DRIP: A Framework for Purifying Trojaned Kernel Drivers

Zhongshu Gu, William N. Sumner, Zhui Deng, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS
Purdue University
West Lafayette, IN, USA, 47907-2107
{gu16, wsumner, deng14, xyzhang, dxu}@cs.purdue.edu

***Abstract*—Kernel drivers are usually provided in the form of loadable kernel extensions, which can be loaded/unloaded dynamically at runtime and execute with the same privilege as the core operating system kernel. The unrestricted security access from the drivers to the kernel is nevertheless a double-edged sword that makes them susceptible targets of trojan attacks. Given a benign driver, it is now easy to implant malicious logic with existing hacking tools. Once implanted, such malicious logic is difficult to detect.**

In this paper we propose DRIP, a framework for detecting and eliminating malicious logic embedded in a kernel driver through iteratively eliminating unnecessary kernel API invocations from the driver. When provided with the binary of a trojaned driver, DRIP generates a purified driver with benign functionalities preserved and malicious ones eliminated. Our evaluation shows that DRIP successfully eliminates malicious effects of trojaned drivers in the system, with the purified drivers maintaining or even improving their performance over the trojaned drivers.

Keywords*—*System Security; Kernel Drivers; Trojan Detection;

I. Introduction

In state-of-the-art design of commodity operating systems, drivers usually take the form of loadable kernel extensions. Privileged users could load them dynamically to support new devices or extend functionalities of a base kernel at runtime. They hide the complexity of interacting with hardware devices and present a neat abstract interface for other kernel components. To achieve these properties, drivers execute with the same privilege as the OS kernel, which makes them susceptible targets of malicious attacks. Unlike the kernel, which is either built by trusted companies or with source code opened to the public, kernel drivers could be provided by third-party vendors as a binary blob.

Given a binary driver, it is difficult to tell whether malicious logic is embedded inside it. From customers' perspectives, it may work correctly with no suspicious symptoms, but the embedded malicious code [1], [2] may have already collected confidential information and cloaked its fingerprint under the cover of a legitimate driver. Even if we assume that vendors only perform the functionalities as they claim, there still exist many binary driver infection techniques [3]–[8] that could implant malicious logic into benign drivers and transform them into trojaned drivers. When the trojaned driver is loaded into an operating system, the hidden malicious code can be loaded simultaneously with the benign code. Hence the challenge is: How can we identify malicious/undesirable logic in the driver and eliminate it at binary level without impairing driver's normal operations?

Existing research efforts to protect device drivers can be divided into two categories, online monitoring and offline profiling. Online approaches [9]–[12] were proposed to isolate the driver in a protection domain and enforce external runtime checks on its execution. They either cannot target intentionally malicious drivers or require protection from the underlying hypervisor. All of them add non-trivial performance overhead due to the realtime monitoring. Offline approaches [13], [14] are designed to exercise the driver during testing to find bugs and vulnerabilities, but they are still incapable of distilling benign operations and eliminating malicious behaviors in the driver.

We develop a system called DRIP[1] to address this problem from a different angle. Based on our observation, we find that malicious/undesirable logic embedded inside many trojaned kernel drivers is orthogonal to drivers' normal functionalities and most such logic achieves malicious effects through interacting with the base kernel through *kernel API invocations.* Removing these interactions in malicious code will not affect the correct execution of the driver and it can also neutralize the malicious behavior. We leverage test suites for the semantic-level behavior of applications [15]–[17] in order to ensure that the driver works correctly when used by those applications. By testing the different application level behaviors, we simultaneously test and ensure all of the underlying benign driver functionality that applications use.

We record interactions between a subject driver and the kernel during testing. Then we try to select and remove a subset of driver-kernel interactions to test whether this removal operation will violate the correct execution of the test suite. We iterate this testing process until all unnecessary interactions are removed, and consequently we can generate a purified driver with malicious/undesirable behaviors removed.

This paper makes the following contributions:

1) A testing approach for differentiating between benign and malicious logic of a trojaned driver. DRIP only requires a high-level test suite to cover and retain core legitimate functionalities of the driver.
2) A Test-and-Reduce algorithm to incrementally reduce unnecessary kernel-driver interactions and extract a

[1]DRIP stands for "DRIver Purifier"

978-1-4799-0181-4/13/$31.00 ©2013 IEEE

minimal subset to ensure the correct execution of the driver.

3) A clustering mechanism to group kernel-driver interactions according to current execution context. It provides additional semantic information to speed up the removal of kernel API invocations in the Test-and-Reduce algorithm.

The rest of this paper is organized as follows. Section II presents the motivation and overview of the DRIP framework. Section III provides the detailed design of DRIP. Section IV gives functional studies of some representative cases and evaluates the performance. Section V discusses the limitations and future work. Section VI describes related work and we conclude in Section VII.

II. Overview of DRIP Framework

A. Goals and Assumptions

The goal of DRIP is to purify a device driver with malicious/undesirable logic embedded that may jeopardize the base kernel. The newly generated driver should have the benign functionalities of a vanilla driver with malicious effects eliminated.

Our approach is based on the assumption that the trojaned driver includes the functionalities of a benign driver. The malicious logic is parasitically attached to the benign logic within the driver's binary and executes persistently when the driver is loaded. We do not target time-bomb malware in which the malicious functions can only be triggered at a specific time because the malicious logic may not be active during our testing. This problem can be addressed by using symbolic execution [18] to cover more execution paths. There are some existing efforts [14], [19], [20] to apply symbolic execution to driver testing and we can leverage them to complement our work. In addition we do not target the malicious code that interacts with kernel through direct memory manipulation. We could consider kernel memory accesses as part of driver-kernel interactions and plan to include this feature in our future work.

We assume that a test suite is available that covers the higher level behaviors of a specific application. As previously mentioned, testing those behaviors also means that the test suite covers the necessary driver functionality that they depend upon. Because we test the application level behaviors, our technique ensures that the application continues to behave correctly with the purified driver. This assumption is reasonable for current software development processes, in which developers often create test cases from requirements even before implementation as part of the design phase. We can also leverage existing test generation techniques [18], [20], [21] to automatically synthesize test cases.

B. Approach Overview

For a particular application and the environment in which it executes, we need to ensure that the application continues to behave correctly. This includes correctly executing any low-level behaviors in the driver that the application relies upon and triggers during its operation. We can do this by treating the driver like a black box, without considering the specifics of its implementation. For example, we might examine a network interface controller (NIC) driver. We can cover the functionality of an FTP server through test cases from curl-loader [17]. If we can ensure the correct execution of curl-loader when using a purified NIC driver, then we have empirically preserved the functionalities of the driver needed by curl-loader. In general, covering the tests of an application will also cover and preserve the low level driver functionality necessary for that application.

