graph isomorphic distance [27, 10]. For example, Microsoft's Strider Gatekeeper [31] monitors the auto-start extensibility points (ASEPs) to determine if any suspicious software is installed in the machine start-up script. Christodorescu et al. [5] characterize variations of worms in terms of semantically equivalent operations in these malware variants. Kruegel [16] took the same approach to analyze kernel rootkit samples, derived equivalent instruction sequence patterns with the same execution semantics, and used them as the basis to statically identify kernel rootkits. The same group [15] also applied a similar technique to statically analyze a particular class of spyware, Browser Helper Object(BHO)-based spyware that leak information, to detect possible information leaking behavior, essentially a form of binary information flow analysis.

The fourth approach is to run the given binary program, monitor its run-time behavior, and raise an alert if the behavior exhibits a different pattern than those associated with known good code. The is approach potentially can catch an entire class of malware without analyzing the underlying binary code. In general,

this approach may be a good fit for user-level rootkit/spyware detection, but is less effective for kernel rootkit detection because it may lead to corruption of the kernel address space.

Finally, one can determine if a kernel rootkit exists inside the operating system based on its side effects. For example, if the results returned by two different interfaces inside the kernel to the same query (e.g. list of files in this directory) differ, there must be a kernel rootkit that hides processes/files sitting in between these two interfaces [30, 24]. This approach, however, cannot detect kernel root kits that do not hide processes/files. As another example, if certain critical kernel data structures such as the system call dispatch table, interrupt vector table, or the list of active processes are modified, there is a good chance that a kernel rootkit already sneaked into the kernel. Many existing kernel rootkit detection systems/products [21, 6, 7, 18, 19] take the last approach together with signature matching.

The above approaches can be taxonomized according to the type of features they use and the detection algorithms applied to these features. More concretely, features used in rootkit detection or identification can be information statically extracted from the binary code, run-time interaction patterns, or side effects left in the kernel address space, whereas algorithms used to reach a detection or identification decision can be based on similarity match to known malware, finding deviation from known benign code, or data-driven classifier that is trained to distinguish between known malware and known benign code.

The kernel rootkit identification system described in this paper, Limbo, is unique in that it applies a data-driven classifier to run-time behavioral features. Because collecting run-time behaviors of kernel drivers is difficult, no known rootkit detection systems take this approach. Limbo solves this problem by developing a user-level emulator that can effectively stress every part of the input kernel driver. This emulator makes it possible to extract a given kernel driver's run-time behavior without running it inside the kernel, thus preventing kernel rootkits from corrupting the kernel address space.

Moser et al. [20] proposed a malware analysis system that attempts to explore as many execution paths of a piece of malware as possible by computing the inputs required to force the malware's control to take a particular path. Although this technique is more accurate than Limbo's forced sampled execution approach because it ensures that the program state along an execution path is always consistent, its implementation complexity is much higher. Moreover, because Limbo is designed to determine if a kernel driver is legitimate or not in real time, this multi-path exploration technique is too slow to be feasible for Limbo.

3 Behavioral Characterization of Kernel Rootkits

To distinguish legitimate kernel drivers from kernel rootkits, we assembled a set of 73 kernel rootkits, which are collected by Symantec's Security Response between November 2005 and May 2006, and a set of legitimate kernel drivers, which includes 234 kernel drivers installed on a standard Symantec corporate machine

and 27 kernel drivers used in several commercial security software products. For each kernel driver and rootkit, we disassembled and manually reverse-engineered it, and then ran it through Limbo's emulator in such a way that every basic block of its binary code is exercised for a certain number of times. Then we manually analyzed the static attributes and dynamic behaviors of these drivers, and derived a set of features that can best distinguish between legitimate kernel drivers and kernel rootkits. The following presents the set of features resulting from this analysis.

The features are classified into the following seven categories, the first two of which are static attributes derived from a driver's binary whereas the last five are dynamic attributes derived from a driver's run-time behavior. Each member in the feature set represents either a logical flag or an integer count.

