SymDrive: Testing Drivers Without Devices

Matthew J. Renzelmann, Asim Kadav and Michael M. Swift
Computer Sciences Department, Under submission: Please do not redistribute
University of Wisconsin-Madison
{*mjr,kadav,swift*} @*cs.wisc.edu*

Abstract

Device driver development and testing has traditionally been a complex and error-prone undertaking. For example, dozens of patches committed to the Linux kernel include the phrase "compile tested only," suggesting that driver testing is often not conducted at all. Furthermore, few open-source Linux device drivers have any driver-specific tests. Finally, driver reliance on specific hardware significantly increases testing complexity, as testing a single driver might require dozens of different devices for all code paths to be executed. These considerations set device-driver testing apart from user-level application testing, which does not impose such demands.

We have developed a new system for testing Linux PCI and USB drivers called SymDrive. The system employs symbolic execution and a driver test infrastructure to enable driver testing without device hardware present. We see two major uses of SymDrive. First, developers can validate and verify patches to ensure that kernel programming rules are correctly followed and errors handled correctly. Second, developers can verify that refactoring and interface changes, often made without testing, do not change driver behavior. We tested SymDrive on seven sound and network drivers, using both PCI and USB interfaces, and found seven bugs in these drivers.

1 Introduction

Device drivers are difficult to test and debug for several reasons. They run in kernel mode, which prohibits the use of many program-analysis tools, such as Valgrind [28], that user-mode developers routinely employ. Driver bugs can cause system crashes that are tedious to debug, such as when interrupts are mistakenly left disabled. They execute in a heavily concurrent environment with extensive synchronization requirements and employ complex memory management techniques. Most importantly, developers cannot test device drivers without a supported device.

The need for hardware often prevents testing altogether: core kernel developers often do not have access to the hardware needed by device drivers. Revisions to the driver/kernel interface occasionally necessitate large, cross-cutting changes affecting many drivers, and the hardware requirements for an individual to test all of these changes could be very high [30]. In these cases, driver code is left without sufficient testing.

Making the situation worse, individual drivers routinely support dozens of distinct pieces of hardware. For example, the Intel E1000 driver in the Linux 2.6.29 kernel supports over 60 distinct network-processing chips. Some chips have limitations that the driver must work around, while others are of newer designs that employ different I/O protocols from earlier implementations. Support for many devices complicates testing: to test a driver thoroughly, a developer must test with each piece of supported hardware.

A second challenge in testing driver code is the difficulty of exercising error-handling code. This code may only be triggered if the device reports an error or malfunctions, such as by returning an invalid value. For example, one of the 18 supported medium access controllers in the E1000 driver requires an additional EEPROM read operation while configuring flow-control and link settings. Testing the error handling in this driver requires the test suite to consider specifics of each supported chip, which may become costly.

The difficulty of testing drivers manifests in the revision control logs for drivers and the Linux mailing lists. A search through the Linux kernel's driver revision history for the phrase "compile tested only" suggests that patches have been incorporated frequently even though the developer was unable or unwilling to execute the patch, which strongly suggests a need to enable broader testing of drivers

We developed *SymDrive*, a system for testing device drivers without the associated device hardware, to improve the quality of driver code. SymDrive uses *symbolic execution* to simulate all possible hardware inputs to a device driver while also eliminating the need for the device to be present. Should a failure along a specific branch of execution take place, SymDrive reports the precise set of data produced by the device that leads to the failure. However, symbolic execution alone does not specify what constitutes a failure.

SymDrive provides a test framework for conducting a partial verification and validation of the driver's implementation. This framework provides assertions over driver behavior, state variables for tracking the driver's logical state, and an object tracker to record the status of memory objects. At every control transfer between the driver and the kernel, the test framework interposes pre- and post-condition evaluation. Furthermore, the test framework infers the driver's class from its behavior, allowing class-specific checking of device drivers.

We target SymDrive at the Linux driver development process. Using a large database of bugs and kernel programming requirements culled from code, documentation, and mailing lists, we constructed 47 checkers to validate and verify driver code. These checkers enforce rules that maintainers commonly check during code reviews: matched allocation/free calls, matched lock/unlock calls, memory leaks, and proper use of kernel APIs. We also provide checkers to detect problems that commonly require subsequent patches, such as vulnerability to denial-of-service resource leaks. Finally, we provide a mechanism to compare the behavior before and after a patch, to verify that the interaction of the driver and device does not change. This is particularly helpful for collateral evolutions [29], when interface changes require driver modifications that commonly are not tested.

SymDrive is closely related to the recent DDT project [22]: both systems provide symbolic execution of drivers. The primary difference is that DDT targets verification by end users, and therefore supports binary code and a small set of generic checks, such as proper memory and lock use. In contrast, SymDrive targets the development process, and focuses on testing specific patches. It includes a comprehensive test framework that allows sophisticated tests, such as those that ensure the driver uses APIs properly, and a differencing mechanism, to determine how a patch changes driver behavior.

We implement SymDrive using KLEE [9], User-Mode Linux [12], and Microdrivers [16]. We applied SymDrive to seven drivers (sound and network, USB and PCI buses), and found seven bugs. Overall, we found that SymDrive can:

- find bugs in drivers, such as memory leaks, incorrect uses of the kernel API, hardware dependence bugs, and mismatches between memory allocator and free routines.
- provide assurance that a refactored driver interacts with the hardware the same way as an unmodified driver along multiple execution paths.
- achieve 100% code coverage in complex driver functions.
- find several forms of generic bugs, such as illegal pointer dereferences, similar to other systems [7, 13, 14, 21, 22, 27].

The paper is organized as follows. In the next section, we describe the design of SymDrive and how the problem affects the design of the system. In Section 3 we discuss how to use SymDrive to test device drivers. Section 4 provides implementation details of SymDrive, and section 5 evaluates its effectiveness. Section 6 discusses related work, and Section 7 concludes.

2 Design

The SymDrive architecture focuses on thorough testing of driver *patches* to ensure that any new code meets its specifications. This reflects the evolutionary nature of development, in which developers check in a new module and then gradually evolve it over time through patches. Even small drivers, such as the NE2000 network driver, have had dozens of patches in the last several years.

Goals The two primary goals of SymDrive are to allow the developer to (i) verify and validate any relevant part or parts of a driver to the greatest extent possible, while (ii) eliminating the need for device hardware during testing. The first goal enables *deeper* testing of drivers for higher quality code, while the second goal enables *broader* testing of drivers, by many more people and on many more machines.

2.1 Design Overview

SymDrive addresses these two goals using a combination of *symbolic execution* and a fine-grained *test framework*. SymDrive uses symbolic execution to execute device-driver code without the device being present. As a driver executes, any input from the device is replaced with a *symbolic value*, which represents all possible values the data may have. When symbolic values are compared, SymDrive forks execution and executes all branches of the comparison, each with the symbolic value constrained by the chosen outcome of the comparison. For example, the predicate $x > 5$ forks execution into one path where $x \leq 5$ and one where $x > 5$. Symbolic execution provides two benefits: executing drivers without hardware, and testing error handling code when the hardware returns unusual values. When the driver interacts with the kernel, SymDrive creates a single, *concrete value* for symbolic values passed to the kernel by choosing one of their possible values.

