Reverse Engineering of Binary Device Drivers with RevNIC

Vitaly Chipounov and George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

This paper presents a technique that helps automate the reverse engineering of device drivers. It takes a closed-source binary driver, automatically reverse engineers the driver's logic, and synthesizes new device driver code that implements the exact same hardware protocol as the original driver. This code can be targeted at the same or a different OS. No vendor documentation or source code is required.

Drivers are often proprietary and available for only one or two operating systems, thus restricting the range of device support on all other OSes. Restricted device support leads to low market viability of new OSes and hampers OS researchers in their efforts to make their ideas available to the "real world." Reverse engineering can help automate the porting of drivers, as well as produce replacement drivers with fewer bugs and fewer security vulnerabilities.

Our technique is embodied in RevNIC, a tool for reverse engineering network drivers. We use RevNIC to reverse engineer four proprietary Windows drivers and port them to four different OSes, both for PCs and embedded systems. The synthesized network drivers deliver performance nearly identical to that of the original drivers.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation

General Terms Performance, Design, Languages

Keywords Device drivers, Reverse engineering, Binary, Closed-source, Proprietary software

1. Introduction

The ability to use a hardware device with an operating system requires that a corresponding device driver be available, i.e., a program that knows how to mediate the communication between the OS kernel and that specific device. The driver may be supplied by either the hardware vendor or the developer of the operating system.

Hardware vendors typically provide drivers for the one or two most popular OSes. It appears that supporting many other platforms is not profitable, because the high cost of development and technical support can be amortized only over comparatively fewer customers. As a result, drivers are rarely available for every OS/device combination. This issue is common to various device classes, including network drivers. Alas, for an operating system to be viable and widely adopted, it must support a wide range of hardware.

Even when drivers are available, they are often closed-source and proprietary. Despite this making them less trustworthy, proprietary drivers are still permitted to run at the highest level of privilege in an operating system. Not surprisingly, buggy drivers are a leading cause of crashes [41]. They can also be a security threat, as was the case of a driver shipped with all versions of Windows XP that was found to contain a zero-day privilege escalation vulnerability [29].

Writing a driver for an OS that is not supported by the device vendor is challenging, because the device specification is often not available. While the interface exposed by an OS to the driver is well known, the specification of the interface between the hardware and the driver is often not public. The classic approach is to manually reverse engineer the original driver, but that involves a lot of work. When the device is too complex to be reverse engineered, developers resort to emulating the source OS using wrappers (e.g., NDISwrapper [32] allows running Windows NIC drivers on Linux). However, this adds performance overhead, can only use drivers from one source OS, and requires changing the wrapper each time the source OS driver interface changes.

Even when the hardware specification is available, writing the driver still requires substantial effort. That is why, for example, it took many years for Linux to support widely used wireless and wired NIC devices. Vendor-provided specifications can miss hardware quirks, rendering drivers based on these specifications incomplete (e.g., the RTL8139 NIC driver on Linux is replete with workarounds for such quirks).

Our proposed approach overcomes unavailability of specifications and costly development with a combination of automated reverse engineering and driver code generation. We

EuroSys'10, April 13–16, 2010, Paris, France.
Copyright © 2010 ACM 978-1-60558-577-2/10/04. . . $10.00

observe that a device specification is not truly necessary as long as there exists one driver for one platform: that one driver is a (somewhat obscure) encoding of the corresponding device protocol. Even if they are not a perfect representation of the protocol, proprietary drivers incorporate handling of hardware quirks that may not be documented in official specification sheets. We also observe that writing drivers involves a large amount of boilerplate code that can be easily provided by the OS developer. In fact, an entire category of drivers can use the same boilerplate; driver writers plug into this boilerplate the code specific to their hardware device.

We implemented our approach in RevNIC, a tool for automating the reverse engineering of network drivers. This tool can be used by hardware vendors to cheaply support their devices on multiple platforms, by OS developers to offer broader device support in their OS, or by users who are skeptical of the quality and security of vendor-provided closed-source proprietary drivers.

Our paper makes three main contributions. First, we introduce a technique for tracing the driver/hardware interaction and turning it into a driver state machine. Second, we demonstrate the use of binary symbolic execution to achieve high-coverage reverse engineering of drivers. Third, we show how symbolic hardware can be used to reverse engineer drivers without access to the actual physical device.

After providing an overview of RevNIC (§2), we describe how RevNIC "wiretaps" drivers (§3) and synthesizes new driver code (§4), we evaluate RevNIC (§5), discuss limitations (§6), survey related work (§7), and conclude (§8).

2. System Overview

To reverse engineer a driver, RevNIC observes the driver-hardware interaction, i.e., the manifestation of the device-specific protocol, encoded in the driver's binary. RevNIC synthesizes an executable representation of this protocol, which the developer can then use to produce a driver for the same or a different OS.

RevNIC employs a mix of concrete and symbolic execution to exercise the driver and to wiretap hardware I/O operations, executed instructions, and memory accesses. For this, RevNIC uses a modified virtual machine (Figure 1). The output of the wiretap is fed into a code synthesizer, which analyzes the trace information and generates snippets of C code that, taken together, implement the functionality of the device driver. The developer then pastes the code snippets into a driver template to assemble a new driver that behaves like the original one in terms of hardware I/O.

Exercising the driver. It is difficult to exercise every relevant code path in the driver using just regular workloads. Many paths correspond to boundary conditions and error states that are hard to induce. E.g., a NIC driver could take different paths depending on the packet type transmitted by the network stack (ARP, IP, VLAN, etc.) or depending on how the hardware responds.

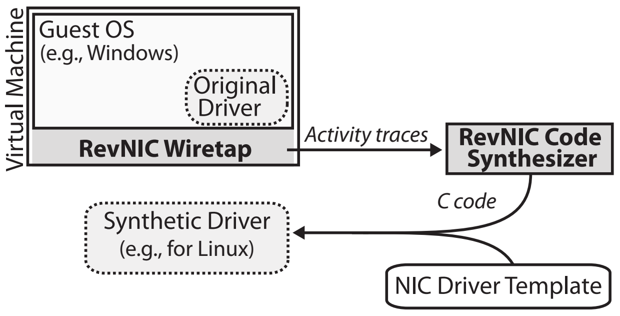

Figure 1: High-level architecture of RevNIC.

In order to induce the device driver to perform its operations, RevNIC guides the execution with a mix of concrete and symbolic workload. The concrete workload initiates driver execution by triggering the invocation of the driver's entry points. RevNIC selectively converts the parameters of kernel-to-driver calls into symbolic values (i.e., values that are not constrained yet to be any specific, concrete value) and also treats the responses from the hardware side as symbolic. This drives the execution down many feasible paths through the driver, as well as exercises all code that depends on hardware input/returns.

By using symbolic values, RevNIC is completely independent of the physical hardware device. Unconstrained symbolic values provide the driver a representation of all the responses from the hardware that the driver thinks could ever be received.

Recording driver activity. The RevNIC wiretap records the driver's hardware I/O, along with the driver's memory accesses and an intermediate representation of the instructions it executes (§3). Such detailed tracing imposes performance overheads, but the overheads are irrelevant to reverse engineering.

Synthesizing driver code. RevNIC infers from the collected activity traces the state machine of the binary driver and produces C code that implements this state machine. RevNIC automatically merges multiple traces to reconstruct the control flow graph (CFG) of the original driver. Since the generated CFG is equivalent to that of the original driver, covering most of the basic blocks is sufficient to reverse engineer the driver—complete path coverage is not necessary.