Based on our experience of analyzing conventional rootkits, we gain the insight that the common goals of malicious code in kernel space are to retrieve information from base kernel and manipulate kernel data to hide footprints of user space malware. It is difficult to generate a completely self-contained malicious module to achieve all these effects without invoking kernel APIs. When we face a trojaned kernel driver, the execution of malicious code is mixed with the execution of benign code at runtime. Benign code of the driver will also invoke kernel APIs to request services from base kernel. So we need to differentiate benign kernel API invocations from malicious ones. With the availability of a test suite covering benign functionalities of the driver, we can iteratively eliminate some of the kernel API invocations at runtime to test whether it will violate the correct execution of the test suite. If the removal will not affect the benign behavior, we consider these invocations unnecessary, and they can be removed from the binary.

Based on this observation, we first take a snapshot of the system and execute the test suite from a deterministic state. We record all kernel API invocations from the driver to the kernel during testing, which can be captured as control flow transitions across the boundary of driver's loading memory region. Then we try to restore to the snapshot, remove a subset of these invocations in memory, and re-execute the same test suite to test whether the removal will affect its correct execution. We chop the removal set of invocations iteratively until all the invocations left are critical to the correct execution of the driver. Because benign functionalities of the driver are covered by the test suite, the removal of kernel API invocations within benign code will fail the test suite, so we consider them critical and preserve them. On the other hand, because malicious code embedded is either orthogonal or complementary to core functionalities of its "host" driver, removal of invocations within malicious code will not violate the correct execution of the test suite, and they are considered unnecessary. Finally we can generate a purified driver with all the unnecessary invocations removed, and the malicious effects from driver are eliminated concomitantly.

C. Procedure Overview

Figure 1 depicts the overall workflow of DRIP to demonstrate how to purify a trojaned driver. We divide the whole procedure into three phases, i.e., profiling, testing and rewriting, as in Figure 1(a). These three phases are transparent to each other. We give a brief description of the specific functionality of each phase first and will elaborate upon them in the following section.

Before starting the purifying process, we construct the *Testing Environment* in Figure 1(b) and prepare the binary file of the trojaned driver. In the profiling phase, we execute the test suite to trigger the execution of this driver, record kernel

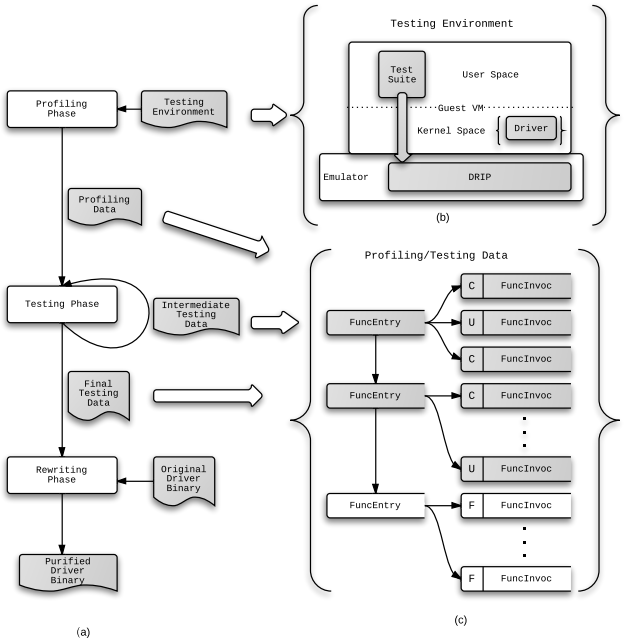

Fig. 1: Workflow of DRIP

API invocations, and cluster them according to their execution context. The output of this phase is the *Profiling Data* and it is organized in the structure presented in Figure 1(c). In the testing phase, we select and remove a subset of these kernel API invocations and test their influence on the correct execution of the test suite. The *Testing Data* shares the same structure as the *Profiling Data*. The only difference is that we mark testing status on every entry in the *Testing Data*. For example, in Figure 1(c), shaded entries in *Testing Data* indicate that they have been tested. This is an iterative process, and we feed the *Intermediate Testing Data* back as input to the testing phase. The testing phase terminates when all the entries in the *Testing Data* have been tested. In the last rewriting phase, we summarize the testing result, apply the changes on the trojaned binary, and generate a purified driver.

III. Detailed Design

In this section, we describe DRIP following the workflow of the driver purification procedure and discuss the design of key DRIP components in detail. First, we describe the setup of the *Testing Environment*. Then we demonstrate the profiling, testing and rewriting phases respectively to explain the procedure of generating a purified driver. Finally, we present the technical details of the prototype implementation.

A. Environment Setup

Before the purification procedure, we set up the *Testing Environment*, prepare the trojaned driver and design a communication channel to send test statuses from the test suite to DRIP.

Testing Environment: As shown in Figure 1(b), the *Testing Environment* consists of a guest virtual machine (VM) and its underlying emulator (we use QEMU [22] in our environment) as the analysis platform. We integrate our DRIP system as a component into the emulator. In the guest VM, we load the trojaned driver in the kernel space and monitor the code execution within its loading memory region. We select or synthesize an automated test suite for the target application to cover the benign behavior of the subject driver and launch it in the user space. In order to ensure that the test suite executes from a deterministic state, we take a snapshot of the VM at the time right before the test suite is about to run.

Communication Channel: If we pick up an existing test suite, it would have no knowledge about the underlying system including DRIP. However DRIP needs to make decisions based on the current status of the test suite. So we design a communication channel between the test suite and DRIP. We can leverage special instructions like *hypercall* or *cpuid*, to send signals to the underlying emulator. The emulator can extract signals when translating these instructions. We design 3 signals, *TESTON*, *TESTSUCC*, and *TESTFAIL*, which respectively stand for the beginning of the test, the end of the test with a successful result, and the end of the test with a failing result. Then we embed the communication channel in the test suite to send these signals at specific time instances.

B. Profiling Phase

In the profiling phase, we record all kernel API invocations/returns during the execution of the test suite. Because all recorded invocations in different *process context*s are mixed, we design a technique called *Context-Sensitive Clustering* to de-interleave invocations into clusters and label each cluster with *FuncEntry* tag. After the recording and clustering of invocations, we organize the runtime information captured into the *Profiling Data* and transfer it to the next testing phase.

Tracking of Driver-Kernel Interactions: Because QEMU can translate every instruction in the guest VM, we track the execution of the driver through monitoring its program counter at the granularity of a basic block. If the current basic block is within the driver's memory region and the previous one is located outside, it means that control flow transits from the kernel into the driver. If the previous basic block is within the driver's region and the address of the current one is out of the driver's boundary, it indicates that the control flow transits from the driver into the kernel. Then all control flow transitions passing the driver boundary can be recorded. The transitions between kernel and driver are either in the form of a *call/jump* instruction or a *ret* instruction.

