SymDrive: Testing Drivers without Devices

Matthew J. Renzelmann, Asim Kadav and Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison
{mjr,kadav,swift}@cs.wisc.edu

Abstract

Device-driver development and testing is a complex and error-prone undertaking. For example, testing error-handling code requires simulating faulty inputs from the device. A single driver may support dozens of devices, and a developer may not have access to any of them. Consequently, many Linux driver patches include the comment "compile tested only."

SymDrive is a system for testing Linux and FreeBSD drivers without their devices present. The system uses symbolic execution to remove the need for hardware, and extends past tools with three new features. First, SymDrive uses static-analysis and source-to-source transformation to greatly reduce the effort of testing a new driver. Second, SymDrive checkers are ordinary C code and execute in the kernel, where they have full access to kernel and driver state. Finally, SymDrive provides an execution-tracing tool to identify how a patch changes I/O to the device and to compare device-driver implementations. In applying SymDrive to 21 Linux drivers and 5 FreeBSD drivers, we found 39 bugs.

1 Introduction

Device drivers are critical to operating-system reliability, yet are difficult to test and debug. They run in kernel mode, which prohibits the use of many runtime program-analysis tools available for user-mode code, such as Valgrind [34]. Their need for hardware can prevent testing altogether: over two dozen driver Linux and FreeBSD patches include the comment "compile tested only," indicating that the developer was unable or unwilling to run the driver. Even with hardware, it is difficult to test error-handling code that runs in response to a device error or malfunction. Thorough testing of failure-handling code is time consuming and requires exhaustive fault-injection tests with a range of faulty inputs.

Complicating matters, a single driver may support dozens of devices with different code paths. For example, one of the 18 supported medium access controllers in the E1000 network driver requires an additional EEPROM read operation while configuring flow-control and link settings. Testing error handling in this driver requires the specific device, and consideration of its specific failure modes.

Static analysis tools such as Coverity [17] and Microsoft's Static Driver Verifier [31] can find many bugs quickly. However, these tools are tuned for fast, relatively shallow analysis of large amounts of code and therefore only approximate the behavior of some code features, such as pointers. Furthermore, they have difficulty with bugs that span multiple invocations of the driver. Hence, static analysis misses large aspects of driver behavior.

We address these challenges using *symbolic execution* to test device drivers. This approach executes driver code on all possible device inputs, allows driver code to execute without the device present, and provides more thorough coverage of driver code, including error handling code. DDT [26] and S^2E [14, 15] previously applied symbolic execution to driver testing, but these systems require substantial developer effort to test new classes of drivers and, in many cases, even specific new drivers.

This paper presents *SymDrive*, a system to test Linux and FreeBSD drivers without devices. SymDrive uses static analysis to identify key features of the driver code, such as entry-point functions and loops. With this analysis, SymDrive produces an instrumented driver with call-outs to test code that allows many drivers to be tested with no modifications. The remaining drivers require a few annotations to assist symbolic execution at locations that SymDrive identifies.

We designed SymDrive for three purposes. First, a driver developer can use SymDrive to test driver patches by thoroughly executing all branches affected by the code changes. Second, a developer can use SymDrive as a debugging tool to compare the behavior of a functioning driver against a non-functioning driver. Third, SymDrive can serve as a general-purpose bug-finding tool and perform broad testing of many drivers with little developer input.

SymDrive is built with the S^2E system by Chipounov et al. [14, 15] , which can make any data within a virtual machine symbolic and explore its effect. SymDrive makes device inputs to the driver symbolic, thereby eliminating the need for the device and allowing execution on the complete range of device inputs. In addition, S^2E enables SymDrive to further enhance code coverage by mak-

ing other inputs to the driver symbolic, such as data from the applications and the kernel. When it detects a failure, either through an invalid operation or an explicit check, SymDrive reports the failure location and inputs that trigger the failure.