RevNIC uses virtualization and symbolic execution instead of mere decompilation for four main reasons. First, static decompilers face undecidable problems (e.g., disambiguating code from data) and can produce inaccurate results [38]. Second, while some decompilers can record dynamic execution traces [20] to improve accuracy, RevNIC explores multiple paths in parallel and covers unexplored code faster using symbolic execution. Third, since the VM catches all hardware accesses, RevNIC can distinguish accesses to a memory-mapped device from regular memory accesses, which is notoriously difficult to do statically on architectures like x86. Identifying such instructions is crucial to preserving memory access ordering in the generated code (e.g., write barriers). Finally, RevNIC must recognize DMA-allocated regions assigned by the OS to the driver (by recording the address values returned by the OS API); doing this in a decompiler requires complex inter-procedural data flow analysis.

Writing a driver template. The template contains all OS-specific boilerplate for interfacing with the kernel (e.g., NDIS API on Windows, network API on Linux). Templates can be arranged in a class hierarchy with an abstract template implementing the basic boilerplate, and derived templates implementing additional functionalities. For example, a base template may target a generic PCI-based, wired NIC, while a derived template further adds DMA capabilities. This modularity allows accommodating all common types of network devices. Depending on the OS, templates can be derived from examples in driver development kits (e.g., the Windows Driver Kit [30]) or automatically generated, with tools like WinDriver [22].

Besides mandatory boilerplate, a template also contains placeholders for the actual hardware interaction. Since devices in a given class (e.g., NICs) tend to operate in the same manner (e.g., initializing, sending, receiving, computing checksums using well-defined standards), the functional differences between drivers are at the level of code implementing this hardware I/O.

Producing the synthetic driver. The developer pastes synthesized code snippets into the template's placeholders in order to specialize the template for the device of interest. The result is then compiled into a driver. In order to paste code snippets correctly, the developer has to know how to write drivers both for the source and for the target OS.

In this paper, we explore the porting to various OSes: Linux, Windows, the µC/OS-II real-time embedded OS for FPGA systems, and our own experimental KitOS. The following sections detail how RevNIC traces driver activity and processes activity traces to produce the synthetic driver.

3. Exercising and Tracing Driver Activity

We now describe how RevNIC uses symbolic execution to exercise drivers, how it copes with the absence of actual hardware, and what information it records to generate a working driver for the target OS. Throughout the paper, we illustrate reverse engineering of Windows NIC drivers, but the techniques can be adapted to other operating systems.

3.1 Achieving High Coverage

To exercise a driver's execution paths automatically, RevNIC uses symbolic execution [24]. The driver is provided with *symbolic* values for parameters coming from the OS (λ, β, etc.), instead of the *concrete* values of those parameters (0xE9, "foo", etc.). Every assignment in the program along a given execution path updates the driver variables with a symbolic expression (e.g., `x` $= \lambda - 2$), instead of a concretely computed value (`x` $= 9 - 2 = 7$). A conditional statement (e.g., "`if x>0 then... else...`") splits (or forks) the execution into two new paths—one for the then-branch and one for the else-branch—with a common prefix. Along the then-path, values are constrained by the `if` condition ($\lambda - 2 > 0$) and along the else-path by the `else` condition ($\lambda - 2 \leq 0$). An execution path can thus be thought of as a path from the root of the program's "execution tree" (initial state) to the leaves (final states). The symbolic execution engine invokes a constraint solver each time it encounters a conditional statement that depends on a symbolic input—by solving the conjunction of the constraints collected along the current path, it decides which branches are feasible to execute.

Running a driver symbolically poses several challenges, such as interacting with the OS and coping with the exponential number of paths. To address this, we developed a technique called *selective symbolic execution* [8]. Compared to other approaches [6, 7, 31, 34], it allows running the driver inside its native environment, without requiring any modification of the driver or the OS. First, selective symbolic execution allows specifying what piece of code should be executed symbolically (e.g., the driver), and what should run natively (e.g., the OS). Second, it allows mixing symbolic and concrete driver inputs. These two types of selections reduce the number of symbolic values in the system, and prune unnecessary paths by reducing the number of forks on branches, thus mitigating the state explosion problem [4].

RevNIC employs *symbolic hardware*, which always returns symbolic values. All reads from hardware done by the driver are intercepted by RevNIC and replaced with symbolic values. This exercises many more code paths than real hardware could. Consider, for example, the NIC interrupt handler: since a read of the status register returns a symbolic value, all conditional branches that depend on the returned value are automatically explored, without requiring a cleverly crafted workload that would induce a real NIC into producing all possible return values (which may even be impossible to do solely using a workload).

As I/O to and from the hardware is symbolic, the actual device is never needed. This allows developers using RevNIC to reverse engineer drivers for rare or expensive devices they do not (yet) have access to. §3.4 shows how RevNIC implements symbolic hardware.

3.2 Mechanics of Exercising the Driver

RevNIC attempts to maximize the basic block coverage of the driver, in order to maximally capture its behavior. RevNIC first determines the entry points of the driver and makes the OS invoke them, in order to initiate their symbolic execution. Then, RevNIC guides the symbolic execution using a set of heuristics whose goal is to maximize coverage while reducing the time spent exercising the driver.

RevNIC's requirements. To exercise the driver, RevNIC must know the semantics of the OS interface. This requires that the OS driver interface and all API functions used by the driver be documented. The documentation must include the name of the API functions, the parameter descriptions, along with information about data structures (type and layout) used by these functions. RevNIC internally encodes this information in order to correctly determine and exercise the entry points provided by the driver.

Discovering driver entry points. RevNIC monitors OS-specific entry point registration calls. In the case of Windows, drivers usually export one function, which is called by the kernel to load the driver. This function registers the driver's entry points with the OS by passing a data structure to a specific OS function (`NdisMRegisterMiniport`). At run-time, the driver can register other entry points, like timers (via `NdisInitializeTimer`). RevNIC monitors calls to such OS APIs to record the contents of the data structures and function pointers. Since these structures contain actual function pointers and have documented member variables, RevNIC knows which entry points need to be exercised and the developer is aware of the functionality each entry point is responsible for. RevNIC includes a default set of NDIS function descriptions and allows users to specify what additional functions to monitor.

RevNIC invokes each entry point of the driver via a user-mode script or program that runs in the guest OS. The script first loads the driver so as to exercise its initialization routine, then invokes various standard IOCTLs, performs a send, exercises the reception, and ends with a driver unload. Interrupt handlers are triggered by the VM, as we shall see shortly. Once an entry point is called, its code is executed symbolically until no more new code blocks are discovered within some predefined amount of time.

Initiating symbolic execution. RevNIC intercepts entry point invocations, then fills with symbolic data the user buffers and the integer parameters passed in, while keeping the other parameters, like pointers, concrete. For example, to exercise sending, RevNIC runs a program that sends packets of various sizes. RevNIC catches the invocation of the send entry point, then replaces the concrete data within the packet and the packet length with symbolic values. This exercises the paths corresponding to the various packet types and sizes. Existing techniques can be employed to mix concrete and symbolic data within the same buffer [17, 19], in order to speed up exploration.

Injecting symbolic values from the OS side, however, may cause execution to reach an impossible error state (e.g., wrongly crash the driver) if the symbolic values subsume concrete values the kernel would never pass to the driver. When any error state is reached, RevNIC terminates the execution path and resumes a different one. Reaching these infeasible error states does not perturb the reverse engineering process, since RevNIC merely aims to touch as many basic blocks as possible and cause them to manifest in the traces. RevNIC's goal is not to expose the driver to a realistically functioning device or OS, but rather to reverse engineer the state machine implemented by the driver.

Guiding driver exploration with heuristics. Symbolic execution generates a large number of possible execution paths, with execution having progressed to various depths down each path. RevNIC executes one path at a time, but frequently switches between them. The choice of which path to execute next is driven by a strategy that, in RevNIC, relies on several heuristics. RevNIC's heuristics aim to choose the paths most likely to lead to previously unexplored code. Discarding early on paths that are unlikely to discover any new code helps cope with the large number of paths. Note that RevNIC allows these heuristics to be modularly replaced, when and if better ones are discovered.