The test framework interposes on driver/kernel control transfers, and executes a different *checker* for each function in the driver/kernel interface. At each transfer, the framework invokes a test function that evaluates test conditions for the function. Thus, the test framework provides operational specifications of driver behavior. We have coded many basic specifications for different driver classes and standard driver routines, including locks and data allocation. Developers can use the test framework features to add more specifications to test new code.

SymDrive executes drivers down a single execution path until the driver is unloaded or a developer-specified point. Then, SymDrive systematically explores other execution paths symbolically. Along each path through the driver, SymDrive checks all driver entry point and kernel function pre- and post-conditions. Any failure terminates

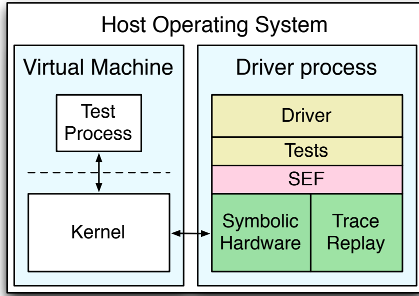

Figure 1: **The SymDrive architecture.**

the current execution path with a detailed error report, and execution continues along the next path.

The implementation of SymDrive is shown in Figure 1. The OS kernel executes in a virtual machine and communicates with the driver under test, executing in a separate process, over IPC. The *symbolic execution framework* (*SEF*) governs the driver's execution as it runs symbolically. SymDrive provides *stubs* in the kernel and in the driver to copy data between the virtual machine and the driver.

2.2 Driver Interactions

A device driver communicates with two distinct entities: the device, through I/O requests, and the kernel, through function calls. We next describe how SymDrive handles both kinds of interaction.

2.2.1 Driver/Device Interaction

Drivers interact with devices according to well-defined, narrow interfaces. For PCI device drivers, this interface is comprised of I/O memory, port I/O, DMA memory, and interrupts. For USB drivers, the interface uses USB request blocks (URBs) that are sent to and from the device. While most previous driver research focuses on PCI devices, we include USB devices as they are prevalent, and USB versions of most standard devices (*e.g.*, network, storage, sound, and video controllers) are available.

For PCI devices, SymDrive provides symbolic data of the appropriate size each time the driver performs a read operation using memory or port I/O. Similarly, SymDrive treats the contents of DMA memory (indicated through DMA mapping functions) as symbolic. Whenever the driver reads from DMA memory, SymDrive provides unconstrained symbolic data. SymDrive provides symbolic interrupts by directly invoking the driver's interrupt handler. Thus, SymDrive effectively simulates the presence of a piece of hardware, albeit one that might also return unexpected values.

USB drivers behave differently because they use an asynchronous packet-based protocol to communicate with the device. These packets, URBs, are used both to send and receive data. A driver submits an URB to the kernel USB framework, which then sends it through a USB host controller to the device. Packet submission is quick, and control returns to the driver immediately, before the device has a chance to respond. The URB includes a completion routine that the kernel invokes when the device sends its response. SymDrive provides symbolic USB hardware with symbolic URBs: a symbolic USB device sends a packet of symbolic data, including a constrained symbolic size to allow responses of different lengths. Similar to interrupts, SymDrive invokes completion routines spontaneously.

2.2.2 Driver/Kernel Interaction

The driver also interacts with the kernel through calls to kernel functions for service. To ensure that symbolic execution is faithful to normal execution, SymDrive must ensure that the kernel responds appropriately when the driver invokes these functions. However, a difficulty arises when the driver forks symbolic execution: it can now invoke the kernel twice, on different execution paths. As described above, with SymDrive the kernel executes normally, preventing it from forking execution. Thus, the kernel can only be invoked along one of the many paths.

One solution to this problem is to fork the kernel's execution state when the driver itself forks, as in DDT [22]. However, this approach is not possible with our approach to symbolic execution, as it would require forking the virtual machine running the OS, which is both time and memory intensive.

Instead, SymDrive supports two methods of interaction with the kernel with different benefits and drawbacks. Developers can choose the execution mode on a per-kernel function basis, and can vary it from one test run to another.

Concrete + Symbolic In this approach, the driver executes kernel functions normally along one execution path through the driver, using a depth-first-search approach to choosing an execution path. Once this path terminates, all subsequent calls to the specified kernel function do not actually invoke the kernel; instead, return values and structure fields modified by the kernel function are marked symbolic. We determine the set of fields through static analysis of the kernel. Thus, the driver will execute with all possible (and a few more) return values. This execution strategy ensures that kernel functions that cause callbacks to the driver, such as registration functions (*e.g.*, `pci_register_driver`), correctly invoke the callback function. However, false positives are possible, as the driver can execute with return values the kernel would never generate.

Symbolic only In this approach, the kernel is never invoked, even on the first path. Again, structure fields of any function parameters that the kernel might modify are marked as symbolic. This approach provides more thorough testing of driver code, because it is not constrained by a specific set of concrete values returned by the kernel.

As an example, suppose that the developer wishes to test the network driver `probe` function thoroughly. In this case, the developer could set the `pci_register_driver` to call into the kernel using the *concrete + symbolic* option, thus causing the kernel to invoke the driver's probe function. For all other kernel functions, the developer could use the *symbolic only* technique, to ensure that maximum code coverage is possible. By default, all kernel functions execute using the *concrete + symbolic* approach, which we have found represents an effective solution.

2.3 Limiting Path Explosion

Symbolic execution, by executing all possible paths through the kernel, can lead to very long testing times. Two important reasons for this are reentrancy, which causes multiple threads to execute in the driver simultaneously, and lengthy, control-heavy driver initialization routines.

2.3.1 Driver reentrancy

Reentrancy in drivers arises for a variety of reasons, including interrupts, timer and work-queue callbacks, URB completions, and simultaneous calls from different threads. To explore all possible paths fully in the presence of reentrance, the symbolic execution framework would have to invoke all possible functions at every instruction boundary in a driver, which could cause a huge explosion of paths to explore.

To make testing more tractable, SymDrive restricts reentrant calls to occur only when control is transferred between the driver and the kernel. For example, SymDrive invokes the driver's interrupt handler both before calling a driver function and when the driver function returns. Similarly, USB drivers communicate with hardware asynchronously using the event-driven URB packet interface. Since SymDrive does not allow the driver to communicate with hardware, it instead calls completion routines at the same time it calls interrupt handlers: whenever control transfers between the driver and kernel.