SymDrive extends S^2E with three major components. First, SymDrive uses *SymGen*, a static-analysis and code transformation tool, to analyze and instrument driver code before testing. SymGen automatically performs nearly all the tasks previous systems left for developers, such as identifying the driver/kernel interface, and also provides hints to S^2E to speed testing. Consequently, little effort is needed to apply SymDrive to additional drivers, driver classes, or buses. As evidence, we have applied SymDrive to eleven classes of drivers on five buses in two operating systems.

Second, SymDrive provides a *test framework* that allows *checkers* that validate driver behavior to be written as ordinary C code and execute in the kernel. These checkers have access to kernel state and the parameters and results of calls between the driver and the kernel. A checker can make pre- and post-condition assertions over driver behavior, and raise an error if the driver misbehaves. Using bugs and kernel programming requirements culled from code, documentation, and mailing lists, we wrote 49 checkers comprising 564 lines of code to enforce rules that maintainers commonly check during code reviews, such as matched allocation/free calls across entry points, no memory leaks, and proper use of kernel APIs.

Finally, SymDrive provides an *execution-tracing* mechanism for logging the path of driver execution, including the instruction pointer and stack trace of every I/O operation. These traces can be used to compare execution across different driver revisions and implementations. For example, a developer can debug where a buggy driver diverges in behavior from a previous working one. We have also used this facility to compare driver implementations across operating systems.

We demonstrate SymDrive's value by applying it to 26 drivers, and find 39 bugs, including two security vulnerabilities. We also find two driver/device interface violations when comparing Linux and FreeBSD drivers. To the best of our knowledge, no symbolic execution tool has examined as many drivers. In addition, SymDrive achieved over 80% code coverage in most drivers, and is largely limited by the ability of user-mode tests to invoke driver entry points. When we use SymDrive to execute code changed by driver patches, SymDrive achieves over 95% coverage on 12 patches in 3 drivers.

2 Motivation

The goal of our work is to improve driver quality through thorough testing and validation. To be successful, SymDrive must demonstrate (i) usefulness, (ii) simplicity, and (iii) efficiency. First, SymDrive must be able to find bugs that are hard to find using other mechanisms, such as normal testing or static analysis tools. Second, SymDrive must require low developer effort to test a new driver and therefore support many device classes, buses, and operating systems. Finally, SymDrive must be fast enough to apply to every patch.

2.1 Symbolic Execution

SymDrive uses symbolic execution to execute device-driver code without the device being present. Symbolic execution allows a program's input to be replaced with a *symbolic value*, which represents all possible values the data may have. A *symbolic-execution engine* runs the code and tracks which values are symbolic and which have *concrete* (*i.e.*, fully defined) values, such as initialized variables. When the program compares a symbolic value, the engine forks execution into multiple *paths*, one for each outcome of the comparison. It then executes each path with the symbolic value constrained by the chosen outcome of the comparison. For example, the predicate $x > 5$ forks execution by copying the running program. In one copy, the code executes the path where $x \leq 5$ and the other executes the path where $x > 5$. Subsequent comparisons can further constrain a value. In places where specific values are needed, such as printing a value, the engine can concretize data by producing a single value that satisfies all constraints over the data.

Symbolic execution detects bugs either through illegal operations, such as dereferencing a null pointer, or through explicit assertions over behavior, and can show the state of the executing path at the failure site.

Symbolic execution with S^2E. SymDrive is built on a modified version of the S^2E symbolic execution framework. S^2E executes a complete virtual machine as the program under test. Thus, symbolic data can be used anywhere in the operating system, including drivers and applications. S^2E is a virtual machine monitor (VMM) that tracks the use of symbolic data within an executing virtual machine. The VMM tracks each executing path within the VM, and schedules CPU time between paths. Each path is treated like a thread, and the scheduler selects which path to execute and when to switch execution to a different path.

S^2E supports *plug-ins*, which are modules loaded into the VMM that can be invoked to record information or to modify execution. SymDrive uses plugins to implement symbolic hardware, path scheduling, and code-coverage monitoring.