The first heuristic explicitly selects paths most likely to discover new code. Every time RevNIC completes executing one basic block of the driver, it decides whether to continue that same execution path (by executing the next basic block) or to switch to a basic block in another execution path. We refer to a <path,block> tuple as "state," as it directly determines program state. RevNIC associates with each basic block a counter that is incremented after that block is executed. The next state to execute is the one corresponding to the basic block with the lowest count. A good side effect of this strategy is that it does not get "stuck" in loops, since it decreases the priority of states that merely re-execute a previously explored loop. We found this heuristic to speed up exploration, compared to depth-first search (which can get stuck in polling loops) or breadth-first search (which can take a long time to complete a complex entry point).

A separate heuristic selects paths to discard in polling loops and large entry points. Symbolic execution forks virtually identical states at a high rate in case of polling loops. To avoid memory exhaustion, RevNIC keeps the paths that step out of the polling loops and kills those that go on to the next iteration. A large number of states can also get RevNIC stuck in one entry point, preventing subsequent entry points from ever being reached. To cope with this, whenever an entry point completes with a successful error code a given number of times, RevNIC discards all paths except one successful one chosen at random. The execution then proceeds to the next entry point, controlled by the script.

A third heuristic guides RevNIC in injecting interrupts at specific points in the execution, to exercise interrupt handlers. For NIC drivers, triggering interrupts after returning from a driver entry point (e.g., send) works well, since that is the moment when the device either triggers a completion interrupt, a receive interrupt, or some other type of interrupt (error, buffer overflow, etc.). This strategy results in virtually complete coverage of the interrupt handlers. In general, however, the code paths in asynchronous event handlers depend on previous execution histories—we are exploring ways to use data dependency information to optimally choose the moment to inject such asynchronous events.

A final heuristic helps RevNIC skip over unnecessary function calls. First, device drivers often call functions irrelevant to the hardware protocol, such as writing debug/log information (via calls like `NdisWriteErrorLogEntry`). Such (OS-specific) functions can be indicated in RevNIC's configuration file, and RevNIC will skip them. Second, some hardware-related functions can be replaced with models, to speed up execution. E.g., a common pattern is to write a register address on one port and read the value on another. This type of function may be called frequently, yet it exercises the same set of paths on every call. RevNIC can report the most frequently called functions on a first run, in order to let the developer specify which ones should be replaced with models on subsequent runs. Such models have a few lines of code and are easy to write: they just need to set the program counter appropriately to skip the call, and return a symbolic value (e.g., in case the modeled function is a register read).

3.3 Wiretapping the Driver

RevNIC exercises the driver, so that the wiretap can see and record as much behavioral information as possible.

First, the wiretap saves the instructions executed by the driver in an intermediate representation. This serves as a basis for C code generation during the synthesis phase (§4). Second, the wiretap records whether the instructions access device-mapped memory or regular memory, along with the value of the corresponding pointer and the transferred data. This simplifies the data flow analysis during reverse engineering (§4.1), by disambiguating aliased pointers. Third, the wiretap records the type of executed basic blocks (conditional vs. direct/indirect jumps vs. function calls) and the contents of the processor's registers at the entry and exit of each basic block. This helps reconstruct the control flow during synthesis.

3.4 Exerciser/Tracer Prototype

The driver exerciser/tracer part of RevNIC has three components, all of which extend the QEMU hypervisor [2]: one initiates symbolic execution starting from a concretely running OS environment, another one performs selective symbolic execution on the driver using a modified version of KLEE [6], and the last one optimizes symbolic execution in order to make driver exploration efficient.

Creating the illusion of real hardware. To make the OS load the driver and initiate symbolic execution, RevNIC uses a "shell" virtual device in the hypervisor to create the illusion that the actual device is present. The shell device redirects the driver's hardware accesses to a symbolic execution engine, which provides symbolic input for each read from that device. The engine also triggers symbolic interrupts by asserting the interrupt pin of the shell device. The shell device consists of a PCI configuration space descriptor, which contains crucial information for loading the corresponding driver: the vendor and product identifier of the device whose driver is being reverse engineered, the I/O memory ranges, and the interrupt line. The developer obtains these parameters from the Windows device manager and passes them to RevNIC on the command line. RevNIC can reverse engineer x86 binary drivers for PCI or ISA devices; adding support for other buses like USB is straightforward.

The shell device relies on RevNIC to detect and handle DMA memory. The shell device cannot handle DMA directly because it does not know how the original device would handle DMA internally. Drivers use specific APIs to register memory to be used in DMA operations. RevNIC detects DMA memory regions by tracking calls to the DMA API and communicating the returned physical addresses to the shell device, which returns symbolic values upon reads from these regions.

Symbolic execution. RevNIC executes the driver symbolically and runs the rest of the system (OS kernel and applications) concretely. It monitors OS attempts to load the driver, in order to track the location of the driver code. RevNIC then parses the operating system's data structures directly from inside the hypervisor. It does not require any modification of the drivers or the guest OS, but needs to know the type of the OS data structures and their location.

RevNIC passes the driver code to a dynamic binary translator (DBT) to generate equivalent blocks of LLVM bitcode [26]—the intermediate representation used by RevNIC in its traces and used for symbolic execution in KLEE. QEMU passes the current program counter to the DBT, which translates the code until it finds an instruction altering the control flow. Then, the DBT packages the translated LLVM bitcode into a *translation block*[1], which is ready for symbolic execution. The DBT cannot translate all the code at once, because the code may not be available in advance (e.g., it could be paged out, it might be self-modifying).

The generated blocks of bitcode are run by the KLEE symbolic execution engine in the context of the current state. A state consists of the contents of the physical memory, the CPU registers, and the virtual devices. KLEE updates the state as it executes a block of LLVM code (i.e., write to CPU registers, memory, and devices according to the executed code). When encountering branches that depend on symbolic values, the current state is copied, conceptually similar to what the `fork` syscall does. When KLEE completes the execution of the block of code, it returns the updated current state to QEMU, along with the set of forked states. RevNIC's heuristics then choose the next state to execute.

RevNIC enables QEMU to switch between the concrete and symbolic execution domains. First, QEMU and KLEE share a common representation of the physical memory. This memory can store symbolic values that occur during symbolic execution of the driver. RevNIC automatically con-

[1] A translation block is a sequence of guest machine instructions translated to LLVM, ending with a terminator (call, return, branch, etc.). A translation block can consist of multiple basic blocks, when an instruction in the middle of the translation block is the target of a branch from outside that block.

cretizes the symbolic values whenever they are read by the OS, thus keeping the OS unaware of symbolic execution. Second, RevNIC copies the CPU state between QEMU and KLEE whenever execution crosses the concrete/symbolic boundary. A calling convention that uses registers to pass parameters would require concretizing symbolic register values. In Windows, the `stdcall` calling convention passes all arguments on the stack, which means that only registers with concrete values need to be copied.

Coping with a large state space. The last component helps RevNIC cope with two aspects that compound the state explosion problem: symbolic memory addresses and the 4 GB limit on system memory in 32-bit machines.

RevNIC avoids the complexity of dealing with symbolic addresses by concretizing them. Keeping track of all possible addresses for each memory access that uses a symbolic location is expensive. Symbolic memory addresses occur when symbolic input is used to reference memory, e.g., when a symbolic IOCTL number is used as an index in a table. They also occur with jump tables produced by compilers for switch statements. Depending on the number of case statements, a compiler constrains the range of values (e.g., $0 \leq i < 16$) by putting a check in the code, and jumping to the default case when the value is out of range. Since there are typically only a few concrete values, RevNIC generates all of them and forks the execution for each such value.