As mentioned earlier, we prepare a test suite for the subject device driver we want to test. When the test suite begins to execute, we issue *TESTON* to notify DRIP of the start of the test. When the test finishes successfully or terminates due to an assertion failure, it also notifies our system with the result through *TESTSUCC/TESTFAIL* respectively. We denote it as one *Testing Cycle* from the beginning of a test to the end. We record all the transitions that are issued through *call/jump* instructions from the driver to the kernel in one *Testing Cycle* and we treat them as kernel API invocations.

After recording kernel API invocations from the driver to the kernel, we need to capture the return value of each invocation because it may be used by subsequent instructions. We

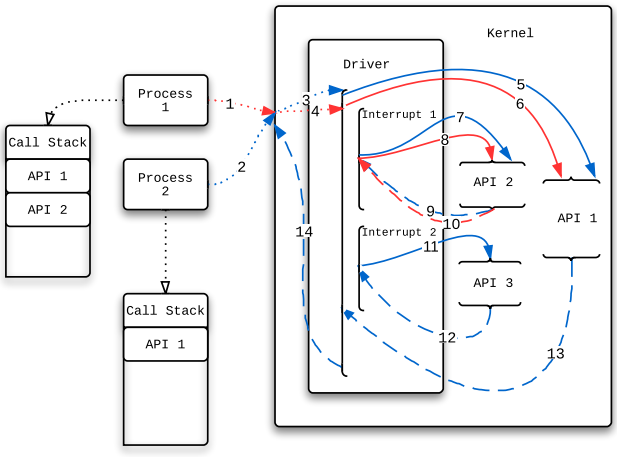

Fig. 2: Function Return Value Mapping

record the transitions that are issued through the *ret* instruction from the kernel to the driver and we treat them as kernel API returns. The return value is stored in a general register, e.g., EAX under x86. Some kernel APIs are void functions or the return values are not used any further. We check the def-use of EAX in subsequent instructions to determine whether the return value is used or not. If EAX is defined first, it indicates that the return value is not used and has no effect on later instructions. If EAX is used first, we need to record this value and map it to the function invocations recorded before.

Due to multitasking in the operating system, a kernel driver's code can be executed concurrently in different *process context*s. Most operating systems also enable the feature of kernel reentrancy and kernel preemption, which means all processes can be interrupted in kernel mode and resumed from a previous checkpoint when the interrupt is handled. These properties make it complicated to create one-to-one mapping from the kernel API return to its invocation. Fortunately, the starting address of the kernel stack for different processes/threads is different and can be used to uniquely identify the *process context*. We leverage this property to identify the current context of the driver code being executed. Processes may be interrupted to handle hardware interruptions and nested interrupts are possible. It conforms to the Last In First Out (LIFO) order in the same *process context*. We maintain a call stack for every active process to record the last function invocation and its expected return address. When a function returns, we can find the call stack according to the current *process context* and map the return value to the last function invocation stored in this call stack and pop it.

We give a simplified example in Figure 2. We assume processes 1 and 2 are running simultaneously in the system and both request the same service of the device driver (dotted red paths 1 and 4 for process 1 and dotted blue paths 2 and 3 for process 2). For the execution of driver code in Process 1's context, it invokes API 1 (solid red path 6) of the kernel and is interrupted before returning from API 1. Then it calls API 2 (solid red path 8) in the handler of interrupt 1. Before returning from API 2, the call stack of Process 1 contains both API 1 and API 2. When returning from API 2 (dashed red path 10), the current *process context* is Process 1 and it can map the return to API 2 in Process 1's call stack and pop API 2 from call stack. For the execution in Process 2's context, both API 2 and API 3 in two interrupt handlers have returned (dashed blue path 9 and 12) and popped from the call stack. There is only API 1 in the call stack. When API 1 returns from the kernel (dashed blue path 13), its current *process context* is Process 2 and then we can map the return value to API 1 and pop it from the call stack.

Context-Sensitive Clustering: After recording all kernel API invocations in one *Testing Cycle* linearly, we find that invocations in different *process context*s may interleave with each other due to multitasking and kernel preemption. In Figure 3(a), we present recorded invocations of a trojaned E1000 NIC driver. It is compromised by the module injection technique [5] and the payload is a DR rootkit. Each entry contains a symbol name (just used for clear demonstration, symbols of the driver are not needed by DRIP), *funcaddr* and *apiaddr*. *Funcaddr* is the function invocation's call site address in the driver and *apiaddr* is the API's entry address in the core kernel. Interleaved invocations make it difficult to design an efficient removal strategy in the later phase because there is no information of connections between invocations.

We design a technique called *Context-Sensitive Clustering* to de-interleave kernel API invocations recorded during the profiling phase. It is based on the observation that, for the trojaned driver, each function in the driver either belongs to the benign logic or to the malicious logic and we can group kernel API invocations issued under the same function together. Thus after clustering according to each function address in the driver, interleaved kernel API invocations belonging to benign/malicious logic are naturally separated and become easier to process in the next phase.

In Figure 3(b), we present the result after applying *Context-Sensitive Clustering* and organize the kernel API invocations in reverse chronological order. The entries in red are function invocations from the DR rootkit and those in blue are from the E1000 NIC driver. We denote the group clustered as a *Context Group* and present one specific example in the red rectangle. This *Context Group* is headed with *hook_execve* entry addr:0xf81f08b0 and it contains three function invocations, *ptregs_execve*, *strstr* and *getname*. It means during the execution of function *hook_execve* whose entry address is 0xf81f08b0 in the driver, it invokes these three kernel APIs. We combine the clustered kernel API invocations with the return values to generate the *Profiling Data* and transfer to the testing phase.

C. Testing Phase

In the testing phase, DRIP eliminates kernel API invocations that do not affect the correct execution of the test suite, which ensures the preservation of the driver's benign functionalities. We obtain the *Profiling Data* that contains clustered kernel API invocations from the preceding profiling phase and rename it as *Testing Data*. Initially, entries in the *Testing Data* are not marked with any status.

When the *Testing Cycle* begins, we load the snapshot to execute the test suite from a deterministic state. Upon receiving

(a)
Kernel API Invocations Recorded before De-Interleaving

(b)
Kernel API Invocations Clustered after De-Interleaving

Fig. 3: Context-Sensitive Clustering

TESTON, a subset of kernel API invocations that have not been marked will be selected and removed from the memory. As aforementioned, we cluster kernel API invocations in the profiling phase into different *Context Groups*. Selection of candidates for removal is based on the clustering. First we select one *Context Group* that is marked as *UNTESTED* and try to remove all function invocations in it. Then we change its status to *TESTING*. We maintain a *FuncStack* to record the current function invocation list that is being tested. Then we enter the VM to resume executing the test suite. If it runs to completion successfully, we mark the current kernel API invocations as *UNNEC*, which means that they do not violate the correct execution of the test suite and can be removed before the next *Testing Cycle*. If the removal causes failure of the test suite, we utilize a divide and conquer approach to split the *Context Group* into two equal subsets and push them into the *FuncStack*. Then we recover current invocations being tested in memory and re-launch the next *Testing Cycle*. If the current set contains only one function invocation, which cannot be divided any further, we mark this function invocation as *CRITICAL* if test fails. If all kernel API invocations in the same *Context Group* have been tested, we mark this context group as *TESTED* and continue to process the next one. We will iterate this process until all kernel API invocations in the *Testing Data* are marked. The detailed algorithm is presented in the Test-and-Reduce Algorithm 1.