3.1 Portable Executable (PE) File Features

- The majority of legitimate kernel drivers contain debug information such as the symbol table, whereas kernel rootkits mostly don't.
- Use of Microsoft's StackGuard buffer overflow protection technology is quite prevalent among legitimate kernel drivers, but is relatively rare among kernel rootkits.

3.2 Import Object Features

- Kernel rootkits tend to have a fewer number of objects in the import table than legitimate kernel drivers. For example, the kernel rootkit, `Apropos.C`, does not have any entries in the import table and imports all the objects it needs at run time rather than at load time.
- Certain import objects occur infrequently in legitimate kernel drivers, but other imports, such as those that manipulate actual hardware, are quite common in legitimate kernel drivers.
- Certain libraries are almost never used in kernel rootkits.
- The number of dynamic imports, i.e., those resolved by `MmGetSystemRoutineAddress`, is higher in kernel rootkits than in legitimate kernel driver. This count does NOT include imports that are dynamically resolved by directly parsing a PE binary's headers within memory.

3.3 Device-Related Features

- Legitimate kernel drivers tend to create fewer virtual devices than kernel rootkits.
- It is more likely for kernel rootkits than for legitimate kernel drivers to create virtual devices that can only be opened exclusively, because kernel rootkits want to ensure that once they open a virtual device, the virtual device cannot be accessed by other entities.

– Each kernel driver is typically attached to a virtual device. It is more likely for a kernel rootkit to attach itself to a critical virtual device, such as the TCP or UDP stack, in order to filter or log data passing through the virtual device.
– When a kernel driver opens the keyboard virtual device, it is more likely to be a kernel rootkit, because this is the common behavior of a kernel-mode key logger.

3.4 Data Structure Access Features

– A Memory Descriptor List (MDL) contains the physical page layout for a contiguous range of the kernel virtual address space. It is more likely for kernel rootkits than for legitimate kernel drivers to allocate an MDL that contains memory allocated to some kernel data structures such as the system service descriptor table (SSDT), and modify these data structures that are normally write-protected.
– It is more likely for kernel rootkits than for legitimate kernel drivers to directly read or write an opaque kernel data structure such as the `EPROCESS` structure. Legitimate kernel drivers rarely access `EPROCESS`, but kernel rootkits modify `EPROCESS` to hide processes, using a technique known as DKOM (Direct Kernel Object Modification) [25].
– To intercept the control transfer of execution paths in the kernel, it is more likely for kernel rootkits than for legitimate kernel drivers to modify various function pointer tables in the kernel, such as the system service descriptor table (SSDT).
– Some kernel rootkits modify the first few bytes of the functions in one of the kernel import libraries (in particular, `ntoskrnl.exe`, `HAL.dll`, and `ndis.sys`) in order to intercept the kernel's control transfer. This technique is known as *in-line hooking*.

3.5 Descriptor Table Features

– Some kernel rootkits use the store IDT (`SIDT`) instruction to read the interrupt descriptor table address in order to tell if they are running under the control of a virtual machine monitor. This technique is popularized by Joanna Rutkowska [23].
– Some kernel rootkits use the load IDT (`LIDT`) instruction to modify the base address of the interrupt descriptor table (IDT), and redirect hardware interrupts to a completely different set of interrupt service routines.
– Some kernel rootkits use the store GDT (`SGDT`) instruction to read the global descriptor table's base address, or use the load GDT (`LGDT`) instruction to set the global descriptor table's base address.

3.6 Miscellaneous Features

– Some kernel rootkits modify the `IA32_SYSENTER_EIP` model specific register, which contains the address of the user to kernel mode system call handler on newer x86 hardware. No legitimate kernel drivers modify this register.

– Kernel rootkits often open themselves or their user-mode components in *exclusive* mode so as to prevent security software from accessing these files.
– Kernel rootkits are more likely to obfuscate their code than legitimate kernel drivers, which almost never use any obfuscation.