Symbolically executing drivers can generate tens of thousands of states. Even though RevNIC is compiled to use the full 4 GB of virtual address space, RevNIC requires more memory. We augmented KLEE's object-level copy-on-write with page-level copy-on-write (to avoid forking large memory objects) and added transparent page swapping to disk (to explore a virtually unlimited number of states, without being restricted by available physical memory). 64-bit versions of QEMU and KLEE would alleviate this problem.

4. Synthesizing New Drivers

RevNIC exercises the driver and outputs a trace consisting of translated LLVM blocks, along with their sequencing and all memory and I/O information. Now we describe how this information is processed (§4.1) and used to synthesize C code for a new driver that behaves like the original (§4.2).

4.1 Turning Traces into a C-encoded State Machine

Generating C code from the traces consists of rebuilding the control flow graph of the driver's functions and converting the corresponding basic blocks from LLVM to C.

Rebuilding the CFG. The driver wiretap produces raw execution traces that contain explicit paths through the driver's execution tree (§3.1). Each such path, from the root to a leaf, corresponds to an execution of the driver, exercising a different subset of the code. The trace does not contain OS code, because RevNIC stops recording when execution leaves the driver. A traced path ends when it is terminated by RevNIC (e.g., due to being stuck in a polling loop), when driver initialization fails, or when the unload routine of the driver completes (and thus there is nothing more left to execute). The traces also contain interspersed snippets of asynchronous execution, like interrupt and timer handlers.

RevNIC merges the execution paths from traces in order to rebuild the state machine (i.e., control flow graph) of the original driver. A CFG contains all the paths that a driver could traverse during its execution. To build a CFG equivalent to that of the original driver, it is sufficient to execute at least once each basic block of the driver. Building is done in two steps: First, RevNIC identifies function boundaries by looking for call-return instruction pairs. Second, the translation blocks between call-return pairs are chained together to reproduce the original CFG of the function. RevNIC splits translation blocks into basic blocks in the process.

Execution paths can contain manifestations of asynchronous events that disrupt the normal execution flow. RevNIC detects these events by checking for register value changes between two consecutively executed translation blocks. The register values are saved in the trace before and after the execution of each block. RevNIC builds the CFG of each such event just like for normal functions.

From CFG to C code. The output of the CFG builder is a set of functions in LLVM format. The last phase turns these functions into C code, reconstructs the driver's state, and determines the function parameters, the return values, and the local variables. Listing 1 shows a sample of generated code. The control flow is encoded using direct jumps (`goto`) and all function calls are preserved.

```c
void function_12606(uint32_t GlobalState)
{
  //Local variables
  uint32_t Vars4[4];

  Vars4[3] = 0x0;
  //Driver's state is accessed using pointer arithmetic
  Vars4[2] = *(uint32_t*)(GlobalState + 0x10);
  write_port32(Vars4[2] + 0x84, Vars4[3]);
  //Remainder omitted...
}
```

Listing 1: Generated code sample (annotated).

RevNIC preserves the local and global state layout of the original driver (Listing 1). Drivers usually keep global state on the heap, a pointer to which is passed to the driver upon each entry point invocation. To access their global state, they use offsets from that pointer. Binary drivers access local variables similarly, by adding an offset to the stack frame pointer. The synthesized code preserves this mechanism by keeping the pointer arithmetic of the original driver.

RevNIC determines the number of function parameters and return values using standard def-use analysis [10] on the collected memory traces. Since the traces contain the actual memory access locations and data, it is possible to trace back the definition of the parameters and the use of the possible return values. To determine whether a function f has a return value, RevNIC checks whether there exists an execution path where the register storing the return value[2] is used without being redefined after f returns. The number of parameters is determined by looking for memory accesses whose addresses are computed by adding an offset to the stack frame pointer, resulting in an access to the stack frame of the parent function.

The generated code may be incomplete if the driver is not fully covered (i.e., the code has an incomplete CFG). Incompleteness manifests in the generated source by branches to unexercised code. RevNIC flags such branches to warn the developer. Missing basic blocks happen for driver functions containing API calls whose error recovery code is not usually exercised. It does not affect the synthesized driver, since error recovery code is part of the template (§4.2). However, in the case when code for hardware I/O is missing, the developer can request QEMU's DBT to generate the missing translation blocks by forcing the program counter to take the address of the unexplored block. Since RevNIC does not execute such blocks, they do not appear in the execution trace: the developer must insert the code for these blocks manually, which slows down the reverse engineering process.

4.2 From State Machine to Complete Drivers

Producing a new driver consists of pasting the synthesized C code into a driver template. A template is written once and can be reused as long as the OS/driver interface does not change. The template contains all the boilerplate to communicate with the OS (e.g., memory allocation, timer management, and error recovery). Depending on the OS, more or less boilerplate code may be required (e.g., a driver for an embedded OS typically has less boilerplate than an NDIS driver). The boilerplate can also vary depending on the OS version. E.g., a template could use the newer NAPI network model on Linux, or the older API. This only affects the organization of the template. Besides the boilerplate, the template also contains placeholders where the actual hardware I/O code is to be pasted. Listing 2 shows a fragment of the `init` function from the Linux NIC template, which we use as a running example. RevNIC includes a NIC template for each supported target OS.

Drivers typically have four types of functions. The first type corresponds to functions that only call hardware I/O routines (e.g., `write_port32` in Listing 1) or other hardware functions: register read-modify-write, disable interrupts on a NIC, read the MAC address, etc. The second type consists of OS-dependent functions that assemble hardware-dependent routines to perform a high-level functionality. E.g., a `send` would call a function that sets the ring buffer index, then call the OS to get the packet descriptor, after which it would invoke another hardware-specific routine, passing a pointer and a size to transmit the packet. The third type is similar to the second, except that it mixes hardware accesses with calls to the OS. This happens, e.g., when the driver inlines hardware functions. Finally, the fourth type includes functions that implement OS-independent algorithms, such as checksum computation.

```
int revnic_pci_init_one (struct pci_dev *pdev,
    const struct pci_device_id *ent)
{
  /* Variable declarations omitted */

  // The template first allocates PCI device resources
  if (pci_enable_device (pdev)) {
    // OS-specific error handling provided by template
  }
  ioaddr = pci_resource_start (pdev, 0);
  irq = pdev->irq;
  if (request_region(ioaddr, ADDR_RANGE, DRV_NAME)) {
    // OS-specific error handling provided by template
  }

  // Then the template allocates persistent state.
  // A pointer to this state is passed to each
  // reverse engineered entry point.
  dev = alloc_netdev(/*..*/, /*..*/, ethdev_setup);
  if (dev) {
    // OS-specific error handling provided by template
  }
  memset(netdev_priv(dev), 0, /*..*/);
  SET_NETDEV_DEV(dev, &pdev->dev);
  revnic = REVNICDEVICE(dev);

  // The synthesized functions may expect specific state
  // (e.g., the I/O address) to be initialized. Here,
  // the I/O address is stored at offset IOADDR_OFFSET.
  revnic->Private.u4[IOADDR_OFFSET] = ioaddr;

  //**********************************************
  // Developers paste calls to RevNIC-synthesized
  // hardware-related functions here.

  // A driver may want to check the hardware first...
  if (ne2k_check_device_presence(&revnic->Private) < 0) {
     // Error recovery provided by the template
     // (e.g., unload the driver)
  }

  // ...before initializing it
  if (ne2k_init(&revnic->Private) < 0) {
    //Device-specific recovery synthesized by RevNIC...
    ne2k_shutdown(&revnic->Private);
    // ...followed by template-provided recovery code.
    // (e.g., unload the driver)
  }
  //**********************************************

  // More OS-specific initialization goes here.
  // Initialize IRQ, I/O addresses, entry points, etc.
  // ...

  // Template adapts the driver's data structures to
  // the target OS. Here, it copies the MAC address
  // from driver's memory to the Linux data structure.
  // Adaptation is done by the driver developer.
  for (i=0; i<MAC_ADDR_LEN; i++) {
    dev->dev_addr[i] = revnic->Private.u1[0x14b + i];
  }
  register_netdev(dev);
  return 0;
}
```

Listing 2: Example `init()` routine of the Linux NIC template (edited for brevity).