Recall that we list every function invocation in the *Context Group* using reverse chronological order. The reason is that the result of earlier function invocations will probably impact the later function invocations. But removing the later invocation first will not impact the earlier ones. In Figure 4, we present the function *h4x_unlink* from KBeast, which is one of the malicious payloads in our evaluation. *h4x_unlink* is used to hijack the *sys_unlink* system call from Linux. It analyzes the

Algorithm 1 Test-and-Reduce Algorithm in Testing Phase

```
Input: ContextGroupListhead ⇐ FirstContextGroup
       FuncStack ⇐ EmptyStack
       CurrContextGroup ⇐ NULL
       CurrFuncList ⇐ NULL
       EntryFunc ⇐ Dispatch(signal)
1: procedure Dispatch(signal)                          ▷ Dispatch based on signal
2:     if signal = TESTON then
3:         PatchTestedFuncs()
4:         PatchCurrFuncList()
5:     else if signal = TESTSUCC then
6:         MarkCurrFuncList(UNNEC)
7:         LoadSnapshot()                              ▷ Load Snapshot of VM
8:     else if signal = TESTFAIL then
9:         if Size(CurrFuncList) = 1 then
10:            MarkFuncList(CRITICAL)
11:        else
12:            RecoverCurrFuncList()
13:            {FuncList_1, FuncList_2} ⇐ SplitList()
14:            PushStack(FuncStack, {FuncList_1, FuncList_2})
15:        LoadSnapshot()

16: procedure PatchTestedFuncs(void)
17:    ContextGroupIter ⇐ ContextGroupListhead
18:    while ContextGroupIter ≠ NULL do
19:        if ContextGroupIter.status = TESTED or TESTING then
20:            for all Func in ContextGroupIter.funclist do
21:                if Func.status = UNNEC then
22:                    RemoveFunc(Func)                ▷ Remove the invocation in Memory
23:        if ContextGroupIter.status = TESTING then
24:            Assert(CurrFuncList ≠ NULL)
25:            return
26:        if ContextGroupIter.status = UNTESTED then
27:            CurrContextGroup ⇐ ContextGroupIter     ▷ Init CurrContextGroup
28:            CurrFuncList ⇐ ContextGroupIter.funclist  ▷ Init CurrFuncList
29:            ContextGroupIter.status ⇐ TESTING
30:            return
31:        ContextGroupIter ⇐ ContextGroupIter.next

32: procedure PatchCurrFuncList(void)
33:    for all Func in CurrFuncList do
34:        RemoveFunc(Func)

35: procedure MarkCurrFuncList(status)                 ▷ Mark statuses in the CurrFuncList
36:    if status = UNNEC then
37:        for all Func in CurrFuncList do
38:            Func.status ⇐ UNNEC
39:    else if status = CRITICAL then
40:        Assert(Size(CurrFuncList)=1)
41:        CurrFuncList[0].status ⇐ CRITICAL
42:    RecoverCurrFuncList()
43:    if IsEmpty(FuncStack) then
44:        CurrContextGroup.status ⇐ TESTED
45:        CurrFuncList ⇐ NULL
46:    else
47:        CurrFuncList ⇐ PopStack(FuncStack)

48: procedure RecoverCurrFuncList(void)                ▷ Recover CurrFuncList
49:    for all Func in CurrFuncList do
50:        RestoreFunc(Func)                           ▷ Restore Invocation in Memory
```

pathname argument and protects its own malicious files from being deleted. We highlight 3 function invocations in blue, which are *kmalloc*, *copy_from_user* and *kfree*, in the function body. If we remove these 3 function invocations together in one *Testing Cycle*, it is safe and will not cause problem. But other kernel API invocations located between these 3 invocations may be marked as *CRITICAL*.

In this example, *o_unlink* cannot be removed because it is the function pointer to the original *sys_unlink*. Removing it can make deletion of files ineffective. This critical function invocation splits the current *Context Group* and forces removal of these 3 function invocations to occur in different *Testing Cycles*. If we do not use reverse chronological order, we will try to remove *kmalloc* first and assign *kbuf* with a fake

```
asmlinkage int h4x_unlink(const char __user *pathname) {
    . . .
    char *kbuf=(char*)kmalloc(256,GFP_KERNEL);
    copy_from_user(kbuf,pathname,255);
    . . .
    r=(*o_unlink)(pathname);
    kfree(kbuf);
    . . .
}
```

Fig. 4: Reverse Chronological Order

address. The subsequent function *copy_from_user* will write to an unsafe address and *kfree* will free a memory block that has never been allocated. This will probably crash the system. Then we will mistakenly mark *kmalloc* as *CRITICAL*, but in fact it is not. If we remove backwards in the following order: *kfree*→ *copy_from_user*→ *kmalloc*, all 3 invocations will be safe to remove. This greatly reduces the risk of mutual influences of function invocations.

In order to accelerate the handling of failing cases, we add two optimization techniques to handle different failing scenarios:

1) *Test Suite Halts in the Middle:* Removal of some function invocations will cause the test case to freeze without progress. We handle this by setting a timer and the time interval is estimated by profiling the execution time of previous successful cases. If the timer expires, we consider this *Testing Cycle* a failure and proceed to execute the next one.
2) *Test Suite Causes OS Crash and Rebooting:* Removal of a critical function invocation may cause OS crash and rebooting. In this case we do not have to wait for the timer to expire. Instead, we add rebooting detection logic by checking whether the paging bit is set in the control register. We can determine that it is a failing case if the system is rebooting after we remove certain invocations.

To eliminate kernel API invocations in the driver, we patch them in the driver's memory. The method of patching varies according to platforms and file formats. For ELF under Linux, the destination address of call instructions is unknown before loading. The module loader resolves symbols of the kernel API in the entries of the relocation section and fixes up the destination in the code section with the absolute address when loading the kernel module. For Portable Executable (PE) under Windows, it utilizes the import address table (IAT) to store the absolute virtual addresses of kernel APIs. The contents are populated when that driver is loaded into the system. The kernel API invocations in PE drivers use two calling styles. The first one uses the indirect call generated by the compiler and retrieves its destination address from IAT. The second one makes a direct call to an indirect jump and the jump destination is stored in the IAT. If the return value is not used by the subsequent instructions or it is a void function, we can simply replace the *call* or *jmp* instruction with a series of nops in memory. If the return value is used later (e.g., as a predicate condition) and can determine the control flow, replacing the instruction with nops will lead to an undefined situation.