For simultaneous calls into the driver, SymDrive restricts concurrency by only allowing one thread into the driver at a time; other threads block in the kernel while the thread executes. The same approach is used for timer and work-queue callbacks to prevent them from being delivered while a thread executes in the driver. This approach effectively serializes access to the driver, though, and prevents SymDrive from effectively testing for concurrency bugs.

```
e1000_config_dsp_after_link_change(…) { …
  if (hw->ffe_cfg_state == e1000_ffe_cfg_active) {
    ret_val = e1000_read_phy_reg(hw, 0x2F5B, &saved);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
    if (ret_val) return ret_val;
    mdelay(20);
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    mdelay(20);
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    hw->ffe_config_state = e1000_ffe_config_enabled;
  } …
}
```

Figure 2: **State space explosion with symbolic execution. Each if-statement forks a new path of execution since `ret_val` is symbolic.**

2.3.2 Driver initialization

State explosion arises when symbolic values used in conditional statements cause execution to fork repeatedly, once for each path through the code. Each path results in a new symbolic execution state. Figure 2 shows a piece of driver code in which each conditional forks execution. Too many states cause symbolic execution to either take a long time to complete, or to stop testing some states. The problem is particularly acute for drivers that contain many conditional statements for checking device response status bits and chipset identifiers, especially in initialization code. Drivers often support many specific devices, and routinely contain extensive branching logic to implement this support.

SymDrive provides a *replay* mechanism to fast-forward driver execution to the point of interest, such as the code affected by a patch. First, a developer with access to hardware creates a trace of all driver/device interactions with a trace tool. The trace includes all data provided by the hardware, and enough context information to allow it to be used for replay. Later, during testing, SymDrive can either provide symbolic values or provide values from the trace tool, which allows the driver to execute concretely on a single path.

Furthermore, device-driver initialization often relies on a large number of highly specific inputs from the device in order to proceed. For example, the E1000 network driver code shown in Figure 3 calculates a checksum over the device EEPROM data, and only continues if the checksum is valid. Forcing symbolic execution to derive the necessary constraints to satisfy the checksum calculation is infeasible, but replay provides a sim-

```
unsigned short eeprom_data, i;
for (i = 0; i < (EEPROM_CHECKSUM_REG + 1); i++) {
    e1000_read_eeprom(..., &eeprom_data);
    checksum += eeprom_data;
}
if (checksum == (u16)EEPROM_SUM)
    return E1000_SUCCESS;
else return -E1000_ERR_EEPROM;
```

Figure 3: **Symbolic execution is not appropriate for checksum calculations, since performance would be poor and the developer would gain little insight into the driver's behavior.**

ple way to fast-forward over this code by providing the real EEPROM data from the device. Replay also simplifies testing of local changes: rather than testing all driver execution, including initialization, a developer may replay device interactions up to the changed code, and then switch to symbolic mode. Note that while the device is needed to generate the trace, this can be done once, and the trace can then be used to execute tests on a machine where the device is not present.

2.4 Limitations

SymDrive's approach to addressing driver reentrancy and kernel interactions has the advantage of simplicity, but limits its bug-finding capacity. Reduced reentrancy prevents SymDrive from detecting race conditions and deadlocks. However, various other static and dynamic approaches have demonstrated success at finding these synchronization errors, thus reducing the need for SymDrive to address the same problem [13, 15]. As processing power and symbolic execution techniques improve, it may become reasonable to reduce or eliminate this restriction, by instead simulating how driver functions would behave if their executions were interleaved. SymDrive's handling of kernel interactions may lead to false positives, since symbolic return values can lead the driver to execute paths not possible with a more limited set of real kernel return values.

3 Testing Device Drivers

The second major component of SymDrive is a *test framework* used to determine whether a symbolically executing driver is behaving correctly. Symbolic execution can detect whether a driver will perform an illegal operation, such as reference an invalid address, on any execution path through a function. However, this proof does not provide any general correctness guarantees: a "hello world" program will not crash but will also not bring up an Ethernet interface properly.

Thus, achieving high-quality driver code requires both *validating* and *verifying* its operation to the extent practical. *Validation* establishes that the driver is solving the right problem and demonstrates that the driver does what it is supposed to do, such as initializing a data structure or registering with the kernel. In contrast, *verification* establishes that the implementation follows the rules to solve the problem. It ensures the absence of known bugs such as memory or lock leaks and null pointer dereferences. With SymDrive, we seek to ensure both driver verification and validation takes place. In the interests of simplicity, we will refer to these two processes collectively as "testing" in this section.

Executing both the driver and the kernel symbolically provide only a limited degree of validation and verification. The kernel contains relatively little error checking code because it trusts drivers to operate correctly; incorrect behavior by a driver often manifests as a resource leak or a malfunctioning device, rather than as an illegal operation detectable by symbolic execution.

3.1 Use Cases

We target SymDrive at the driver development process in Linux, in which developers submit patches to maintainers, who review the code before accepting it into the kernel. However, the design applies to closed-source development as well, in which driver developers perform their own code reviews.

What should SymDrive test? We identify the most promising use cases for SymDrive by studying sources of errors in Linux driver code. To find examples of errors, we scanned mailing lists and reviewed driver revision histories in the kernel source repository.

First, we scanned over 2,000 messages posted to several Linux development mailing lists [3, 4] for messages relating to driver patches. These mailing lists are a combination of proposed patches and developer discussion. This exercise unveils two types of bugs: (1) those that are in the kernel source and are fixed by the proposed patch, (2) and those in the proposed patch themselves. The latter type often results in rejected patches, since other developers spot errors in the patch before the patch is accepted. In addition, the discussion associated with each patch provides clues about the kinds of bugs that testing misses. In addition to mailing lists, we examined several hundred entries in the kernel's revision history associated with various device drivers. These patches are examples of the first type of bug.

Checking patches The primary emphasis of SymDrive is to test patches to driver code. As discussed previously, Linux driver maintainers receive patches via mailing lists, and each requires review before the driver maintainers can merge them with the kernel. For example, the Linux Driver Project [3] hosts a mailing list to which anyone can submit driver patches. Developers without access to device hardware frequently author these patches, and are unable to test them thoroughly as a re-

sult. Dozens of patches are currently incorporated into the kernel that were "compile tested only." Other developers often spot problems before they incorporate the patch into the kernel.

SymDrive instead allows more thorough testing of patches *before* they are submitted, saving valuable time spent on code reviews. Many checks that reviewers perform manually can be coded with SymDrive. For example, maintainers often look for memory leaks on error paths, which can be checked by SymDrive. A check, once part of SymDrive, can then be applied to subsequent drivers either by developers themselves of by maintainers. Thus, developers can verify that proposed patches satisfy common driver programming requirements, without requiring as much manual review.

Verifying refactorings and collateral evolutions A second target of SymDrive is to verify that driver code refactoring and collateral evolution is correct. Many of the patches to drivers reorganize it for better readability and maintainability or to follow kernel programming guidelines. For example, a single developer submitted well over 100 patches to the `et131x` driver that refactored the code without changing its functionality. Such extensive refactoring could clearly benefit from additional testing, and would enhance developers' ability to refactor drivers when the hardware is unavailable. These refactorings should not change the code semantically.