4 Design and Implementation of Limbo

4.1 X86 and Windows Kernel Emulator

To successfully execute kernel drivers, Limbo needs an emulator that can interpret all X86 instructions and support library and kernel functions that these drivers are likely to call. PAM32 is an X86 ISA emulator for user-level Windows applications. We chose PAM32 because it is a proven tool that has been used for years inside Symantec for a wide variety of projects ranging from malware reverse engineering to threat signature creation, and comes with a set of useful utilities for deriving static/dynamic characteristics of Windows binary programs. PAM32 interprets each instruction in a Windows binary one by one, and enables collection of various run-time statistics, such as instruction frequency histogram and counting of devious instruction sequences such as "return to a return instruction", depending on the configuration parameter setting when it is launched. Unfortunately the original version of PAM32 does not support calls to internal kernel functions, and therefore cannot execute kernel drivers that do make such calls. To support kernel driver execution, we make the following modifications to PAM32:

– Support for privileged instructions such as I/O instructions and modifications to control registers.
– Emulation for over 90 Windows kernel functions that our test kernel drivers rely on for correct functioning.
– Support for accesses to kernel data structures, such as KeServiceDescriptorTable, including their creation, initialization and emulation.
– Support for flooded emulation, which is designed to discover as many run-time behaviors of the kernel driver under test as possible by forcing the driver's control along both arms of every encountered conditional branch.
– Collection of 32 run-time features, each of which corresponds to a binary attribute or an execution behavior that potentially can distinguish kernel rootkits from legitimate kernel drivers, as described in Section 3.

Although the original PAM32 engine is compiled as C++, the actual code is written entirely in C. For ease in prototyping, the extensions to PAM32 have been written as C++ classes. In addition, we make use of STL container classes to hold critical data.

Currently PAM32 emulates only a subset of kernel functions in the Windows operating system because the emulation routines are developed manually. An open research question is whether it is possible to automate the process of adding kernel function emulation to emulators such as PAM32. To emulate a new

kernel function, we fist identify the missing kernel function and the associated DLL, e.g., `PsSetLoadImageNotifyRoutine` in `ntoskrnl.exe`. Then we look up in the Microsoft DDK to identify the input arguments, the return value, and the calling convention of the missing kernel function. Next, we add the missing function to the export list, which is used to resolve the import table of a kernel driver when it is loaded into Limbo's emulator. The most challenging part in emulating a kernel function is to implement the logic underlying the emulated kernel function. Inside the emulator, each emulated kernel function is implemented as a case statement that is labeled with the name of the emulated kernel function. Each such statement consists of three components: reading input arguments, setting the return variable with a value of proper type, and clearing up the stack according to the function's call convention. The following example shows how the kernel function `NTOSPsSetLoadImageNotifyRoutine` is emulated in Limbo's emulator:

```
case NTOSPsSetLoadImageNotifyRoutine:
{
    // Read arguments
    DWORD loadImageRoutine =
           PAM32ReadStack(hLocal, 4);

    .......

    // Set the return value
    SET_RETURN_VAL(NTSTATUS_SUCCESS);

    // Clean the stack
    ReturnFromApi(hLocal, 4);
    break;
}
```

4.2 Forced Sampled Execution

Because Limbo tests a kernel driver's legitimacy based on its run-time behavior, it is essential that it explore as many execution paths through the driver as possible. In theory, to derive proper inputs that force a particular execution path of a binary code requires solving a system of constraints derived from the code's logic. Therefore, to truly explore all possible paths of a kernel driver requires solving a potentially exponential number of systems of constraints, and is thus computationally infeasible for Limbo given its real-time constraint. To "force out" a binary code's execution behavior without incurring expensive constraint solving computation, Limbo applies a technique called *flood emulation*, which was originally developed to detect heuristically detect virus.

Flood emulation is an example of *forced sampled execution*, and makes two approximations to the ideal of fully exploring a program's all possible paths. First, instead of computing inputs that can drive a program's control along a particular path, flood emulation simply forces the program counter (PC) of the

emulator to a certain value so that the program's control can go to a specific location. Note that this change of PC value is not part of the program's underlying logic, and may actually result in an impossible path, i.e., an execution path through a program that should never take place according to the program's logic. Second, to avoid exploring an exponential number of execution paths of a kernel driver under test, flood emulation uses a context-independent sampling approach to traverse the control flow graph of the kernel driver under test.