As we have already recorded the return value of every kernel API invocation in the profiling phase, we can replace the calling instruction with a *mov* instruction that fills the return value in the EAX register. This kind of replacement can be applied to the ELF driver and the first calling style of PE drivers. For the second calling style of PE drivers, we replace the indirect jump with *ret* to return to the original direct call to eliminate this invocation.

The other issue we need to consider during memory patching is the calling convention of the kernel API. If the caller is responsible for cleaning up the stack, no additional effort is needed because *push* and *pop* operations are performed in the same function. If the callee is responsible for cleaning up the stack, the situation becomes more complicated. In this scenario, arguments are pushed into stack by the caller and the callee unwinds the stack before returning. We choose to remove the *push* operations before the function invocation in the caller to solve this problem. We can record the number of stack bytes that need to be unwound. This is determined by the 16-bit parameter of the last *ret* instruction in the kernel function. We then trace back from the kernel API invocation instruction to search for *push* instructions and replace these instructions with nops.

If the patching operation is successful for the current *Testing Cycle*, which means the function invocation is tested to be *UNNEC*, we record all the modified content and the address of this function invocation for the rewriting phase. After writing new content into the memory address of a kernel API invocation, we mark this specific basic block as a candidate for memory invalidation. When the snapshot is reloaded in the next *Testing Cycle* and the emulator tries to execute this basic block, we invalidate the cache of this basic block and force the emulator to perform binary translation on it because the instructions inside it have been modified and it should execute the newly translated code.

D. Rewriting Phase

The last phase of DRIP's driver purification procedure is to remove kernel API invocations marked as *UNNEC* in the binary file. We have already tested and retrieved the list of unnecessary API invocations and their addresses in the memory from previous phases. The procedure of patching the binary is similar to patching memory in the testing phase. We need to map the loading addresses of API invocations to their relative addresses inside the binary and apply the changes recorded in the testing phase to code sections.

Finishing these steps is not enough for the purified driver to work correctly. Every relocatable driver has its own relocation table consisting of a list of pointers. These pointers point to addresses in the binary that need to be fixed up after the driver is loaded into the system. If we remove the function invocations whose addresses are included in the relocation table, we also need to remove these relocation entries in the relocation table. Otherwise the loader of the OS will still fix up the function address and cause memory corruption. Because holes are not permitted in the relocation table for both ELF and PE, we swap the value of each removed entry with the value of last entry in the relocation table to fill the hole and adjust the table size in the header accordingly. For PE files, we also need to calculate the new checksum value and write it into its PE header, otherwise Windows will refuse to load the driver with the wrong checksum.

After finishing all these steps, we generate a new relocatable binary as a purified driver and it can be loaded into the system for execution.

E. DRIP Prototype

We have implemented a proof-of-concept prototype of DRIP. The prototype is built as a component of QEMU. As a full system emulator, QEMU dynamically translates the guest VM's code at the granularity of basic blocks and executes them on the emulated CPU. Such a platform enables us to perform binary analysis on the code region of drivers, intercept dynamic control flow, and patch the memory at runtime to test effects of our kernel API invocation removal operations. In addition to processor emulation, QEMU also provides a set of emulated devices, which provides an alternative to verify the correctness of test cases through mapping the high-level program to low-level hardware events. For example, we can simulate keystrokes in emulated hardware and capture the keys in the test suite to test the keyboard driver.

To prove the generality of DRIP, We have tested the prototype on two guest operating systems, Ubuntu 10.04 and Windows XP SP2[2]. We believe that it is easy to extend our current system to support more operating systems of different versions because DRIP does not rely on the semantics of a guest VM. We support relocatable file formats for both PE and ELF, which are standard formats for Windows and Linux drivers.

IV. Evaluation

In this section, we present the evaluation results for the DRIP prototype in two aspects, effectiveness and performance. The hardware configuration of our testing platform is a Dell OptiPlex 780 with Intel® Core™ 2 Duo CPU E8400 3.00GHz CPU and 4GB memory. We develop and run the DRIP system on Ubuntu 11.10 (Linux kernel version 3.0.0) to generate the purified driver. To prove that changes in the underlying infrastructure do not affect the functionality of purified drivers, we use VMware Workstation 8.0 as the hypervisor and Windows 7 as the host operating system to perform evaluation on purified drivers. We allocate 1GB memory for each guest VM. The guest OSes are Ubuntu 10.04 (Linux kernel version 2.6.32) and Windows XP SP2.

A. Evaluation of Effectiveness

In the effectiveness evaluation, we use trojaned drivers infected by binary driver rewriting tools as input to DRIP and generate the corresponding purified drivers. Then we scrutinize the behavior of the generated driver manually to validate that the malicious behavior has been eliminated and the functionality of the benign parts of the driver and the kernel are not impaired. We present five representative case studies on drivers in different categories in detail and present other results briefly in Table I.

[2]We use Ubuntu 10.04 and Windows XP SP2 because the trojaned driver samples we perform evaluation on do not support newer versions yet.

Name	Infection Type	Platform	Purified	Note
E1000+KBeast	Module injection	Linux	✓	E1000 NIC driver infected with KBeast as payload
E1000+DR	Module injection	Linux	✓	Case Study I
E1000+Adore-ng	Module injection	Linux	✓	E1000 NIC driver infected with Adore-ng 0.56 as payload
E1000+Sebek	Module injection	Linux	✓	E1000 NIC driver infected with Sebek-lin26-3.2.0b as payload
E1000+Redir	ERESI	Linux	✓	Cast Study II
Kbdevents	Embedded	Linux	✓	Case Study III
Null+SSDT	DaMouse	Windows	✓	Null.sys infected by DaMouse
Kbdclass+SSDT	DaMouse	Windows	✓	Case Study IV
E1000325+SSDT	DaMouse	Windows	✓	E1000325.sys infected by DaMouse
Beep+Klog	Binary Transformaion	Windows	✓	Case Study V
E1000325+Klog	Binary Transformation	Windows	✓	E1000325.sys infected with Klog as payload

TABLE I: Results of effectiveness evaluation against a spectrum of trojaned drivers

Case Study I: E1000 NIC driver with DR rootkit implanted under Linux: In phrack issue 61 [3], truff described a driver infection technique to hide the rootkit and ensure that it will be reloaded after rebooting. The basic idea is to rename the malicious function *evil* with *init* in the section *.strtab* to trick the system to load it. It only applies to Linux kernel 2.4.x, so it is no longer valid for the latest Linux kernel because the module loading procedure has been changed in new kernel version. From Linux Forum [5], coolq extended this approach to Linux Kernel 2.6.x by modifying the module *init* function entry in the relocation section *.rel.gnu.linkonce.this_module* to guide the system to load the initialization function in the malicious module. In the latest issue 68 of phrack [4], styxˆ proposes a similar approach to infecting modules in kernel versions 2.6.x and 3.0.x. It redirects *init_module* to load function *evil* instead of original *init* function. In order to enable malicious modules to invoke the original init function, it also updates the symbol binding of *init* from local to global. The effects of these two approaches are equivalent and we choose to use the former method to inject DR rootkit into an E1000 NIC driver as our target.