Collateral evolutions, which occur when a driver interface changes, are another use case for SymDrive. These interface changes require widespread modifications to driver code, for example to add a new parameter. However, the changes are often made by a single developer without access to hardware, so they may not be tested. With SymDrive, these changes can be verified, and hence even larger interface changes could be possible.

3.2 Testing with Specifications

To support the above-mentioned use cases, SymDrive provides developers the means to specify a set of conditions that all drivers of a particular class must enforce. The test framework checks these specifications as SymDrive executes the driver. For example, all network drivers must call the `register_netdev` function during initialization, unless initialization fails. No network driver should claim initialization was successful and fail to call this function. Symbolic execution by itself can never catch these kinds of bugs, because of their domain-specific nature. Similarly, normal testing has difficulty verifying these conditions, because they must hold across *all possible* execution paths, not just the one that occurred during testing.

The test framework is comprised of (i) a set of call outs into test code, where conditions can be checked, (ii) an API library of code to check common conditions, and (iii) checks for specific conditions on specific interfaces. The test framework interposes call outs on all interfaces between the driver and kernel and allows tests to execute whenever control passes between the kernel and the driver, in either direction. While this allows tests to be developed independently of driver code, it limits the framework for testing conditions that are visible at the driver/kernel interface. The test framework executes symbolically, so it evaluates conditions along each branch of execution independently.

Test-framework call outs The test framework invokes a test function before and after every call into the driver. Each function in the driver/kernel interface provides a unique test function. Before entering the driver, test code can establish preconditions, and when leaving the driver, test code can verify postconditions that must hold.

Test-framework API The test framework provides an API library containing utility code for common classes of tests. Using this API, checks can remain small and readily understandable. The API provides four classes of functions, which we describe more in Section 4.5. First, the API provides basic supporting functions to querying the current driver's bus and class. This allows checks that apply only to certain classes of drivers, such as PCI network drivers. Second, the API provides a function to report current execution context, indicating whether blocking is safe. Third, the API provides methods to record state variables, such as whether the driver has completed initialization. Finally, the API provides an object tracker, which records the state of memory regions (*e.g.*, how was it allocated and how big is it?) or locks (*e.g.*, is it initialized?).

With this API, writing additional tests is straightforward. For example, verifying that a custom allocator is paired with a custom free routine needs only a single call to the test framework API to tell it about the allocation. The test framework will then automatically support all drivers that use the new allocator, and the drivers themselves require no changes.

Test-framework checks The test framework API library provides a number of common checks, similar to checking for illegal operations. For example, it verifies that all objects that are freed were first allocated, and that locks released were first acquired. However, we expect driver developers to use the API to develop their own tests.

4 Implementation

This section describes the implementation of SymDrive, which includes the components outlined in Figure 1. SymDrive consists of five major components:

- A *virtual machine* for executing the kernel under test and a test program.

- *Stubs* for remote communication between the kernel and the driver.
- A *symbolic execution framework* for tracking symbolic data values and constraints, with symbolic hardware.
- A *trace generator* for creating traces of driver/device interaction and a replayer for use during testing.
- The *test framework* and associated API for invoking test code before and after invoking the driver.

We next describe the implementation of each component in turn.

4.1 Virtual Machine

SymDrive uses User Mode Linux (UML) as a virtual machine to provide a model of the kernel's execution. As UML does not provide any hardware support, SymDrive implements virtual PCI and USB buses that cause the kernel to invoke the driver's functionality. These virtual buses invoke driver entry points, and provide easily configurable virtual devices to support operation of the desired drivers.

Creating a virtual PCI device requires a few basic identifiers, such as the device manufacturer and class. This information is available in existing device drivers.

In contrast, creating a simulated USB device is more involved. When a USB device is inserted, the USB core code in Linux exchanges several packets with the device to obtain the device's supported configurations, interfaces, and endpoints, in addition to the basic information of manufacturer and device class. SymDrive includes a tool to reproduce the details of the actual device in a virtual equivalent, allowing this information to be replayed to the virtual USB bus during testing.

4.2 Remote Driver Execution

SymDrive reuses DriverSlicer from Microdrivers [16], to enable the device driver to execute outside of the kernel. DriverSlicer generates stubs and marshaling code to transfer data and control between the driver and kernel appropriately. This control and data transfer takes place via an RPC mechanism over named pipes. This approach appears to the kernel as an architecture-specific feature that pauses kernel execution, which enables communication with the driver process even for high-priority code such as the packet transmit routine in network drivers.

In order to marshal data, a developer must annotate kernel data structures indicating how pointers are used. For example, character pointers must be annotated to distinguish between a null-terminated string, an array of bytes, and a single character. These annotations allow DriverSlicer to generate code to marshal and unmarshal data structures properly. Fortunately, the annotations are almost exclusively in kernel code, which need only be annotated once. DriverSlicer limitations also necessitate additional annotations in individual drivers in some cases, but DriverSlicer tells the developer where the annotation is necessary and how to write the annotation, making them easy to add.

4.3 Symbolic Execution with KLEE

SymDrive uses a modified version of KLEE [9] for symbolic execution. KLEE provides the execution environment and constraint solving capability necessary for symbolic execution. It also manages the forking that takes place as symbolic values are compared against each other. All driver code, including the test framework, executes using KLEE. The kernel executes natively, without KLEE.

By default, KLEE employs a depth-first search strategy (DFS) to explore the state space from the symbolic execution. To resolve issues related to driver loops performing repeated symbolic read operations, SymDrive switches to a breadth-first search strategy (BFS) once the developer unloads the driver. Without using BFS, SymDrive would waste too much time exploring uninteresting execution paths. DDT employs a similar heuristic that favors unexecuted code, though the purpose is the same [22].

4.4 Replay

The replay mechanism provides device inputs for the driver from a *hardware trace* to fast forward it to the point of interest. SymDrive includes a tracer that modifies a driver to log all interactions with the device. Instrumenting a driver requires the developer to add a single C preprocessor directive to the top of each driver file. The resulting instrumented driver uses customized device I/O routines for logging hardware interaction.

Running the instrumented driver on a system with compatible hardware produces a log of all device I/O operations. In the case of PCI drivers, the trace indicates the function name and line number of the original read operation, as well as the I/O operation type, such as `inb`, and the port number and data read. For USB drivers, the trace includes all the URBs received by the driver instead, as well as results from synchronous USB device interface functions such as `usb_control_msg`. The trace is a simple text file that developers can store in a source repository or share over the Internet.