[2] This is specific to the Windows `stdcall` calling convention; other conventions can be implemented as well.

Given a NIC driver template, a developer inserts the calls to OS-independent functions generated by RevNIC into the template. The amount of effort required to build a working driver is usually minimal: look at the context in which the functions were called in the original driver (in an interrupt, a send packet routine, a timer entry point, etc.) and paste them in the corresponding places in the template. The developer also has to adapt various OS-specific data structures to the target OS, e.g., to convert the Windows `NDIS_PACKET` structure to the equivalent `sk_buff` Linux structure. This is the most time-consuming part of reverse engineering, but could be simplified by annotating the generated code with type information (e.g., based on the source OS's header files). The developer also needs to match OS-specific API calls to those of the target OS. In Windows, such APIs and structures are public and documented.

Filling in a template is straightforward when only functions of types 1, 2, and 4 occur in the traces. However, when hardware I/O is mixed with OS-specific code (type 3), more effort is required. Without RevNIC, the developer would have to look at the disassembly or the decompiled code to understand what the driver does. This requires distinguishing regular from device-mapped memory accesses, understanding complex control flow, and grasping the interaction with the OS. Instead, RevNIC provides the developer with execution traces annotated with hardware I/O, which can be used to retrace the execution instruction by instruction. This makes it easier to understand the interaction between the driver's components and eases integration into the template.

NIC driver templates follow common patterns, which we found to be similar across different OSes. E.g., the `init` entry point implemented by the Linux, Windows and μC/OS-II templates first allocates device resources, calls a hardware-specific function that checks for the presence of the hardware, registers the device, and brings it to its initial state. Likewise, an interrupt handler in these three OSes first calls a hardware routine to check that the device has indeed triggered the interrupt, before handling it.

Common template patterns facilitate driver synthesis for multiple platforms. E.g., each template contains one lock to serialize the entry points (this ensures correct operation but may affect performance). The developer strips all OS-specific locks that might be present in the original driver, because they are not needed anymore. Then, the developer pastes that code in the same places across all templates, without worrying about target OS-specific synchronization.

Ideally, the merging of the synthesized driver code with the template would be fully automated. However, the process of translating from one OS to another requires refactoring the original driver's OS-specific functions and translating API calls to fit them in the generic template for the target OS. While human developers can guess quite easily how to translate these, an automated translator would need to correctly reconstruct the driver binary's missing type information to understand how the driver manipulates the data structures (e.g., lists) in order to adapt them to the target OS.

5. Evaluation

In this section we address several important questions: Do RevNIC-generated drivers have functionality (§5.2) and performance (§5.3) equivalent to the original drivers? How much effort does the reverse engineering process entail (§5.4)? How far can RevNIC scale (§5.5)? To answer these questions, we use RevNIC to port four closed-source proprietary Windows NIC drivers to three other operating systems as well as back to Windows, producing a total of 11 driver binaries. At no time in this process did we have access to the drivers' source code.

5.1 Experimental Setup

We first present the evaluated drivers, describe the three target operating systems, and give details on the hardware used for measurements.

Evaluated Drivers. We used RevNIC to reverse engineer the Windows drivers of four widely used NICs (Table 1). Three of the four ship as part of Windows, attesting to their popularity. The drivers range in size from 18KB to 35KB, which is typical for NIC drivers in general (e.g., 80% of network drivers in Linux 2.6.26 are smaller than 35KB). The number of functions the drivers implement ranges from 48 to 78 and the number of used OS API functions ranges from 37 to 51. The Windows driver files for AMD PCNet, RTL8139, SMSC 91C111, and RTL8029 are `pcntpci5.sys`, `rtl8139.sys`, `lan9000.sys`, and `rtl8029.sys`, respectively. Linux has equivalent drivers for the same chipsets: `pcnet32.c` (2300 LOC), `8139too.c` (1900 LOC), `smc91x.c` (1300 LOC), and `ne2k-pci.c` / `8390.c` (1200 LOC).

Target Platforms. We use RevNIC to port the PCNet, RTL8139, and RTL8029 drivers to Linux 2.6.26, and 91C111 to μC/OS-II. This shows RevNIC's ability to port drivers between systems with different APIs of varying complexity. It also enables a comparison of the performance of synthesized drivers to that of the native drivers on the target OS.

We used RevNIC to port all drivers to our custom operating system, called KitOS, running on "bare" hardware. This OS initializes the CPU into protected mode and lets the driver use the hardware directly, without any OS-related overhead (no multitasking, no memory management, etc.). This experiment evaluates the performance of the synthesized drivers in the absence of OS interference. Bare hardware is the mode in which RevNIC would be used during the initial development of new drivers (KitOS boots instantly and starts executing immediately the driver, thus shortening the compile/reboot cycle, allowing developers to fix driver bugs quicker). Once the driver works properly, the developers can "transplant" the driver to the target OS.

Reverse Engineered Windows Driver	RevNIC Ported from Windows to ...	Driver Size	Code Segment Size	Imported Windows Functions	Functions Implemented by the Original Driver
AMD PCNet	Windows, Linux, KitOS	35KB	28 KB	51	78
Realtek RTL8139	Windows, Linux, KitOS	20KB	18 KB	43	91
SMSC 91C111	μC/OS-II, KitOS	19KB	10 KB	28	40
Realtek RTL8029 (NE2000)	Windows, Linux, KitOS	18KB	14 KB	37	48

Table 1: Characteristics of the proprietary, closed-source Windows network drivers used to evaluate RevNIC.

We also ported the PCNet, RTL8139, and RTL8029 drivers back to Windows XP SP3. Porting to the same OS enables quantifying the overhead of the generated code with respect to the original Windows driver. In practice, porting to the same OS is useful when the binary driver exists for one version but not the other (e.g., 32-bit vs. 64-bit Windows), or when the original driver causes the OS to crash or freeze.

Test Hardware. We evaluate the performance of the synthesized drivers by running them on an x86 PC, an FPGA-based platform, and two virtual machines. This allows us to measure performance of generated drivers in a wide range of conditions. The PC and VMs run fresh installations of Windows XP SP3, Debian Linux 2.6.26, and KitOS. The FPGA system runs the μC/OS-II priority-based preemptive real time multitasking OS kernel for embedded systems.

We measure the performance of the RTL8139 driver on a PC based on an Intel Core 2 Duo 2.4 GHz CPU with 4 GB of RAM. The physical NIC is based on a Realtek RTL8139C chip, widely used in commodity desktop systems.

We evaluate the 91C111 driver on the FPGA4U [36] development board. It is based on an Altera Cyclone II FPGA with a Nios II processor, 32 MB of SDRAM, and an SMSC 91C111 network chip. The FPGA and the SDRAM run at 75 MHz, while the 91C111 chip runs at its native frequency of 25 MHz. This allows quantifying the overhead on a severely resource-constrained system.