2.2 Why Symbolic Execution?

Symbolic execution is often used to achieve high coverage of code by testing on all possible inputs. For device drivers, symbolic execution provides an additional bene-

fit: executing without the device. Unlike most code, driver code can not be loaded and executed without its device present. Furthermore, it is difficult to force the device to generate specific inputs, which makes it difficult to thoroughly test error handling.

Symbolic execution eliminates the hardware requirement, because it can use symbolic data for all device input. An alternate approach is to code a software model of the device [33], which allows more accurate testing but greatly increases the effort required. In contrast, symbolic execution uses the driver itself as a model of device behavior: any device behavior used by the driver will be exposed as symbolic data.

Symbolic execution may provide inputs that correctly functioning devices may not. However, because hardware can provide unexpected or faulty driver input [25], this unconstrained device behavior is reasonable: drivers should not crash simply because the device provided an errant value.

In comparison to static analysis tools, symbolic execution provides several benefits. First, it uses existing kernel code as a model of kernel behavior rather than requiring a programmer-written model. Second, because driver and kernel code actually execute, it can reuse kernel debugging facilities, such as deadlock detection, and existing test suites. Thus, many bugs can be found without any explicit description of correct driver behavior. Third, symbolic execution can invoke a sequence of driver entry points, which allows it to find bugs that span invocations, such as resource leaks. In contrast, most static analysis tools concentrate on bugs occurring within a single entry point.

2.3 Why not Symbolic Execution?

While symbolic execution has previously been applied to drivers with DDT and S^2E, there remain open problems that preclude its widespread use:

Efficiency. The engine creates a new path for every comparison, and branchy code may create hundreds or thousands of paths, called *path explosion*. This explosion can be reduced by distinguishing and prioritizing paths that complete successfully. This approach enables executing deeper into the driver: if driver initialization fails, the operating system could not otherwise invoke most driver entry points. S^2E and DDT require complex, manually written annotations to provide this information. These annotations depend on kernel function names and behavioral details, which are difficult for programmers to provide. For example, the annotations often examine kernel function parameters, and modify the memory of the current path on the basis of the parameters. The path-scheduling strategies in DDT and S^2E favor exploring new code, but may not execute far enough down a path to test all functionality.

Simplicity. Existing symbolic testing tools require extensive developer effort to test a single class of drivers, plus additional effort to test each individual driver. For example, supporting Windows NDIS drivers in S^2E requires over 1,000 lines of code specific to this driver class. For example, the S^2E wrapper for the `NdisReadConfiguration` Windows function consists of code to (a) read all of the call's parameters, which is not trivial because the code is running outside the kernel, (b) fork additional paths for different possible symbolic return codes, (c) bypass the call to the function along these additional paths, and (d) register a separate wrapper function, of comparable complexity, to execute when this call returns. Since developers need to implement similarly complex code for many other functions in the driver/kernel interface, testing many drivers becomes impractical in these systems. Thus, these tools have only been applied to a few driver classes and drivers. Expanding testing to many more drivers requires new techniques to automate the testing effort.

Specification. Finally, symbolic execution by itself does not provide any specification of correct behavior: a "hello world" driver does nothing wrong, nor does it do anything right, such as registering a device with the kernel. In existing tools, tests must be coded like debugger extensions, with calls to read and write remote addresses, rather than as normal test code. Allowing developers to write tests in the familiar kernel environment simplifies the specification of correct behavior.

Thus, our work focuses on improving the state of the art to greatly simplify the use of symbolic execution for testing, and to broaden its applicability to almost any driver in any class on any bus.

3 Design

The SymDrive architecture focuses on thorough testing of drivers to ensure the code does not incorrectly use the kernel/driver interface, crash, or hang. We target test situations where the driver code is available, and use that code to simplify testing with a combination of symbolic execution, static code analysis and transformation, and an extensible test frameworkexecuting in the kernel.