Executing the driver with SymDrive will use the trace when it provides relevant data for the driver (*e.g.*, the next trace entry is for the correct hardware read operation, port, and driver function). Otherwise, SymDrive provides symbolic values. When a developer wants to test specific driver functionality, he or she comments out parts of the trace, thus forcing SymDrive to supply symbolic data at the desired point.

```
void __pci_register_driver_check(...) { // Test #1
  if (precondition) {
    assert (state.registered == NOT_CALLED);
    set_state (&state.registered, IN_PROGRESS);
    set_driver_bus (DRIVER_PCI);
  } else /* postcondition */ {
    if (retval == 0) set_state (&state.registered, OK);
    else set_state (&state.registered, FAILED);
  }
}

void __kmalloc_check // Test #2
  (..., void *retval, size_t size, gfp_t flags) {
  if (precondition)
    mem_flags_test(GFP_ATOMIC, GFP_KERNEL, flags);
  else /* postcondition */
    generic_allocator(retval, size, ORIGIN_KMALLOC);
}

void _spin_lock_irqsave_check // Test #3
  (..., void *lock) {
  // generic_lock_state supports pre/post-conditions
  generic_lock_state(lock,
    ORIGIN_SPIN_LOCK, SPIN_LOCK_IRQSAVE, 1);
}
```

Figure 4: **Example tests. The first test is common to all PCI drivers, and ensures that all PCI drivers are registered exactly once. The second verifies that the driver allocates memory with the appropriate `mem_flags` parameter. The third ensures lock/unlock functions are properly matched.**

4.5 Test framework

The test framework is responsible for calling checkers and provides an API library to support common test functions. As described in Section 3, the test framework is implemented as a set of call-out functions, invoked both before and after every driver entry point and kernel function called from the driver. DriverSlicer, when generating marshaling code, automatically generates empty call-out functions to be populated with checks.

4.5.1 Writing checkers

Writing a checker involves implementing any checks within the call-out function. Test #1 in Figure 4 shows an example call-out for `pci_register_driver`. The test framework invokes the checker function with the parameters and return value of the function and sets a `precondition` flag to indicate whether the checker was called before or after the function. In addition, the test framework provides a global `state` variable to track the driver state across multiple functions. As shown in this example, a checker can verify that the state is correct as a precondition, and update the state based on the result of the call. The test framework also provides an `assert` function that signals a bug if the predicate fails along any execution path and causes the path to stop execution and print a stack trace.

The library API provides additional support routines to simplify checkers by providing common functionality to track the state of a memory object, such as which function allocated it, or a lock, to record if it has been initialized or acquired. Test #2 and #3 illustrate example checkers, which call the `generic_allocator` routine to record allocations, and `generic_lock_state` to track the state of a lock.

4.5.2 Example Checkers

We have implemented checkers for a variety of common device-driver bugs using the test framework and library API.

Execution Context Linux and other operating systems prohibit the use of some calls when executing as part of an interrupt handler or while holding a spinlock. In particular, calls to allocators must specify whether they can block or whether the call must return immediately. The execution context checker verifies that flags passed to memory-allocation functions such as `kmalloc` are valid in the context of the currently executing code. For example, if the driver is executing the `start_xmit` path of a network driver, it can only allocate memory with the `GFP_ATOMIC` flag.

Checking the driver's execution context requires tracking the context of each call into the driver, which is not available during normal execution. Thus, the test framework API provides a state machine to track the driver's current context. Each time the kernel invokes the driver, the test framework updates the driver's current execution context. The test framework API tracks execution context using a stack. Several events cause the test framework to push a new state onto the stack. For example, a driver invoked in a context that supports blocking, such as its `probe` routine, enters a non-blocking context if it acquires a spinlock. Thus, each time the driver acquires or releases a spinlock, the test framework API pushes or pops the necessary context. This mechanism simplifies checking which operations the driver is allowed to conduct, and which it is not.

User-Invokable Allocations Driver code that allocates memory or prints a log message in response to a request may lead to resource denial-of-service attacks [2]. For example, if the driver calls `printk` with a high priority message, but an unprivileged user could invoke it, the checker will display an error because the user could execute a DOS attack by filling up the system error log.

To detect these bugs, SymDrive tracks whether an unprivileged user could directly invoke the current code. We manually identified driver entry points that can be invoked through system calls, such as `read` for character drivers, and expose this through the driver's current execution context. Thus, the test framework API allows developers to generalize this test to support other cases in which an unprivileged user can repeatedly cause the driver to allocate a resource.

Kernel API Misuse The kernel requires that drivers follow the proper protocol when using kernel APIs, and errors often lead to a non-functioning driver or a resource leak. For example, all PCI drivers must call `pci_register_driver` in the driver's `init_module` initialization routine, and make a corresponding call to the `pci_unregister_driver` function in its `cleanup_module` routine. Similarly, USB drivers require a call to `usb_register_driver`, which SymDrive can verify such drivers make.

The test framework state variable provides additional context for these tests. For example, a checker can track the success and failure of significant driver entry points, such as the `init_module` and PCI `probe` functions, and ensure that if the driver is registered on initialization, it is properly unregistered on shutdown. Furthermore, checkers can verify that the driver is in the required state when invoked from the kernel. Test #1 in Figure 4 shows a use of these states to ensure that a driver only invokes `pci_register_driver` once.

SymDrive checks for a variety of straightforward conditions using this state machine approach. Other checks include those shown in Table 1.

Collateral Evolutions Another use of the state machine is verifying that collateral evolution patches are correctly applied. Tools such as Coccinelle [29] aid in generating patches, but do not verify their correctness. Since individual patch authors often do not have access to all of the necessary devices, SymDrive can provide a substantial benefit by testing these changes.

SymDrive can verify that collateral evolutions are correctly applied by ensuring that patched drivers do not regress on any tests. In addition, a developer can add a check to ensure that the *effect* of a patch is reflected in the driver's execution. As an example, recent kernels stopped requiring that network drivers update the `net_device->trans_start` variable in their `start_xmit` functions. A SymDrive checker can verify this change by checking that `trans_start` is constant across the `start_xmit` function and generate a warning if the driver modifies this field.

Memory Leaks Memory leaks are a pernicious problem in drivers, and it is particularly difficult to find small leaks among the sea of other allocations. SymDrive provides an object tracker, implemented as a simple hash table, that stores memory address, length, object state, and a description of the object's origin. With the tracker, a checker can record object allocation and deallocation, and detect even small memory leaks. We have implemented checkers to verify allocation and free requests from 12 pairs of functions, as well as other special cases, such as SKBs passed into a network driver's `start_xmit` function that the driver is then responsible for freeing.

The API library simplifies adding support for additional allocation and deallocation routines down to adding a one-line call into the API. Test #2 in Figure 4 shows the `generic_allocator` call to the library used when checking `kmalloc`, which records that `kmalloc` allocated the returned memory. A corresponding checker for `kfree` verifies that `kmalloc` allocated the supplied address.

SymDrive's object tracker provides considerably more flexibility than the Linux kernel's memory leak detector, `kmemleak` [1]. The kernel's detector provides a global view of all memory allocations, which aids in finding repeated leaks but is less suitable for finding small leaks in a specific driver.