Finally, we evaluate the RTL8029 driver on QEMU and the PCNet driver on VMWare. Virtualization is seeing increasing use in networked computing infrastructures, so performance in such an environment is important. The virtual machines are QEMU 0.9.1 and VMWare Server 1.0.10. The host OS is Windows XP x64 edition SP2 in both cases, running on a dual quad-core Intel Xeon CPU at 2 GHz, with 20 GB of RAM. QEMU uses a TAP interface for networking, while VMWare runs a NAT interface. VMs allow us to better zoom in on driver bottlenecks, which can be harder to observe on a real machine. For example, VMs disregard the rated speed of the NIC, so one can send data at even 1 Gbps using a driver for a 100 Mbps NIC (since there is no physical cable, the virtual NIC can confirm transmission immediately after the driver has given it all the data).

5.2 Effectiveness

RevNIC can extract all essential functionality from network device drivers. Table 2 shows the capabilities of the original NIC drivers compared to those of the reverse engineered drivers. A check mark indicates functionality available both in the original and the synthesized driver.

We identify the functionality implemented in the original driver by looking at the `QueryInformation` status codes supported by Windows, checking the configuration parameters in the registry that reveal additional functionality, and looking at the datasheets. For RTL8029 and PCNet, given that the virtual hardware does not have LEDs and does not support Wake-on-LAN, we could not directly test these functions. However, the corresponding code was exercised and reverse engineered. The RTL8029 and the 91C111 chips support neither DMA nor Wake-on-LAN.

Functionality	AMD PCNet	RTL8139	SMSC 91C111	RTL8029
Init/Shutdown	✓	✓	✓	✓
Send/Receive	✓	✓	✓	✓
Multicast	✓	✓	✓	✓
Get/Set MAC	✓	✓	✓	✓
Promiscuous Mode	✓	✓	✓	✓
Full Duplex	✓	✓	✓	✓
DMA	✓	✓	N/A	N/A
Wake-on-LAN	N/T	✓	N/A	N/A
LED Status Display	N/T	✓	✓	N/T

Table 2: Functionality coverage of reverse engineered drivers (N/A=Not available, N/T=Cannot be tested).

We manually checked the correctness of the reverse engineered functionality by comparing hardware I/O operations. For this, we ran the original driver on real hardware and recorded its I/O interaction with the device. Then, we ran the reverse engineered driver and compared the resulting I/O traces with that of the original driver. We exercised each function using a workload specific to the functionality in question. E.g., to check send and receive, we transmitted several files via FTP. Checking the packet filter (i.e., promiscuous mode) involved issuing standard IOCTLs.

Finally, we manually checked that the original driver is a correct encoding of the hardware protocol specification. For this, we compared I/O interaction traces with the I/O sequence prescribed by the hardware specification. We focused on the send/receive functionality, since it is crucial for a network driver. We did not find meaningful discrepancies between the collected sample traces and the specifications.

5.3 Performance

We evaluate the performance of the reverse engineered drivers by measuring throughput and CPU utilization. We first compare the original Windows driver to the synthesized Windows driver, in order to quantify the overhead of the code generated by RevNIC. Then we show the performance of drivers ported to a different operating system. We wrote a benchmark that sends UDP packets of increasing size, up to the maximum length of an Ethernet frame. In the case of KitOS, the benchmark transmits hand-crafted raw UDP packets, since KitOS has no TCP/IP stack. The reverse engineered drivers turn out to have negligible overhead on all platforms.

Figure 2 shows throughput and Figure 3 shows CPU utilization for the RTL8139 drivers. Synthesized drivers incur practically no overhead. The driver for KitOS is the fastest, since there is no TCP/IP stack overhead. For unknown reasons, the original Windows driver's performance drops for UDP packets over 1 KB; the reverse engineered driver does not have this problem. We also observe that the synthesized Windows driver has a slightly higher CPU utilization than the original, while both the native Linux and the ported Linux driver have a similar one for most packet sizes.

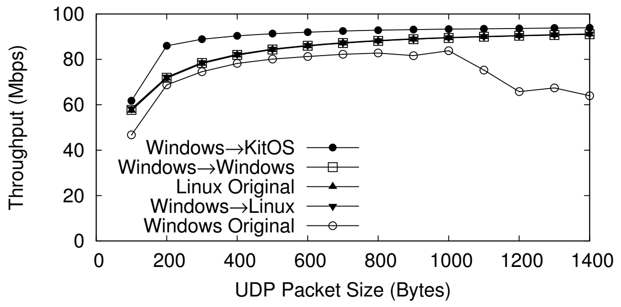

Figure 2: RTL8139 driver throughput on x86.

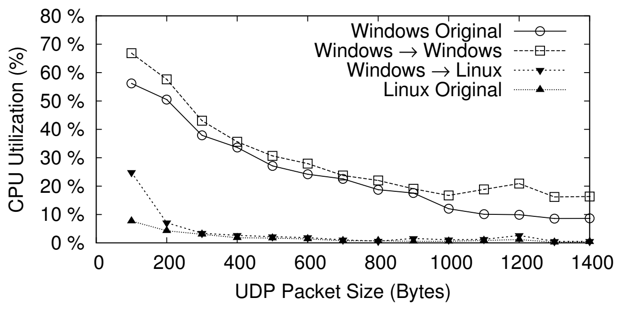

Figure 3: CPU utilization for RTL8139 drivers on x86.

Turning our attention to embedded systems, we note that synthesizing a driver for severely resource-constrained environments is one of the toughest performance challenges for RevNIC. Original drivers are typically hand-optimized, whereas RevNIC's drivers are not. In Figure 4, we show the

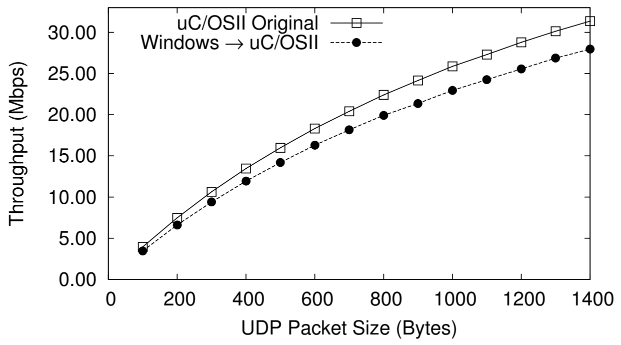

Figure 4: 91C111 driver ported from Windows to an FPGA.

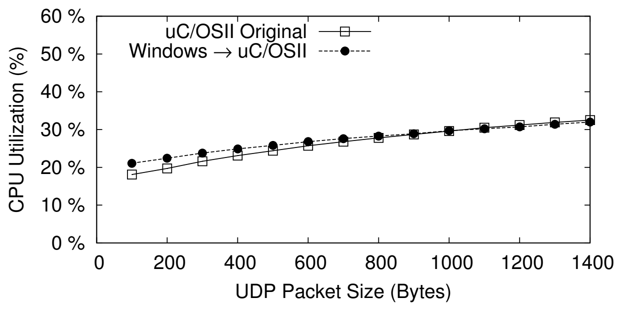

Figure 5: CPU fraction spent inside the 91C111 driver.

performance of the 91C111 driver ported to the FPGA platform. Throughput is within 10% of the original driver, and we suspect this difference is mainly due to its cache footprint: the RevNIC-generated binary has 87KB, compared to 59KB for the native driver. With further optimizations on the generated code, we expect this 10% gap to be narrowed. CPU time spent in the synthesized 91C111 driver is comparable to that of the original (Figure 5), ranging roughly from 20% to 30% for both drivers. The overall CPU usage is 100%, since DMA is not available. The maximum achievable throughput is limited by the FPGA's system bus, shared between the NIC, the SDRAM, and other components.

Finally, Figure 6 and Figure 7 show performance in virtualized environments. For QEMU, we show the RTL8029 driver, since QEMU provides an RTL8029-based virtual NIC. CPU utilization is close to 100% in all cases, since RTL8029 does not support DMA. The driver ported from Windows to Linux is on par with the native Linux driver. The lean KitOS driver again has the highest throughput. The difference between Linux and Windows is due to different behavior of TCP/IP stack implementations in the VM.