As Limbo's emulator interprets the kernel driver under test instruction by instruction, it breaks the driver code into a series of basic blocks, and ensures that each encountered basic block is executed at least once, but no more than N times, where N is a configurable parameter. Whenever Limbo's emulator encounters a conditional branch instruction, it uses the following algorithm to determine how to proceed next:

- If the actual destination block of the conditional branch, which could be its target or fall-through arm, has not yet reached its execution iteration limit (N), the emulator continues with the destination block. In addition, if the non-destination arm has never been discovered, the emulator saves the CPU state, the current stack and the entry point of the non-target block in a special stack called the *branch stack* for later exploration.
- If the actual destination arm of the conditional branch has reached its execution iteration limit and the non-destination arm has NEVER been discovered, the emulator forces the driver's control to the non-destination arm by setting the PC accordingly.
- Otherwise, Limbo's emulator pops the top-most item on the branch stack, restores the emulator state accordingly and starts executing the associated block of instructions.

Essentially flood emulation traverses the program's control flow graph in a depth-first fashion, back-tracking the traversal only when the last block's execution iteration limit is exceeded. In addition to per-block execution iteration limits, Limbo's emulator also limits the total number of instructions emulated to ensure that the total driver legitimacy test time is bounded. This limit is on the order of millions of instructions in the current Limbo prototype. Under this traversal algorithm, the main reason that flood emulation may fail to exercise a test program's basic block is that the emulator never has a chance to discover the basic block before the total instruction count limit is reached. In practice, this does not appear to be a problem because the product of the number of basic blocks in a program and the per-block execution iteration limit is typically smaller than the total instruction count limit.

The depth-first traversal scheme may fail to expose interesting behaviors of a basic block because it never has a chance to be executed at all or because it never has a chance to be executed under a context in which interesting behaviors occur. For example, for a function involved in a recursive function call chain, it is possible that the execution iteration limits of this function's basic blocks are used up in the beginning of the emulation run; as a result the function never has a chance to be executed in a context where it is called from another function, which

will pass an interesting function pointer as an input argument. To address this problem, the sampled execution strategy should spend the execution iteration limit of each basic block more intelligently, so that it can exercise each basic block in as many distinct contexts as possible. One way to sample the control flow graph is to limit the number of times at which a basic block is executed when it is under the same sequence of last K stack frames. The current Limbo prototype, however, uses the simplest sampling strategy: execute every encountered basic block as many times as reaching its execution iteration limit.

The other limitation of the current Limbo emulator is that it restores only the CPU and stack state when back-tracking a previous basic block. When Limbo's emulator forces the control to go to a particular arm of a conditional branch, it does not attempt to adjust the values of the associated control variables so that their values are consistent with the fact that the control goes to the chosen arm. Neither does it "undo" side effects left by another arm. Consequently, in many cases, the Limbo emulator is actually executing impossible paths in the test program. Although this seemingly simplistic approach may appear illogical at first sight, it actually could effectively expose a test binary's instruction-level behaviors that are useful for malware identification, as demonstrated by the empirical results shown in Section 5. The reason is that our interest here is to get a glimpse of the test binary's run-time behavior rather than to faithfully trace the binary's control flow.

When a kernel driver under test is loaded into Limbo's emulator, the emulator initializes driver-specific variables, the emulator state, the feature set, and the state of emulated kernel structures. During loading, the emulator maps the driver's PE sections to the emulated memory space, and resolves its import objects. If the driver has an import that is not yet supported by the emulator, it will still be resolved with a dummy address. The emulator executes the driver by decoding and executing each instruction in software, and leaving results of the instruction on the emulated CPU and memory state. If the instruction pointer refers to an emulated kernel function, the emulator emulates the function call and returns a result back to the driver. During an emulation run, the emulator continuously records features, the extraction of which could be triggered by specific instructions, specific kernel function calls, accesses to certain kernel memory areas, or other conditions. An emulation run terminates when the total instruction count limit is reached, when the control returns from the main entry point of the driver under test, or when the branch stack used in flood emulation becomes empty. After an emulation run is completed, the emulator extracts several additional features via post-processing, and cleans up the emulator state.