Locking Protocols Locking involves the initialization of a lock variable, followed by matched calls to corresponding lock and unlock routines. We have written checkers to verify that calls to lock and unlock functions are never unbalanced, regardless of the execution path taken through a function. SymDrive supports 24 distinct pairs of kernel locking and unlocking functions, as well as a variety of lock-initialization routines.

In order to track the state of the driver's locks, the test framework uses the same object tracker as above. Each time the driver acquires or frees a lock, the checker updates the object tracker with the state of the lock. If the driver attempts to use an incorrect locking function, the test framework prints an error. In addition, SymDrive reports an error if a driver function returns to the kernel without releasing all driver-acquired locks.

Adding additional locking routines to the test framework is straightforward, assuming the lock has most of the same semantics as spinlocks or semaphores. Test #3 in Figure 4 shows an implementation of the test for the `spin_lock_irqsave` function. The `generic_lock_state` function records that the lock is a spin lock and that interrupts were disabled so that the unlock function can verify they restore interrupts.

Driver ioctl Requirements Another source of driver bugs are driver `ioctl` functions, which can cause security vulnerabilities. For example, the function `aac_cfg_ioctl` in the SCSI RAID controller driver `aacraid` did not check that the process executing the `ioctl` has sufficient privileges. SymDrive allows developers to add checkers to guarantee that all drivers of a particular class include the necessary calls to the kernel "capable" function depending on the arguments to the `ioctl`.

The checker, not shown, executes as a postcondition that first checks the class of the driver. If it is a SCSI driver, the checker verifies that if specific `ioctl` commands were provided then the driver invoked `capable`

Test Category	Assertion
Memory leak	The driver frees all allocated SKBs and those passed to the `start_xmit` routine.
Network `probe`	The `probe` functions in network drivers invoke `register_netdev` if they return success.
PCI device state	Calls to `pci_enable_device` (or `pci_enable_device_mem`) and `pci_disable_device` match.
PCI driver state	Calls to `pci_register_driver` and `pci_unregister_driver` match.
USB device state	`usb_enable_device` and `usb_disable_device` match.
Function ordering	PCI network drivers call `pci_disable_device` after calling `unregister_netdev`.
Function ordering	Network drivers finish `probe` successfully before the kernel invokes the `ndo_open` function.
Function ordering	Network driver `ndo_open` functions called only once.
Return values	During the network `ndo_open` call, the driver only calls `netif_start_queue` if open is successful.
Return values	If the network `ndo_open` call is not successful, the driver calls `netif_start_queue`.
Return values	The driver's `ndo_stop` function calls `netif_stop_queue`.
Return values	If `register_netdev` fails, or if the driver does not call it, then the `probe` function returns failure.

Table 1: **Sample of SymDrive's kernel API checkers.**

with the appropriate privilege. Adding a test for another `ioctl` command is easy: simply add the `ioctl` command identifier and a corresponding capability to a table.

Refactoring Refactoring patches reorganize code but should not change the driver's interaction with the hardware. SymDrive can ensure that refactoring does not change how a driver functions by recording all interaction with the hardware along all execution paths. A developer can compare a record of hardware interactions with and without the patch. If no discrepancies are present, the refactoring did not affect the driver/device interface.

SymDrive outputs the sequence of hardware operations (operations that read or write from the device) along all symbolic execution paths using a prefix tree (trie) data structure. Even if a refactoring adds additional branches to the driver, the prefix tree should not change if the branches result in the same device interaction. The developer can then compare the textual representation of these two trees using a `diff` operation. If a discrepancy is apparent, the information provided allows the developer to identify the erroneous hardware operation easily.

This feature is not perfect and is subject to timing issues when the driver executes different functions out of order because the kernel invokes them differently between runs. However, because the prefix tree includes the name of the driver function that initiated each device operation, the developer can simply compare traces of specific functions or ignore parts of the trace not related to the refactored functions.

5 Evaluation

The purpose of the evaluation is to verify that SymDrive achieves its goal: can it verify and validate driver code, and can it do so without the device present? We test whether SymDrive can thoroughly test a variety of drivers, and whether it can find driver bugs unknown to us.

We have tested SymDrive on the drivers shown in Figure 2. These are the devices for which we have hardware, to generate traces, and for which we have annotated the kernel/driver interfaces. These drivers include sound and network drivers on both the PCI and USB buses. We report line counts using the CLOC utility [5]. In addition to executing these drivers symbolically, SymDrive executes a large fraction of the Linux kernel sound driver API symbolically. This sound library contains 30,180 lines of code. All tests took place on a machine running Red Hat Enterprise Linux 5.5 equipped with a quad-core Intel 2.66GHz Q9400 CPU and 8GB of memory.

Driver	Class	Bus	LoC
8139too	Network	PCI	1904
ca0106	Sound	PCI	3052
cmipci	Sound	PCI	2717
e1000	Network	PCI	13973
ens1371	Sound	PCI	2110
pegasus	Network	USB	1541
usb-audio	Sound	USB	8094

Table 2: **We have tested SymDrive with the seven PCI and USB drivers shown here.**

5.1 Invoking the Driver with SymDrive

Using SymDrive by itself does not invoke driver entry points. To test a driver, we carry out the following operations for each driver:

1. Create a virtual hardware device in UML by loading a module with appropriate parameters.
2. Load the driver with `insmod` and wait for initialization to complete.
3. Execute a workload.
4. Unload the driver
5. Allow SymDrive to continue symbolically executing the driver's code for another 10 minutes or less. If SymDrive is unable to cover all possible paths in that time, we abort further execution.

To test specific driver functionality, we execute workloads that trigger the appropriate driver entry point. For example, when the kernel attempts to send packets, it invokes the network driver's `start_xmit` routine. SymDrive itself calls the driver's interrupt handlers. Using `ifconfig` or related `ethtool` commands also invoke

driver entry points. Similarly, the `mpg123` music player exercises a large piece of driver code by attempting to play an MP3 file.

In addition to these tests, we verified that SymDrive achieves 100% code coverage in several complex driver functions. Furthermore, as we developed the checkers in the test framework, we manually implemented test bugs to ensure the checker triggered when the driver violated its specification.

5.2 Bugs Found

We applied SymDrive with the checkers described in the previous section to the seven drivers listed in Table 2. Across these drivers, we found six distinct bugs, one which appeared in two different drivers

1. An allocator/deallocator mismatch in the e1000 driver, found using the object tracker. The test framework reported that the attempt to free the pointer was invalid because the driver allocated it with another function.
2. A memory leak in the ca0106 driver, found using the object tracker. The test framework reported leaked objects after unloading the driver.
3. A missed call to `put_device` on a `device_register` failure path in the sound library, found using the test framework API state machine. The test framework reported that the driver never called `put_device` after unloading the driver.
4. A hardware dependence bug in 8139too, found via an invalid pointer dereference along an unlikely execution path. The test framework itself reported no failure since this check takes place in the SEF.
5. A race condition in ca0106, in which the driver cleared a function pointer used in the interrupt handler before stopping interrupts. The SEF also detected the error.
6. Use of the `GFP_ATOMIC` memory flag in contexts that may block in both the pegasus and usb-audio drivers, which is unnecessary, found via the test framework API execution context state machine.