The DR rootkit leverages a debug register-based hooking engine, which does not require modification to the system call table, to perform traditional rootkit behavior, like hiding processes, sockets, and files. To be more specific, it determines the name of a file it wants to hide. (In the version we obtain the name is *AAA*.) Then it hides the presence of this file in the file system by modifying the file listing result in the directory. When executing this file, the rootkit escalates *AAA*'s privilege to root, hides all the sockets created, hides all the child processes forked, and prevents other processes from opening files owned by *AAA*.

The trojaned driver contains both the functionality of a benign E1000 NIC driver and a malicious kernel rootkit. We pass it to DRIP to cripple its malicious behavior and retain the benign NIC driver behavior. We select and synthesize test cases from LTP (Linux Test Project) [15], Linux utility programs, and Iperf to cover the benign functionalities of E1000 NIC driver and the reliability of the overall system. We have validated that the purified driver behaves the same as a benign E1000 NIC driver with the malicious operations from the DR rootkit eliminated.

Case Study II: E1000 NIC driver with kernel function redirection under Linux: From case study I, we learn that we can implant malicious code inside the initialization function to install system call hooks. In fact, when the driver code invokes the kernel function, we can intercept and redirect any function invocation to a malicious function first. The malicious function can act as a proxy to invoke the original function and return the result to the original invocation. This kernel function redirection technique is proposed in the libkernsh of ERESI [8].

We prepare an interposition kernel module, which contains malicious functions from the KBeast rootkit and link it with the E1000 NIC driver to generate a trojaned driver. The relocation table of this new driver contains all the addresses of code/data that need to be fixed up during loading. We scan this table to find the function invocation we want to hijack and modify it to detour to the malicious function in the interposition module. The payload, KBeast, is a new kernel rootkit based on other well-known rootkits and supports the latest Linux kernel versions. It contains traditional rootkit functionalities, e.g., process hiding, files hiding, keystroke logging, local root escalation, etc. Its basic idea is to patch the system call table of Linux and detour system calls to its fake functions that are crafted by the attacker. Because system calls are hijacked, KBeast can easily manipulate the intermediate results and return fake results to the user. We select similar test cases as in Case Study I to build our test suite to ensure the reliability of the system and core benign functionalities of the E1000 NIC driver. After purification, we validate that KBeast's cloaking effects on the system have been eliminated and we still preserve the E1000 driver's original functionalities.

Case Study III: Kbdevents under Linux: Kprobes [23] is a lightweight debugging mechanism in the Linux kernel that allows developers to intercept kernel routines at runtime to collect debugging information. Kbdevents [24] is a Linux kernel module based on Kprobes to intercept keyboard events. It can be used as a debugging tool to verify the correctness of the keyboard driver. On every key pressed, Kbdevents has additional functionality to launch user scripts from kernel space, e.g., *keylogger* to dump keystrokes into a file, *printscr* to take screenshots and *typewriter* to imitate typewriter sounds. These supplementary capabilities are not necessary for debugging purposes. So we can perform purification on Kbdevents to minimize it to contain only the debugging functionality. We build a special test suite to simulate keystrokes out of VM, i.e., generate keyboard interrupt from QEMU, and capture them in the guest VM to verify the correctness of Kbdevents' debugging functionality. After purification, we find all

kernel API invocations related to launching user scripts from the kernel (e.g., *call_usermodehelper_{setup,exec}*) have been removed from the driver. The purified driver can still intercept keystrokes to debug the Linux keyboard driver.

Case Study IV: Infected Kbdclass driver by DaMouse under Windows: DaMouse [6] is a PE driver infection technique under Windows. It implants existing malicious code into a windows device driver in the system. It utilizes a virus coding technique called Entry-Point Obscuring (EPO) to patch API invocation inside the device driver. When this patched API is invoked, it installs a permanent System Service Dispatch Table (SSDT) hook to redirect the system call to the hook function inside the driver. The hook function contains malicious code to filter the results and can eventually complete the procedure by invoking the original system call.

In this case study, we use DaMouse to infect *kbdclass.sys*, the keyboard class driver in Windows. DaMouse patches the Kbdclass driver and install the SSDT hook at *NtOpenProcess*. Then system calls to *NtOpenProcess* are redirected to the hook function called *NewNtOpenProcess*. The filter code in the hook function determines whether the target process is *iexplorer.exe*, which belongs to the Internet Explorer. If so, the *NtOpenProcess* request will be denied. The symptom noticeable to the user is that he/she cannot open a new web page in the Internet Explorer. For other processes, the malicious code extracts the *NtOpenProcess*' arguments, e.g., pid and name, of calling process and dumps the result through *DbgPrint*. We build the test suite for Kbdclass through sending keystrokes from QEMU into VM, which is similar to Case Study III, and verify them in the test program within the VM. After purification, we can keep the keyboard driver's functionality, Internet Explorer can open new tabs successfully and there is no process information leakage any more.

Case Study V: Beep driver infected with klog as payload under Windows: In previous case studies, we have applied DRIP to purify drivers infected by existing binary infection tools. In this case study, we try to prove the generality of DRIP by purifying trojaned drivers generated by a binary transformation tool newly developed in our own research efforts. This technique enables transplanting binary functional module extracted from one binary into another binary. We extract the malicious functions, i.e., keyboard attaching and keystrokes dumping, from klog, which is a well-known Windows keyboard sniffer. Then we utilize our own binary transformation technique to implant the extracted functions into the beep driver of Windows. In order to check if the beep driver works properly, we add some functionality-checking logic in the emulated pc speaker in QEMU to verify the beep events. After purification, we load the purified beep driver into the production environment and it works as expected and keyboard can no longer dump sniffed keystrokes to a file any more.

B. Performance Evaluation

The time it takes for DRIP to purify a specific driver is highly dependent on the driver's code complexity, coverage of test suite, and hardware configuration. We present the complete performance statistics of purification process for each trojaned driver in Table II. It shows the ratio of "Removed Function Invocations" to "Recorded Function Invocations", the purification time, and the number of testing cycles. Our results indicate that DRIP is suitable for offline driver purification.