For VMware, which provides an AMD PCNet virtual NIC, we get similar results. Even though DMA is used, CPU utilization is still 100% in all cases, because the virtual hardware sends packets at maximum speed, generating a higher interrupt rate than that of real hardware. Performance

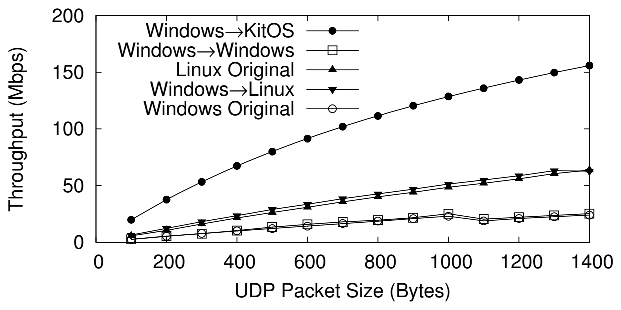

Figure 6: RTL8029 throughput (QEMU).

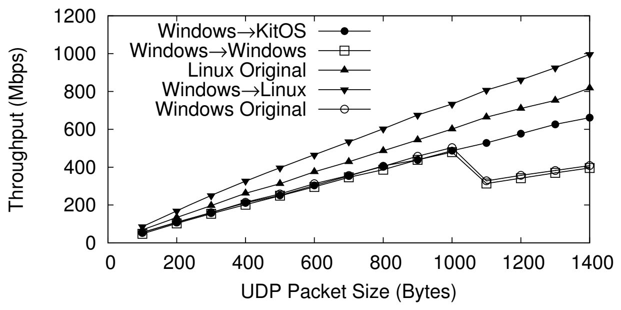

Figure 7: AMD PCNet throughput (VMWare).

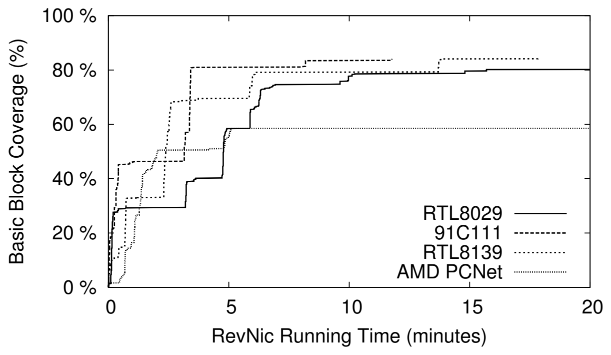

Figure 8: Basic block coverage.

Target OS	Person-Days
Windows	5
Linux	3
μC/OS-II	1
KitOS	0

Table 3: Time to write a template.

on KitOS is lower, but same as that of the original Windows driver, most likely due to interactions with VM quirks.

5.4 Automation

RevNIC exercises drivers and generates code in less than an hour. Template instantiation, though manual, takes orders of magnitudes less time than writing new drivers from scratch and does not require obtaining, reading, and understanding hardware specifications.

Obtaining portable C code. Obtaining the code for OS-independent and hardware-specific functionality is fully automated and fast. In Figure 8, we show how driver coverage varies with RevNIC running time—most tested drivers reach over 80% basic block coverage in less than twenty minutes, due to our use of symbolic execution. RevNIC stops either when all hardware-related functions get close to 100% coverage, or when a specified timeout expires (§3.2).

The running time and memory usage of the RevNIC code synthesizer is directly proportional to the total length of the traces it processes. RevNIC can process a little over 100 MB/minute. For the drivers we tested, code synthesis took from a few seconds to a few minutes.

Writing a driver template. Producing NIC driver templates for the four OSes took a few days (Table 3). Writing a template is done manually, but it is a one-time effort, considerably simplified by using existing driver samples shipped with SDKs.

We first wrote one generic template for all NIC devices, and then extended it to provide DMA functionality for RTL8029 and 91C111. Running the driver in KitOS does not require a template, since the driver can directly talk to the hardware, without interacting with the OS. The template for μC/OS-II took only one day, because this embedded OS has a simple driver interface.

Integrating Hardware Interaction Code in the Template. A large portion of drivers' code is hardware-specific. In Figure 9, we show what fraction of the driver is fully reverse-engineered by RevNIC. Overall, about 70% of the functions are fully synthesized. The other functions contain mostly OS-related code and correspond to high-level functions of the device drivers, like send and receive. They also include functions that mix OS and hardware calls (~10%–15% per driver). These functions are only partly exercised, and the corresponding traces serve as hints to the developer for the template integration.

Writing drivers, even from specifications, is hard. Table 4 shows how much effort it took to write and/or debug Linux open source drivers. We looked at how many developers were acknowledged in the headers of the source files and at the reported time span for development. The numbers also include adaptations to newer versions of Linux. Even when assuming that developers do not work full-time on a driver, but on a best-effort basis (like in the open source community), it still takes a considerable effort. The change logs of the RTL8139 driver suggest that most time went into coding workarounds for undocumented hardware quirks.

In contrast, RevNIC uses the original proprietary driver, which has all the hardware details for all supported devices

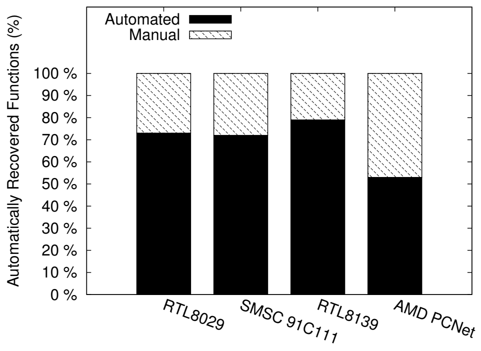

Figure 9: Breakdown of OS-specific vs. hardware-specific functions (in percentage of recovered functions).

readily available. When using RevNIC, most developer time goes into instantiating the driver template. This is roughly proportional to the size of the driver and the number of hardware functions it implements. Table 4 also includes the time to debug RevNIC, since porting the drivers and debugging our prototype were done together. Debugging required manually checking the synthesized C code against the original driver's binary; this took 1-3 days, depending on driver size.

Device	Manual (Linux)		RevNIC	
	Persons	Span	Persons	Span
RTL8139	18	4 years	1	1 week
SMSC 91C111	8	4 years	1	4 days
RTL8029	5	2 years	1	5 days
AMD PCNet	3	4 years	1	1 week

Table 4: Amount of developer effort. RevNIC numbers include time to debug the RevNIC prototype itself.

5.5 Scalability

We have shown that our approach works for drivers from 18KB to 35KB. This is representative of most NIC drivers.

Scalability is limited by the performance of symbolic execution. Symbolic execution is subject to exponential state growth and memory consumption [6, 7, 34, 39], both of which affect RevNIC. We are developing orthogonal techniques that will potentially alleviate these problems, for example by running symbolic execution (and RevNIC) on large clusters [11], and carefully selecting what data should be considered symbolic [8]. Improvements in the field of symbolic execution will automatically benefit to RevNIC.

6. Discussion and Limitations

RevNIC cannot produce a driver that is "more correct" than the original binary with regards to hardware interaction. It is hard to fix buggy hardware interaction, when there are no specifications. However, certain classes of bugs, like unchecked use of array indexes coming from hardware or buffer overflows are eliminated by reverse engineering, resulting in a safer driver.

Reverse engineering of proprietary IOCTLs is similar to that of standard entry points. IOCTLs encode functions that were not foreseen by the OS driver interface designers. Instead of triggering proprietary behavior using standard OS APIs (e.g. `send`/`receive`), RevNIC uses the vendor-supplied configuration tools for doing so. For the standard interface, the semantics of the behavior are provided by the OS interface; for the proprietary one, the semantics are derived from the tool's documentation.