In addition, SymDrive reported several unusual code fragments, such as calls to `spin_lock_irqsave` followed by `spin_unlock`, `spin_lock`, and finally `spin_unlock_irqrestore`. Although correct, this code deserves additional review. SymDrive also reported a redundant permission check in the `ioctl` function of the E1000 driver.

We verified bugs #1, #2, and #4, and found them fixed in more recent kernel versions. Bugs #3, #5, and #6 we checked manually. The kernel's `device_register` function has a comment specifically reminding developers to call `put_device` if the call fails. We verified the `GFP_ATOMIC` issue in pegasus and usb-audio both by examining the functions in the stack traces and ensuring that the driver could safely block.

To find these bugs, we executed each driver using the workloads described earlier, and enabled all test framework checks. We used the *concrete+symbolic* execution mode in all cases, except for the `device_register` bug. To find that bug, we used the *symbolic only* execution mode for the `device_register` function. The ca0106 driver did not require a hardware trace to discover the memory leak or race condition, thus demonstrating the value of SymDrive even when a trace is not available.

Both the pegasus and usb-audio drivers (bugs listed under #6) are designed such that the driver always submits URBs using the `GFP_ATOMIC` flag even though the `GFP_KERNEL` flag would have been a better choice in some cases. This design stems from reusing the same code on both high and low priority paths, and no mechanism is currently present to distinguish the priority levels, so the driver conservatively chooses `GFP_ATOMIC` under all circumstances.

These bugs clearly demonstrate the diversity of potential issues facing driver developers. Although these bugs do not necessarily result in a driver crash, with the exception of bug #4, they all represent issues that need addressing. Furthermore, they demonstrate the importance of using automated tests, as using the drivers normally would not uncover the bugs.

5.3 Refactoring

To verify that our refactoring checks work correctly, we verify that SymDrive can distinguish between patches that change the driver/device interactions and safe refactorings that do not. We consider two existing drivers, 8139too and pegasus, apply five patches to each driver from the mainline kernel that refactor the code to improve readability or to upgrade the driver to current kernel programming practices. We use SymDrive to execute the original and the patched drivers and record the hardware interactions.

In each case, comparing the prefix tree representations of each driver's interaction with the hardware revealed no changes, because the refactoring is designed not to impact the driver's behavior. Thus, SymDrive confirmed that the refactorings do not affect the driver/device interaction.

To verify that patches to the driver/device interface are detected, we applied a recent patch to 8139too that changes the order of two low-level device operations and to pegasus that also changed its interaction with the hardware. The result in both cases clearly shows that the hardware operations were no longer the same. In the case of 8139too, the result was very clear since the trie showed the order of the two operations had reversed. The result from pegasus, a USB driver, was more difficult to

interpret, because the change occurred in a function several calls removed from the actual device access. Nevertheless, it was clear that the patch impacted the driver's interaction with the hardware. The information provided by the prefix tree in both cases provides enough detail that a developer could find the affected lines of code quickly.

5.4 Driver Setup Time

Testing a driver with SymDrive requires some additional effort beyond the normal kernel build process. After developing the bulk of SymDrive, we added support for the cmipci sound driver. It took less than four hours to run cmipci with SymDrive for the first time, of which three hours were spent adding support to our infrastructure for functionality not used by any of our previous drivers. We expect this effort to decrease as we test more drivers. The remaining time was spent setting up the driver with our build environment (5 minutes), installing the device in a test machine and recording a trace (25 minutes), annotating the driver for DriverSlicer (10 minutes), and finally 15 minutes to actually test the driver. A second developer with access to the trace would need only 15 minutes to build and test the driver symbolically.

5.5 Driver Execution

A final consideration in testing is execution time; fast tests can be run more frequently and find bugs earlier. In order to measure execution time, we timed how long it takes to load and unload each driver. We perform this test both using KLEE and symbolic execution and running the driver as a normal binary to measure the additional overhead of executing a driver with the SEF. We loaded and unloaded each driver three times, and used a trace to minimize the amount of symbolic data used. Refactoring trace support was enabled in all tests. Table 3 shows the results. Native execution is fairly fast, taking under 7 seconds, while symbolic execution can take over seven minutes. These tests times are reasonable when compared to the time it takes to install a driver and reboot a test machine.

We also measured the time it takes to execute a function symbolically, which varies based on how frequently it interacts with the device. As an example of how long it takes to thoroughly evaluate a function heavy on hardware operations, we consider the 8139too function `rtl8139_init_board`. SymDrive took 92 seconds to execute 113 paths through this function, on top of the 10 seconds shown in Figure 3 for the remaining driver load/unload code. Thus, testing even a complex function requires little additional time.

We also examined memory usage. With symbolic execution, memory usage is highly dependent on the number of symbolic variables used and the number of paths executed. After loading the large usb-audio driver, with the trace, Linux reported that the driver process had a resident set size of approximately 194MB. We then modified the trace to exclude the `snd_usb_ctl_msg` function, which the driver uses extensively for hardware communication. After loading and unloading the driver a second time, we let it execute 378 additional execution paths. During this time, the largest memory usage we observed was 243 MB. In our tests, we have never seen the driver process use more than approximately 1GB of memory, since SymDrive focuses execution on specific driver function. However, if the developer employs a large amount of symbolic data in a branch heavy piece of code, the driver process may consume significant system resources.

Driver Load/Unload Times (seconds)

	Native		With KLEE	
Driver	`insmod`	`rmmod`	`insmod`	`rmmod`
8139too	0.3	0.4	4.0	4.5
ca0106	2.5	0.5	56.5	6.9
cmipci	2.2	0.3	30.3	4.4
e1000	6.1	2.1	59.7	368.0
ens1371	3.6	0.5	89.5	14.2
pegasus	0.9	0.3	35.4	6.6
usb-audio	3.8	0.3	188.4	14.9

Table 3: **This table shows how long it takes to load and unload each driver using the trace, in seconds. The second and third columns show the times when SymDrive is compiled without KLEE, and driver code executes natively, while the last two columns show execution time with KLEE.**

5.6 Limitations

SymDrive does report false positives under two circumstances. First, since SymDrive terminates driver execution when executing a symbolic path that returns to the kernel, the object tracker incorrectly reports some driver objects as having not been freed, when in reality, the driver would free them if SymDrive allowed it to continue executing.

SymDrive also reports false positives by default with the various `spin_lock` functions, if the driver does not pair them in a standard way. The example from the sound library, in which `spin_lock_irqsave` was matched with `spin_unlock`, is an example. In this case, SymDrive must make a tradeoff between finding actual bugs, in which the developer accidentally mismatched the locking functions, and false positives, as in the example.