4.3 Classifier Training

The current Limbo prototype uses a total of 32 features in its kernel rootkit detection algorithm. The majority of them correspond to the static and dynamic attributes that we associate with kernel rootkits in the behavioral characterization study, as described in Section 3. Each of these features is either a binary flag

that indicates whether a particular attribute is present or a counter of the number of certain dynamic events that are characteristic of existing kernel rootkits. The set of kernel drivers used in classifier training is the same as those used in the manual characterization study discussed in Section 3. It consists of 73 kernel rootkits and 261 legitimate kernel drivers. We call this set the *training* kernel driver set.

We used a Naive Bayes classifier [32] training tool to output a binary classifier that can determine if an unknown kernel driver is a kernel rootkit or not based on the 32 features extracted from an emulation run of the driver. More concretely, this classifier training algorithm assumes that the 32 features are independent of one another, and computes from the training set the conditional probability that a particular feature assumes a certain value when the underlying binary is a kernel rootkit or a legitimate kernel driver. With these conditional probability distributions, the resulting classifier can determine an unknown kernel driver's legitimacy by computing the probability that it is a kernel rootkit or a legitimate kernel driver, and rendering the final verdict with the classification with a higher probability.

Outcome	Accuracy
True Positive	90.4% (66/73)
False Negative	9.6% (7/73)
True Negative	98.1% (256/261)
False Positive	1.9% (5/261)
Overall Accuracy	96.4% (322/334)

Table 1. *The classification accuracy of applying the Naive Bayes classifier trained with the* training *kernel driver set on the same kernel driver set.*

5 Effectiveness and Efficiency Evaluation

To evaluate the effectiveness of Limbo's forced sample execution approach to kernel rootkit identification, we run each driver in the *training* kernel driver set through Limbo's PAM32 emulator to collect the corresponding feature set, and feed these feature sets into a standard Naive Bayes classifier training tool. While running samples in this training set, we make sure each of them runs to the total instruction count limit by adding emulation support for whatever kernel functions, kernel data structures and privilege instructions these samples happen to need. After iteratively hand-tuning the training parameters, the classifier training tool produces a 2-category classifier that maximizes the margin between the two categories.

To confirm the validity of Limbo's approach to real-time kernel rootkit identification, we first applied the classifier derived from the training kernel driver

set back to the same kernel driver set. The classification results under 10-fold cross validation are shown in Table 1. Because a smaller false positive rate is more important than a smaller false negative rate, the classifier is hand-tuned to reduce the false positive rate (1.9%) even at the expense of false negative rate (9.6%). A false negative means that the classifier mistakes a kernel rootkit for a legitimate kernel driver. Despite the relatively high false negative rate, the overall accuracy of the resulting classifier is high at 96.4%, which means that 96.4% of the kernel drivers tested are correctly classified.

Outcome	Accuracy
True Positive	92.6% (288/311)
False Negative	7.3% (23/311)
True Negative	98.6% (437/443)
False Positive	1.4% (6/443)
Overall Accuracy	96.2% (725/754)

Table 2. *The classification accuracy of applying the classifier trained with the* training *kernel driver set on the* evaluation *kernel driver set.*

Because Limbo's emulator is carefully tuned to run each sample in the first kernel driver set successfully, the feature sets extracted are more complete. However, when used in the field, there is no guarantee that Limbo's emulator can run each kernel driver under test to its total instruction count limit. To gauge the effects of incomplete emulation and the effectiveness of Limbo's classifier, we collect a second set of kernel drivers for which Limbo's emulator and classifier have not been specifically tuned. In this set, called the *evaluation* kernel driver set, the known kernel rootkits are those collected by Symantec Security Response between January 2005 and May 2006, for a total of 311 samples; the known legitimate kernel drivers are taken from 5 desktop machines in Symantec Research Labs and the same 27 Symantec's own kernel drivers in the training set, for a total of 443 samples.