RevNIC-generated code is not as readable as the original source, because it does not reconstruct high-level C statements, like loops. The generated code relies on `goto` for control flow. We believe that existing transformation techniques [10] can make generated code more readable. However, the produced code is substantially more accessible than disassembly. The generated code is easier to understand and adapt, because it uses familiar C operators, instead of x86 instructions. Moreover, C code can be easily compiled to any OS or processor architecture, unlike assembly.

Of course, RevNIC can be rerun easily every time there is an update to the original binary driver. The resulting source code can be compared to the initially reverse engineered code and the differences merged into the reverse engineered driver, like in a version control system. One could also use binary diffing methods [5] to update the synthesized driver every time there is a new patch for the proprietary driver that fixes hardware-related bugs. Thus, we expect RevNIC-generated code to require minimal maintenance.

Although porting between two OSes by instantiating a driver template requires substantial code refactoring, one could automate it to a certain extent. E.g., Coccinelle [33] automatically translates device drivers between two versions of the same OS. RevNIC could treat two different OSes as an evolution from one to the other. Another possibility is to synthesize a specification from the binary (instead of code) and use existing tools, like Termite [37], to automatically generate a driver for any target OS, solving once and for all the safety and portability problems of device drivers.

RevNIC currently supports NIC drivers, but it is in theory possible to extend it to any class of device drivers. Exercising the driver and generating the code is device-agnostic: all RevNIC needs is OS and hardware input. The developer has to write a device driver template for the new class of devices, and this requires a general understanding of what the device class is supposed to do and how it interfaces with the OS (e.g., that a sound card is supposed to play sound by copying a buffer to some memory, very much like a NIC sends a packet after it is copied to some buffer).

Finally, RevNIC is not meant for reverse engineering the internals of a device, only its interaction with the driver. For instance, devices like graphics cards can compile and run code internally (such as vertex shaders). Reverse engineering firmware or the particular programming language of a chip is beyond the scope of RevNIC. What a tool like RevNIC could do for a graphics driver is to extract the initialization steps and set up a frame buffer, possibly extracting 2D acceleration, if it only involves I/O. It could also make a synthesized driver replay hardware interactions (e.g., upload firmware to the card) the same way the original driver would.

7. Related Work

Building portable drivers has been a goal of several previous projects, including Devil [28], HAIL [40] and UDI [35]. Recent work, like Termite [37], proposes a formal development process in which a tool generates drivers from state machine specifications. These approaches require vendors to provide formal specifications of the hardware protocols. RevNIC complements these efforts by extracting the encoding of the protocol from existing device drivers, making the task of reverse engineering existing drivers more productive. In some sense, RevNIC can help tools like Termite become practical. VM-based approaches [27] can reuse existing binary drivers, but are generally heavyweight. Other approaches can directly reuse existing drivers by emulating the source OS, e.g., NDISwrapper [32]. However, the emulation layer has to be updated with each version of the source OS, is prone to bugs, adds overhead, and works only on the OS for which it was developed. In contrast, RevNIC makes the reverse engineered driver independent from the source OS.

Most of the existing techniques for improving device driver safety rely on source code [42–44]. Since RevNIC can obtain a source code representation of a driver binary, it can enable the use of all these tools on closed-source, proprietary device drivers, to improve their reliability. E.g., some binary drivers do not have proper timeouts in polling loops; this would be straightforward to fix using [23]. OS-related safety properties could be checked prior to compilation [1], or the driver could be split to enhance reliability [16]. Furthermore, if the driver generation follows a formal development approach (as in [15] or [37]), it is possible to guarantee that a reverse engineered driver will not crash the system, will not hang, and will not introduce security vulnerabilities.

In using VMs to observe system activity, we build upon a rich set of prior work, including tools such as Aftersight [9] and Antfarm [21]. Reverse debugging [25] used VMs to debug systems, including device drivers. Symbolic execution has also been used for program testing [6, 7, 18, 39] and malware analysis [12, 31, 45]. We extended these approaches to provide kernel-mode instrumentation. In RevNIC, we combine VM-based wiretapping with symbolic execution to exercise control on the analyzed system. Reverse engineering often uses static decompilation [3]; this, however, faces a number of challenges (e.g., disambiguating code from data), so we minimized RevNIC's reliance on static decompilation.

Recent work has been aimed at automatically reverse engineering message formats in network protocols [13] as well as files [14], based on traces containing these messages. Our reliance on driver activity traces is similar but, due to the specifics of device drivers, RevNIC manages to also reverse engineer the relationship between hardware registers, not just the format. RevNIC extracts the semantics of driver code dynamically, using traces of memory accesses.

8. Conclusion

In this paper, we presented a new approach and tool for reverse engineering binary device drivers. One can use this approach to either port drivers to other OSes or to produce safer drivers for the same OS.

The tool, called RevNIC, does not require access to any device documentation or driver source code—it relies on collecting hardware interaction traces and synthesizing, based on these, a new, portable device driver. We showed how RevNIC reverse engineered several closed-source Windows NIC drivers and ported them to different OSes and architectures. RevNIC is easy to use and produces drivers that run natively with performance that is on par with that of the target OS's native drivers.

9. Acknowledgments

We are indebted to our shepherd, Julia Lawall, and the anonymous reviewers for helping us improve our paper. We thank Vlad Georgescu and Volodymyr Kuznetsov for their contributions to RevNIC's development. We are grateful to Daniel Dunbar and Cristian Cadar for sharing an early version of KLEE and for answering related questions.

References

[1] 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 *EUROSYS Conf.*, 2006.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[3] Boomerang decompiler. http://boomerang.sourceforge.net/.

[4] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symp. on Security and Privacy*, 2008.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conf.*, 2008.

[10] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[11] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[12] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Symp. on Operating Systems Principles*, 2007.

[13] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering. In *USENIX Security Symp.*, 2007.

[14] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Conf. on Computer and Communication Security*, 2008.

[15] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Workshop on Hot Topics in Operating Systems*, 2007.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Conf. on Programming Language Design and Implementation*, 2008.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symp.*, 2008.

[20] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com.

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conf.*, 2006.

[22] Jungo. WinDriver device driver development tookit, version 9.0. http://www.jungo.com/windriver.html, 2007.

[23] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Symp. on Operating Systems Principles*, 2009.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[25] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[27] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symp. on Operating Systems Design and Implementation*, 2004.

[28] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Symp. on Operating Systems Design and Implementation*, 2000.

[29] Microsoft security advisory #944653: Vulnerability in Macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[30] Microsoft Windows Driver Kit. http://www.microsoft.com/whdc/devtools/WDK, 2009.

[31] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symp. on Security and Privacy*, 2007.

[32] NDISwrapper. http://ndiswrapper.sourceforge.net, 2008.

[33] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EUROSYS Conf.*, 2008.

[34] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[35] Project UDI. Uniform Driver Interface. http://udi.certek.com/, 2008.

[36] R. Beuchat, P. Ienne et al. FPGA4U. http://fpga4u.epfl.ch/.

[37] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Symp. on Operating Systems Principles*, 2009.

[38] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Working Conf. on Reverse Engineering*, 2002.

[39] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Symp. on the Foundations of Software Engineering*, 2005.

[40] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: a language for easy and correct device access. In *Intl. Conf. on Embedded Software*, 2005.

[41] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), 2006.

[42] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[43] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Symp. on Operating Systems Principles*, 1993.

[44] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Symp. on Operating Systems Design and Implementation*, 2008.

[45] H. Yin, H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Conf. on Computer and Communication Security*, 2007.