Name	Ratio[1]	Time	NTC[2]
E1000+KBeast	57/69	42min 13s	37
E1000+DR	13/25	21min 23s	40
E1000+Adore-ng	7/23	20min 46s	39
E1000+Sebek	13/34	19min 19s	35
E1000+Redir	37/53	35min 38s	34
Kbdevents	8/12	8min 25s	13
Null+SSDT	5/7	4min 4s	12
Kbdclass+SSDT	13/21	15min 31s	32
E1000325+SSDT	20/24	22min 15s	19
E1000325+Klog	22/28	24min 35s	19
Beep+Klog	24/35	31min 1s	44

TABLE II: Performance evaluation results with a spectrum of trojaned drivers

[1] Ratio here represents the ratio of "Removed Function Invocations" to "Recorded Function Invocations".
[2] NTC stands for "Number of Testing Cycles"

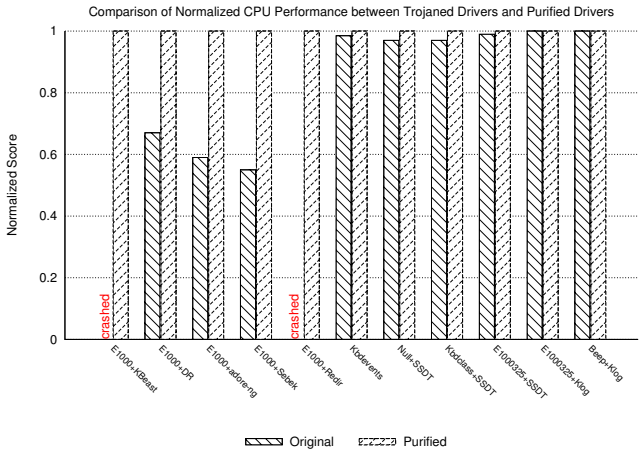

Fig. 5: Comparison of Normalized CPU Performance between Trojaned Drivers and Purified Drivers

We next measure the system performance overhead with the purified driver and compare it with system performance with the trojaned driver. We use SPECINT 2000 under Windows and UnixBench under Linux to measure the CPU performance. We normalize the performance results and present them in Figure 5. The left bars indicate the normalized performance scores (the higher the better) after loading the original trojaned driver. The right bars are normalized performance scores after loading the purified driver. In the experiments with trojaned E1000+KBeast/E1000+Redir, the system crashed when executing the test case *file copy* in UnixBench. The reason is that KBeast rootkit cannot survive the workload of test case *file copy* in the UnixBench and both trojaned drivers contain the KBeast's code. After purification, both drivers support the benchmark successfully because the KBeast functionality has been eliminated. For the other experiments, the purified drivers improve benchmark performance by 1% to 45% compared

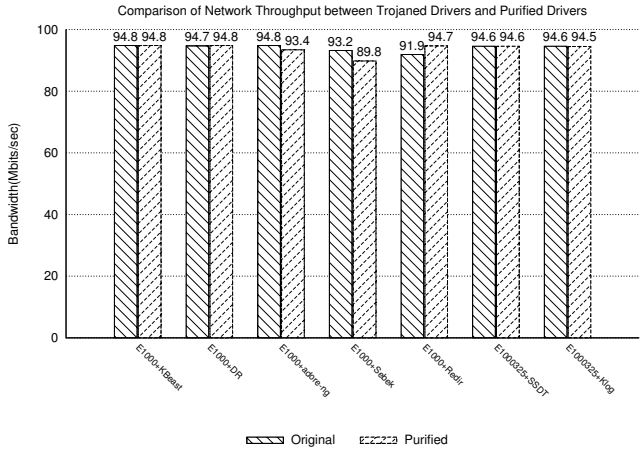

Fig. 6: Comparison of Network Throughput between Trojaned Drivers and Purified Drivers

with that under trojaned drivers. This is intuitive to understand because the purified drivers are without unnecessary kernel API invocations and thus execute less code than the trojaned drivers.

Besides testing CPU performance, we also utilize Iperf to measure the network throughput for all cases involving the NIC driver. We compare the TCP throughput of the trojaned driver with the purified driver and present the result in Figure 6. The left bars are bandwidths for trojaned drivers and the right bars are for purified drivers. From the results, we observe that 4 out of 7 purified drivers maintain the same or slightly better throughput compared with the trojaned drivers. The worst-case overhead observed is only 4% for the purified E1000+Sebek driver.

Our performance evaluation results demonstrate that purified drivers generated by DRIP can maintain (almost) the same network performance as under their trojaned versions. Moreover, the purified drivers lead to better CPU performance with the removal of embedded malicious operations.

V. Discussion

In this section, we discuss the limitations of DRIP and propose some possible solutions.

Coverage of Test Suite: A test suite can only ensure the correctness of the tested behaviors within a specific application and the environment in which it executes. Correspondingly, by using a test suite, we only guarantee to preserve the driver functionalities that those tests exercise. This may not cover all benign functionality within a driver, or it may require new tests in order to preserve behaviors not originally covered by a test suite. For practical deployment, we recommend adjusting test suite and generating new purified drivers based on different deployments of applications. This can also be treated as specialization of the driver. If we do not need some of the redundant features, we can use DRIP to minimize the functionalities of the driver.

False Positives of Removed Kernel API Invocations: We remove function invocations that are not necessary to our test suite, but it does not mean all these removed invocations are useless. For example, we have found some memory deallocation invocations have been removed. This may not impact the execution of the test suite over a short period, but it will cause memory leaks and affect the performance of the system in the long term. We can add some test cases to prevent these invocations from being removed. For example, we can measure the memory usage of the driver and report failure if it exceeds a threshold. Another simple solution is to add these well-known functions with specific functionalities into a white list of functions to keep and skip them when profiling the driver.

Self-contained Malicious Code: Some malicious code can jeopardize the kernel without invoking any kernel APIs. For example, some kernel malware can directly modify the kernel memory to achieve their malicious effects. They can evade DRIP's purification as we monitor at the granularity of API invocations. But the functionalities of such self-contained malicious code are limited and it is hard for them to adapt to new kernel versions. We will improve DRIP to monitor at the level of memory operations during the profiling phase to address this problem in the future.

Limitation of the Testing Environment: Our *Testing Environment* is based on QEMU to test the drivers. We can support kernel drivers that extend the core kernel functionalities. For the device drivers that control real hardware devices, we can only support those whose devices are emulated by QEMU. In the DDT research effort [14], the authors propose the symbolic device, which presents itself as a virtual device to facilitate symbolic execution of driver code. This technique can complement DRIP to address the problem that some devices are not emulated by QEMU. We consider integration of symbolic device our future work.

VI. Related Work

Online Device Driver Isolation: Nooks [9] involves a shadow driver mechanism to conceal driver failures from applications by monitoring the state of real drivers during normal operation. It inserts itself when failure occurs, thus improving the reliability of the overall system. SafeDrive [25] improves kernel extension reliability by adding type-based checking to driver code and enforcing runtime memory safety. In order to leverage user level programming tools and reduce kernel level faults introduced by drivers, Microdriver [26] partitions an existing driver into a kernel level driver handling performance critical tasks and a user level driver processing low-performance issues. The Nexus [27] operating system moves the device driver to user space and it leverages device-specific reference monitors to validate that all the interactions between drivers and devices conform to safety specifications. To protect untrusted device drivers from compromising a system, SUD [10] leverages recent hardware support to confine operations of devices and allows unmodified Linux device drivers to run in user processes by emulating a Linux kernel environment in user space. These research efforts are designed to isolate buggy drivers at runtime. Compared with DRIP, they incur additional performance overhead and cannot target intentionally malicious drivers.