Again, we ran each kernel driver in the *evaluation* kernel driver set through Limbo's emulator to completion or termination, and calculated the corresponding set of static/dynamic features. The results of applying the classifier derived from the *training* kernel driver set to the *evaluation* kernel driver set under 10-fold cross validation are shown in Table 2. Surprisingly, the overall accuracy only degraded slightly, from 96.4% to 96.2%. After examining the detailed breakdown, we found that both the false negative rate and the false positive rate are actually decreased, a strong indication that Limbo's approach is relatively robust. The fact that the overall classification accuracy remains practically the same suggests that Limbo's ability to sample execution behaviors, choice of features, and classification algorithm make a promising base for building future kernel rootkit detection technology.

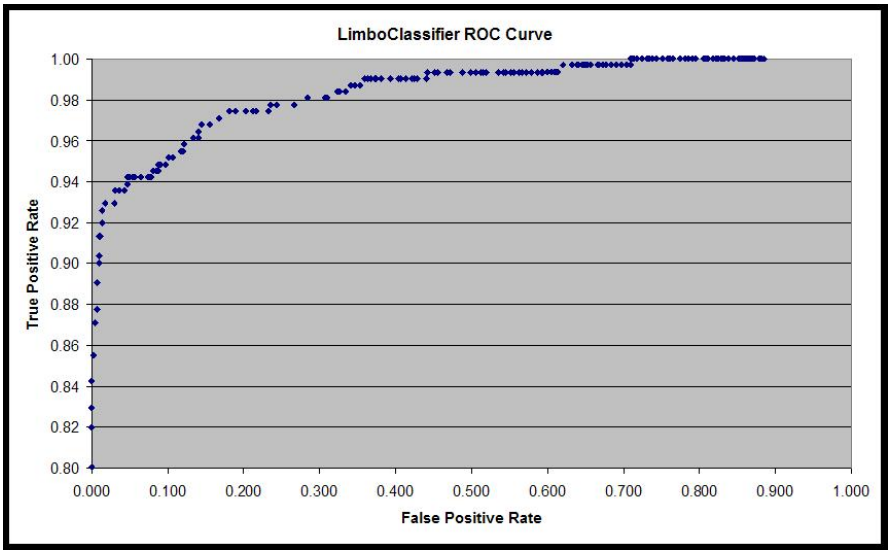

Fig. 1. The receiver operating characteristics curve of the Limbo classifier under the *evaluation* kernel driver set.

To evaluate the trade-off between sensitivity and specificity, we plot the receiver operating characteristics (ROC) curve of Limbo's classifier when varying threshold values used in the classifier training process, The ROC curve for the *evaluation* kernel driver set, as shown in Figure 1 shows the trade-off between the classifier's true positive rate and false positive rate. It allows us to interactively hand-tune the resulting classifier until it strikes the most desirable tradeoff between these two metrics.

Finally, to further gauge how well Limbo is able to cope with rootkits appearing in the future, which might use new techniques to evade the emulator or to perform more similarly to legitimate kernel drivers, we collected a third kernel driver set, which corresponds to malicious kernel rootkits submitted to Symantec during June 2006 (a total of 69 instances). These drivers were then classified using a classifier that is trained on the *evaluation* kernel driver set described above (which contained drivers up to May 2006). The classification results are shown in Table 3. Again, these results suggest that Limbo has the potential to catch entirely new threats without requiring frequent retraining or additional tuning.