The design of SymDrive is shown in Figure 1. The OS kernel and driver under test, as well as user-mode test programs, execute in a virtual machine. The symbolic execution engine provides symbolic devices for the driver. SymDrive provides stubs that invoke checkers on every call into or out of the driver. A test framework tracks execution state and passes information to plugins running in the engine to speed testing and improve test coverage.

During the development of SymDrive, we considered a more limited design in which symbolic execution was limited to driver code. In this model, exploring multiple paths through the kernel was not possible; callbacks to

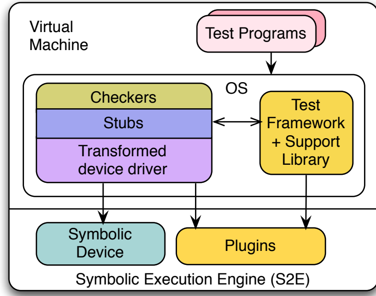

Figure 1: **The SymDrive architecture. A developer produces the transformed driver with SymGen and can write checkers and test programs to verify correctness.**

Component	LoC
Changes to S^{2}E	1,954
SymGen	2,681
Test framework	3,002
Checkers	564
Support Library	1,579
Linux kernel changes	153
FreeBSD kernel changes	81

Table 1: **Implementation size of SymDrive.**

the kernel instead required a model of kernel behavior to allow them to execute on multiple branches. After implementing a prototype of this design, we concluded that full-system symbolic execution is preferable because it greatly reduces the effort to test drivers by using real kernel code rather than a kernel model.

We implemented SymDrive for Linux and FreeBSD, as these kernels provide a large number of drivers to test. Only the test framework code running in the kernel is specialized to the OS. We made small, conditionally compiled changes to both kernels to print failures and stack traces to the S^{2}E log and to register the module under test with S^{2}E. The breakdown of SymDrive code is shown in Table 1.

SymDrive consists of five components: (i) a modified version of the S^{2}E symbolic-execution engine, which consists of a SymDrive-specific plugin plus changes to S^{2}E; (ii) symbolic devices to provide symbolic hardware input to the driver; (iii) a *test framework* executing within the kernel that guides symbolic execution; (iv) the *SymGen* static-analysis and code transformation tool to analyze and prepare drivers for testing; and (v) a set of OS-specific *checkers* that interpose on the driver/kernel interface for verifying and validating driver behavior.

3.1 Virtual Machine

SymDrive uses S^{2}E [15] version 1.1-10.09.2011, itself based on QEMU [7] and KLEE [10], for symbolic execution. S^{2}E provides the execution environment, path forking, and constraint solving capability necessary for symbolic execution. All driver and kernel code, including the test framework, executes within an S^{2}E VM. Changes to S^{2}E fall into two categories: (i) improved support for symbolic hardware, and (ii) the SymDrive path-selection mechanism, which is an S^{2}E plugin.

SymDrive uses invalid x86 opcodes for communication with the VMM and S^{2}E plugins to provide additional control over the executing code. We augment S^{2}E with new opcodes for the test framework that pass information into our extensions. These opcodes are inserted into driver code by SymGen and also invoked directly by the test framework.

The purpose of the opcodes is to communicate source-level information to the SymDrive plugins, which uses the information to guide the driver's execution. The opcodes (i) control whether memory regions are symbolic, as when mapping data for DMA; (ii) influence path scheduling by adjusting priority, search strategy, or killing other paths; and (iii) support tracing by turning it on/off and providing stack information.

3.2 Symbolic Devices

Drivers interact with devices according to well-defined, narrow interfaces. For PCI device drivers, this interface is comprised of I/O memory accessed via normal loads and stores, port I/O instructions, bus operations, DMA memory, and interrupts. Drivers using other buses, such as SPI and I^{2}C, use functions provided by the bus for similar operations.