HUKO [12] provides a hypervisor-based approach to enforce mandatory access control policies on the untrusted

extension. It limits the attacker's ability to jeopardize the integrity of the kernel. Gateway [11] is also a hypervisor-based approach to trace kernel malware behavior. It monitors kernel APIs invoked by untrusted kernel extensions and isolates the driver at an address space separate from the kernel. These two approaches both require the underlying hypervisor to support online monitoring and they do not aim at purifying trojaned malicious drivers.

Offline Device Driver Testing: SDV [13] statically checks source code paths of device drivers to make sure they use the Windows API correctly. DDT [14] exercises the closed source device drivers to find bugs by using symbolic execution. These two offline approaches are designed to test buggy drivers thoroughly but not for removing malicious behaviors from the driver. On the other hand, both of them can complement DRIP for improving the coverage of test suites.

Sandboxing: SFI [28] proposed the concept of sandboxing to prevent unsafe instructions in untrusted modules from writing or jumping to an address outside of their domain. Based on SFI, XFI [29] leverages control flow integrity to prevent circumvention and support fine-grained memory access control. Vx32 [30] and NaCl [31] isolate the execution of an application to a sandbox with restricted memory access and system interface to prevent it from jeopardizing the system. BGI [32] is a compiler-based software fault isolation approach to provide byte-granularity memory protection. In order to enforce API integrity, LXFI [33] utilizes a compiler plug-in to generate instrumented driver code with security policies specified by programmers. We can enhance DRIP to perform finer-grained purification at the level of memory operation by leveraging their ideas in the future.

Emulation-Based Analysis: Emulation-based techniques have been widely used in malware profiling and analysis. Panorama [34] captures the privacy-breaching behavior of malware by leveraging whole-system taint tracking. K-Tracer [35] dynamically analyzes a rootkit's malicious behavior by using backward slicing and chopping techniques. HookFinder [36] and HookMap [37] perform dynamic analysis to identify kernel hooks implanted by rootkits. PoKeR [38] profiles a kernel rootkit's behavior by traversing from static objects to locate dynamic objects and performing address-object mapping. Rather than detect malware, DRIP extends the emulation platform to perform trojaned malicious behavior elimination from a driver. Virtuoso [39] involves a technique to create introspection-based security tools automatically out of a VM by tracing and combining the execution traces of In-VM programs. RevNIC [40] is a technique that helps automatically reverse engineer the logic of a network device driver and synthesize a new driver with the same functionality for a different platform. Rather than combining traces to re-create a new binary, the goal of DRIP is to identify malicious logic in existing drivers and perform binary rewriting to eliminate their malicious effects.

VII. Conclusion

We develop and evaluate DRIP, a framework to eliminate malicious/unnecessary behaviors of a trojaned kernel driver and preserve its benign functionalities for a target application. Through our evaluation, we demonstrate the effectiveness of DRIP to achieve this goal. After loading a purified driver, we can maintain or even improve the system's performance compared with running the same workload under the trojaned driver.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research has been supported by DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA.

References

[1] "Sony, Rootkits and Digital Rights Management Gone Too Far," http://blogs.technet.com/b/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx.

[2] "Legit bootkits," http://www.securelist.com/en/analysis/204792203/Legit_bootkits.

[3] "Infecting loadable kernel modules," http://www.phrack.org/issues.html?issue=61&id=10#article.

[4] "Infecting loadable kernel modules kernel versions 2.6.x/3.0.x," http://www.phrack.org/issues.html?issue=68&id=11#article.

[5] "Module injection in 2.6 kernel," http://www.linuxforum.net/forum/showflat.php?Cat=&Board=security&Number=536404&page=0&view=collapsed&sb=5&o=31&fpart.

[6] "Driverless Kernel Mode Rootkit," http://www.rohitab.com/discuss/topic/28440-driverless-kernel-mode-rootkit.

[7] "Kernel Driver Backdooring," http://securitylabs.websense.com/content/Blogs/2730.aspx.

[8] "The ERESI Reverse Engineering Software Interface," http://www.eresi-project.org.

[9] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. ACM, 2003, pp. 207–222.

[10] S. Boyd-Wickizer and N. Zeldovich, "Tolerating Malicious Device Drivers in Linux," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIX Association, 2010, pp. 9–9.

[11] A. Srivastava and J. Giffin, "Efficient Monitoring of Untrusted Kernel-Mode Execution," in *Proceedings of the 18th Annual Network and Distributed Systems Security Symposium*, 2011.

[12] X. Xiong, D. Tian, and P. Liu, "Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[13] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. ACM, 2006, pp. 73–85.

[14] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing Closed-Source Binary Device Drivers with DDT," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIX Association, 2010, pp. 12–12.

[15] "Linux Test Project," http://ltp.sourceforge.net.

[16] "skipfish web application security scanner," https://code.google.com/p/skipfish.

[17] "curl-loader," http://curl-loader.sourceforge.net.

[18] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 209–224.

[19] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing Drivers without Devices," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 279–292.

[20] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," pp. 265–278, 2011.

[21] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. ACM, 2005, pp. 263–272.

[22] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator." USENIX, 2005.

[23] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the Guts of Kprobes," in *Proceedings of the 2006 Linux Symposium*, vol. 2, 2006, pp. 109–124.

[24] "Linux kernel module that allows you to set events on pressed keys," http://en.chuso.net/linux-kernel-module-that-allows-you-to-set-events-on-pressed-keys.html.

[25] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 45–60.

[26] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The Design and Implementation of Microdrivers," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. ACM, 2008, pp. 168–178.

[27] D. Williams, P. Reynolds, K. Walsh, E. Sirer, and F. Schneider, "Device Driver Safety Through a Reference Validation Mechanism," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 241–254.

[28] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[29] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 75–88.

[30] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, 2008, pp. 293–306.

[31] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[32] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast Byte-Granularity Software Fault Isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 45–58.

[33] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek, "Software fault isolation with API integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 115–128.

[34] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[35] A. Lanzi, M. Sharif, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[36] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and Understanding Malware Hooking Behaviors," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[37] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering Persistent Kernel Rootkits Through Systematic Hook Discovery," in *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2008, pp. 21–38.

[38] R. Riley, X. Jiang, and D. Xu, "Multi-Aspect Profiling of Kernel Rootkit Behavior," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 47–60.

[39] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[40] V. Chipounov and G. Candea, "Reverse Engineering of Binary Device Drivers with RevNIC," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 167–180.