False positives and negatives could arise either because Limbo's feature set is not perfect in distinguishing between legitimate kernel drivers and malicious kernel rootkits, or because Limbo cannot always extract the required features from the test kernel drivers. An in-depth analysis of the false positives and negatives from the above experiments reveals that their root cause is Limbo's limited feature extraction capability. For example, we added to Limbo's emulator the emulation of five more kernel functions in order to extract all dynamic features

Outcome	Accuracy
True Positive	97.1% (67/69)
False Negative	2.9% (2/69)

Table 3. *The classification results of applying a classifier trained on the* evaluation *kernel driver set on a set of kernel rootkits that appeared temporally after the* evaluation *kernel driver set.*

from the two kernel rootkits in Table 3 that Limbo previously mis-classified. After successfully extracting their features, Limbo could indeed correctly recognize them as kernel rootkits without any retraining, i.e., zero false negative! In addition, we applied the new emulator to the false positives and negatives in Table 2 to re-extract their features, re-ran the same classifier on these features, and improved the total accuracy rate from 96.2% to 98.5%.

Because Limbo checks each kernel driver before it is to be loaded into the operating system, it introduces additional delay in the driver loading process. The time taken to determine if a new kernel driver is legitimate or not mostly depends on the total instruction count limit, because most of the time goes to extraction of the driver's feature set by executing it in the emulator. The current Limbo emulator executes instructions about 100 times slower than when they are executed on the same hardware natively. When the total instruction count limit is set to 10 million, Limbo is able to consistently complete the feature extraction and driver classification process under 500 msec on a 2.4GHz, 2GB RAM Pentium-4 machine running Windows XP Professional. This level of performance is considered reasonable for most interactive users. Note that Limbo only checks the legitimacy of *unknown* kernel drivers when they are about to be loaded into the kernel. As a consequence, it does not affect the system start-up time because most if not all of the kernel drivers loaded at system start-up have gone through legitimacy checks and thus are considered *known*. Finally, Limbo could further incorporate a white-listing mechanism to avoid checking signed kernel drivers.

6 Attack Analysis

The current Limbo prototype has several limitations, most of which are related to its emulation fidelity. As with all emulators, it is impossible to emulate all features of an operating system, processor, and runtime environment. For example, different processors have different instruction sets. Different machines have different hardware configurations. The inability to completely emulate these items enables attackers to evade an emulation-based system. However, we believe evasion techniques that exploit holes in emulators will become less effective as system emulation technologies improve and techniques that detect evasion attempts advance.

Traditional binary obfuscation techniques designed to defeat signature-based AV scanning software are less effective against Limbo, because Limbo relies more on run-time behaviors than on static instruction sequences. Behavior-level obfuscation also seems difficult, because the run-time behaviors of most legitimate kernel drivers follow well-defined patterns and show little variety. Despite this, we recommend re-training Limbo's classifier periodically so that it can upgrade its distinguishing features in accordance to emerging kernel drivers and rootkits.

7 Conclusion and Future Work

Rootkit identification is challenging because fundamentally it requires one to solve the problem of determining if a given piece of binary code is malicious or not based on its static attributes and/or dynamic behaviors. Kernel rootkit identification is a more difficult problem because reliably extracting a kernel driver's run-time behavior is a significant technical challenge. The goal of the Limbo project is even more formidable: perform kernel rootkit identification in real time, before each kernel driver is to be loaded into the Windows operating system. This paper describes the design, implementation and evaluation of the first known real-time kernel rootkit identification system that achieves high identification accuracy for the current generation of kernel rootkits active in the wild. The Limbo technology, as described in this paper, is scheduled to go into all of Symantec's Norton Security products. More concretely, the research contributions of this work include

- The first comprehensive characterization of the run-time behaviors of the current generation of kernel rootkits, and a set of rootkit identification features based on these behaviors,
- A forced sampled execution approach to extract the run-time behaviors of kernel rootkits that is simple and effective, and
- A fully operational prototype that successfully demonstrates its ability to pro-actively identify kernel rootkits before they are loaded into the kernel in real time.