We are not currently aware of any specific false negatives among the checks that SymDrive does support. However, bugs in the test framework itself or failure to consider all the details of a test could easily lead to a false negative.

Finally, SymDrive's use of a trace appears to conflict with SymDrive's goal of identifying chipset-specific bugs. However, running tests with traces from each supported device provides complete coverage.

6 Related work

SymDrive draws on past work in a variety of areas, including symbolic execution, static and dynamic analysis, test frameworks, and formal specification.

Symbolic execution. There are numerous prior approaches to symbolic execution [8, 9, 11, 17, 22, 33, 36, 37]. However, most of these approaches apply to standalone programs with limited environmental interaction. In addition, many of them search for specific kinds of bugs such as crashes and assertion violations, rather than allowing the developer to track the program's logical state as it executes, to verify correctness. To limit symbolic execution to manageable state, previous work limited the set of symbolically executed paths by applying smarter search heuristics and/or by limiting program inputs [10, 18, 22, 23, 24]. SymDrive uses a combination of approaches to handle this issue – with traces to avoid symbolic execution altogether and a combination of BFS and DFS to limit searching.

Most recently, Kuznetsov et al. have applied symbolic execution to finding bugs in binary drivers [22] using the DDT system. DDT's approach has proven excellent at finding a variety of classes of bugs common to all drivers, such as invalid memory accesses and race conditions. Additionally, DDT's ability to fork the kernel state has advantages relative to SymDrive's approach, such as its reduced susceptibility to false positives. In contrast to DDT, SymDrive targets kernel developers and is designed for testing drivers more thoroughly by providing developers with a test specification interface to write new checkers. SymDrive also provides the ability to mix concrete and symbolic execution to fast forward execution to functions of interest, and to verify that a refactored driver is equivalent to its original.

Static and dynamic analysis. Static analysis tools can find specific kinds of bugs common to large classes of drivers, such as misuses of the driver/kernel [6, 7] or driver/device interface [21] and ignored error codes [19, 34]. Similarly, dynamic instrumentation can identify memory leaks and corruption, and observe how the driver responds in low-memory conditions. Microsoft's Driver Verifier (DV) and Static Driver Verifier (SDV) utilities are examples of these dynamic and static approaches [25, 26]. Compared to the symbolic execution used by SymDrive, static bug finding tools are often faster and more scalable, analyzing many drivers at once. However, they often have limited ability to resolve aliases or handle complex pointer arithmetic, both of which are common in drivers. Thus, static and dynamic analysis tools effectively complement tools like SymDrive because of their narrow, but deep, approach to finding specific classes of bugs.

Test frameworks. Test frameworks provide automated testing environments for drivers. IBM maintains the Linux Test Project (LTP) with the aim of providing a set of tests that "validate the reliability, robustness, and stability of Linux," including the kernel and associated drivers [20]. The main drawback to test suites such as the LTP is the need for the device hardware in order to execute the test. In addition, LTP tests execute at the system-call level, and thus cannot verify properties of individual driver entry points.

Formal Specifications for Drivers. Formal specifications also improve reliability by expressing a device's or a driver's operational requirements. Once specified, other parts of the system can verify that the driver is operating correctly. The Nexus operating system [35] uses device safety specifications to guarantee safe driver behavior. Formalizing the device driver's behavior with a statically checked specification is another solution [32]. A similar approach is to use the specification to generate a driver automatically that implements the specification [31]. However, these approaches all require a separate specification for every chip, which is difficult to scale to an entire operating system. In contrast, SymDrive provides fewer guarantees, but its checkers apply to a much broader range of code.

7 Conclusions

SymDrive uses symbolic execution combined with a driver test infrastructure to test driver code without physical access to the corresponding device. Our results show that SymDrive can find subtle bugs in mature driver code, and can distinguish between patches that affect the driver/device interface and those that do not. Thus, SymDrive is a valuable tool that eases the burden of testing drivers.

In the future, we plan to investigate a variety of improvements to SymDrive. First, we are investigating how to extend the test framework with additional checkers. Second, we intend to expand the range of driver classes that SymDrive supports. Currently, SymDrive supports network and sound drivers, but we believe that expanding the test infrastructure to support other classes will be straightforward. Finally, we would like to implement an automated testing service for patches submitted to Linux mailing lists to supplement the code reviews by kernel maintainers.

References

[1] Kernel memory leak detector. Linux kernel source: `Documentation/kmemleak.txt`.

[2] Cve-2006-2936 possible dos in write routine of ftdi_sio driver. https://bugzilla.redhat.com/show_bug.cgi?id=197610, 2006.

[3] Linux driver project. http://www.linuxdriverproject.org/, 2010.

[4] Linux kernel mailing list. https://lkml.org/, 2010.

[5] Al Danial. Cloc: Count lines of code. http://cloc.sourceforge.net/, 2010.

[6] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, et al. Thorough static analysis of device drivers. In *Eurosys '06*, 2006.

[7] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the 29th POPL'02*, 2002.

[8] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proc. of Intl. Conf. on Reliable Software, 1975*, 1975.

[9] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*, 2008.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security*, 2008.

[11] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *HotDep*, 2009.

[12] J. Dike. User-mode linux. http://user-mode-linux.sourceforge.net, June 2005.

[13] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5), 2003.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[15] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN/SIGSOFT PASTE*, June 2001.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS 13*, Mar. 2008.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05*, 2005.

[18] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the NDSS'08*, 2008.

[19] H. Gunawi, C. Rubio-González, A. Arpaci-Dusseau, R. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *6th USENIX FAST*, 2008.

[20] IBM. Linux test project. http://ltp.sourceforge.net/, May 2010.

[21] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP*, Oct. 2009. ACM.

[22] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, June 2010.

[23] E. Larson and T. Austin. High coverage detection of input-related security facults. In *Proceedings of 12th USENIX Security Symposium*. USENIX Association, 2003.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.

[25] Microsoft Corporation. How to use driver verifier to troubleshoot windows drivers. http://support.microsoft.com/kb/q244617/, Jan. 2005. Knowledge Base Article Q244617.

[26] Microsoft Corporation. Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, May 2010.

[27] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *In Proceedings of 1st NSDI*, 2004.

[28] N. Nethercode and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[29] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys'08*, 2008.

[30] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *Eurosys '06*, 2006.

[31] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with termite. In *SOSP '09*, 2009.

[32] L. Ryzhyk, I. Kuz, and G. Heiser. Formalising device driver interfaces. In *Proc.of Workshop on Programming Languages and Systems*, Oct. 2007.

[33] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13*, 2005.

[34] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proceedings of the Sixth European Dependable Computing Conference*, 2006.

[35] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI 8*, Dec. 2008.

[36] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *In TACAS*, 2005.

[37] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proc. of Eurosys '10*, 2010.