SymDrive provides a symbolic device for the driver under test, while at the same time emulating the other devices in the system. A symbolic device provides three key behaviors. First, it can be *discovered*, so the kernel loads the appropriate driver. Second, it provides methods to read from and write to the device and return symbolic data from reads. Third, it supports interrupts and DMA, if needed. SymDrive currently supports 5 buses on Linux: PCI (and its variants), I^{2}C (including SMBus), Serial Peripheral Interface (SPI), General Purpose I/O (GPIO), and Platform;[1] and the PCI bus on FreeBSD.

Device Discovery. When possible, SymDrive creates symbolic devices in the S^{2}E virtual machine and lets existing bus code discover the new device and load the appropriate driver. For some buses, such as I^{2}C, the kernel or another driver normally creates a statically configured device object during initialization. For such devices, we created a small kernel module, consisting of 715 lines of code, that creates the desired symbolic device.

SymDrive can make the device configuration space symbolic after loading the driver by returning symbolic data from PCI bus functions with the test frame-

[1] The "platform bus" is a Linux-specific term that encompasses many embedded devices. Linux's ARM implementation, for example, supports a variety of SoCs, each with its own set of integrated devices. The drivers for these devices are often of the "platform" type.

work(similar to S^2E's NDIS driver support). PCI devices use this region of I/O memory for plug-and-play information, such as the vendor and device identifiers. If this data is symbolic, the device ID will be symbolic and cause the driver to execute different paths for each of its supported devices. Other buses have similar configuration data, such as "platform data" on the SPI bus. A developer can copy this data from the kernel source and provide it when creating the device object, or make it symbolic for additional code coverage.

Symbolic I/O. Most Linux and FreeBSD drivers do a mix of programmed I/O and DMA. SymDrive supports two forms of programmed I/O. For drivers that perform I/O through hardware instructions, such as `inb`, or through memory-mapped I/O, SymDrive directs S^2E to ignore write operations, because they do not return values that influence driver execution, and to return symbolic data from reads. The test framework overrides bus I/O functions, such as those used in I^2C drivers, to function analogously.

Symbolic Interrupts. After a driver registers an interrupt handler, the test framework invokes the interrupt handler on every transition from the driver into the kernel. This model represents a trade-off between realism and simplicity: it ensures the interrupt handler is called often enough to keep the driver executing successfully, but may generate spurious interrupts when the driver does not expect them.

Symbolic DMA. When a driver invokes a DMA mapping function, such as `dma_alloc_coherent`, the test framework uses a new S^2E opcode to make the memory act like a memory-mapped I/O region: each read returns a new symbolic value and writes have no effect. Discarding writes to DMA memory reflects the ability of the device to write the data via DMA at any time. The driver should not assume that data written here will be available subsequently. When the driver unmaps the memory, the test framework directs S^2E to revert the region to normal symbolic data, so writes are seen by subsequent reads.

3.3 Test Framework

The test framework is a kernel module executing with the virtual machine that assists symbolic execution and executes checkers. SymDrive relies on the test framework to guide and monitor symbolic execution in three ways. First, the test framework implements policy regarding which paths to prioritize or deprioritize. Second, the test framework may inject additional symbolic data to increase code coverage. As mentioned above, it implements symbolic I/O interfaces for some device classes. Finally, it provides the VMM with a stack trace for *execution tracing*, which produces a trace of the driver's I/O operations.

The test framework supports several load-time parameters for controlling its behavior. When loading the test framework with `insmod` or FreeBSD's `kldload`, developers can direct the test framework to enable high-coverage mode (described in Section 3.3.1), tracing, or a specific symbolic device. To configure the device, developers pass the device's I/O capabilities and name as parameters. Thus, developers can script the creation of symbolic devices to automate testing.

SymDrive has to address two conflicting goals in testing drivers: (i) executing as far as possible along a path to complete initialization and expose the rest of the driver's functionality; and (ii) executing as much code as possible within each function for thoroughness.