There are several directions we plan to pursue to further improve the identification accuracy of the Limbo system. First, future kernel rootkits are likely to follow the foot steps of computer viruses by incorporating logic to break emulators. To address this problem, we plan to leverage virtual machine technology to improve the emulation fidelity of Limbo's emulator, particularly in kernel function calls, kernel data structure accesses and privileged instruction execution. Second, the way the current Limbo prototype samples the control flow graph of the kernel driver under test does not make the best use of the per-block execution iteration limit, in the sense that it does not attempt to cover as many contexts for a given basic block as possible. We are working on a more intelligent sampled execution strategy so as to expose more varieties of run-time behaviors from the driver under test. Third, we plan to classify kernel rootkits into categories according to their functionalities and run-time behaviors, and apply this high-level

category information to the training of the classification algorithm to further improve the accuracy of kernel rootkit identification. Finally, it will be interesting to apply the same methodology to user-level rootkit or spyware identification and see how effective it is. This requires a separate behavioral characterization effort to deduce a set of features that can best distinguish between legitimate and malicious binaries. Because the number of possible behaviors is significantly larger, the amount of effort that such a characterization study requires is expected to be much higher.

References

1. Nancy Altholz and Larry Stevenson. *Rootkits for Dummies*. John Wiley and Sons Ltd., 2006.
2. Avira. Avira rootkit detection. `http://www.antirootkit.com/software/Avira-Rootkit-Detection.htm`.
3. James Butler. Vice - catch the hookers! In *Conference Proceedings of Black Hat 2004*, July 2004.
4. James Butler and Sherri Sparks. Raising the bar for windows rootkit detection. In *Phrack 63*, July 2005.
5. M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2005.
6. Bryce Cogswell and Mark Russinovich. Rootkitrevealer v1.71. `http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx`, November 2006.
7. F-Secure Corporation. F-secure blacklight rootkit elimination technology. `http://securityticker.blogspot.com/2006/05/f-secure-backlight.html`.
8. Symantec Corporation. Norton antivirus. `http://www.symantec.com/home_homeoffice/products/overview.jsp?pcid=is&pvid=nav2006`.
9. Symantec Corporation. Internet security threat report. `http://www.symantec.com/enterprise/threatreport/index.jsp`, September 2006.
10. Halvar Flake. Automated reverse engineering. In *Proceedings of Black Hat 2004*, July 2004.
11. Fuzen. Fu rootkit. `http://www.rootkit.com/project.php?id=12`.
12. Greg Hoglund and Jamie Butler. The companion website of the rootkit book. `http://www.rootkit.com`.
13. Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
14. M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. In *European Research Journal of Computer Virology*, 2005.
15. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection1. In *Proceedings of Usenix Security Symposium*, 2006.
16. C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2004.
17. McAfee Avert Labs. Rootkit detective. `http://vil.nai.com/vil/stinger/`.
18. Brian Livingston. Icesword author speaks out on rootkits. `http://itmanagement.earthweb.com/columns/executive_tech/article.php/3512621`.

19. Trend Micro. Rootkitbuster. http://www.trendmicro.com/download/rbuster.asp.
20. Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*, 2007.
21. N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of Usenix Security Symposium*, August 2004.
22. Panda Research. Rootkit cleaner. http://research.pandasoftware.com/blogs/research/archive/2006/12/14/Rootkit-cleaner.aspx.
23. Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.invisiblethings.org/papers/redpill.html.
24. Joanna Rutkowska. Thoughts about cross-view based rootkit detection. In *http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf*, June 2005.
25. Joanna Rutkowska. Rootkits detection on windows systems. In *Proceedings of ITUnderground Conference 2004*, October 2004.
26. Joanna Rutkowska. System virginity verifier: Defining the roadmap for malware detection on windows systems. In *http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt*, September 2005.
27. Todd Sabin. Comparing binaries with graph isomorphisms. http://www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm.
28. Sophos. Sophos anti-rootkit. http://www.sophos.com/products/free-tools/sophos-anti-rootkit.html.
29. Mark Stamp and W. Wong. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3), December 2006.
30. Y. Wang, D. Beck, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN-DCCS)*, June 2005.
31. Y. Wang, R. Roussev, C. Verbowski, A. Johnson, M. Wu, Y. Huang, and S. Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *Proceedings of Usenix Large Installation System Administration Conference (LISA)*, 2004.
32. Wikipedia. Naive bayes classifier. http://en.wikipedia.org/wiki/Naive_Bayes_classifier.