3.3.1 Reaching Deeply

A key challenge in fully testing drivers is symbolically executing branch-heavy code, such as loops and initialization code that probes hardware. SymDrive relies on two techniques to limit path explosion in these cases: *favor-success scheduling* and *loop elision*. These techniques allow developers to execute further into a driver, and test functionality that is only available after initialization.

Favor-success scheduling. Executing past driver initialization is difficult because the code often has many conditionals to support multiple chips and configurations. Initializing a sound driver, for example, may execute more than 1,000 branches on hardware-specific details. Each branch creates additional paths to explore.

SymDrive mitigates this problem with a *favor-success* path-selection algorithm that prioritizes successfully executing paths, making it a form of best-first search. Notifications from the test framework increase the priority of the current path at every successful function return, both within the driver and at the driver/kernel interface. Higher priority causes the current path to be explored further before switching to another path. This strategy works best for small functions, where a successful path through the function is short.

At every function exit, the test framework notifies S^2E of whether the function completed successfully, which enables the VMM to prioritize successful paths to facilitate deeper exploration of code. The test framework determines success based on the function's return value. For functions returning integers, the test framework detects success when the function returns a value other than an `errno`, which are standard Linux and FreeBSD error values. On success, the test framework will notify the VMM to prioritize the current path. If the developer wishes to prioritize paths using another heuristic, he/she can add an annotation prioritizing any piece of code. We use this approach in some network drivers to select paths where the carrier is on, which enables execution of the driver's packet transmission code.

In order to focus symbolic execution on the driver, the

test framework prunes paths when control returns to the kernel successfully. It kills all other paths still executing in the driver and uses an opcode to concretize all data in the virtual machine, so the kernel executes on real values and will not fork new paths. This ensures that a single path runs in the kernel and allows developers to interact with the system and run user-mode tests.

Loop elision. Loops are challenging for symbolic execution because each iteration may fork new paths. S^2E provides an "EdgeKiller" plugin that a developer may use to terminate complex loops early, but requires developers to identify each loop's offset in the driver binary [15] and hence incur substantial developer effort.

SymDrive addresses loops explicitly by prioritizing paths that exit the loop quickly. Suppose an execution path *A* enters a loop, executes it once, and during this iteration more paths are created. If path *A* does not exit the loop after one iteration, SymDrive executes it for a second iteration and, unless it breaks out early, deprioritizes the second iteration because it appears stuck in the loop. SymDrive then selects some other path *B* that path *A* forked, and executes it. SymDrive repeats this process until it finds a path that exits the loop. If no paths exit the loop promptly, SymDrive selects some path arbitrarily and prioritizes it on each subsequent iteration, in the hope that it will exit the loop eventually. If this path still does not exit the loop within 20 iterations, SymDrive prints a warning about excessive path forking as there is no evident way to execute the loop efficiently without manual annotation.

This approach executes hardware polling loops efficiently and automatically, and warns developers when loops cause performance problems. However, this approach may fail if a loop is present in uninstrumented kernel code. It can also result in worse coverage of code that executes only if a polling loop times out. Moreover, loops that produce a value, such as a checksum calculation, cannot exit early without stopping the driver's progress. However, we have not found these problems to be significant.

SymDrive's approach extends the EdgeKiller plugin in two directions. First, it allows developers to annotate driver source rather than having to parse compiled code. Second, source annotations persist across driver revisions, whereas the binary offsets used in the EdgeKiller plugin need updating each time the driver changes.

Annotating code manually to improve its execution performance does reduce SymDrive's ability to find bugs in that code. Wherever annotations were needed in the drivers we examined, we strove to write them in such a way as to execute the problematic loop at least once before terminating early. For example, after a checksum loop, we would add a line to return a symbolic checksum value, which could then be compared against a correct one.

3.3.2 Increasing Coverage

SymDrive provides a *high-coverage* mode for testing specific functions, for example those modified by a patch. This mode changes the path-prioritization policy and the behavior of kernel functions. When the developer loads the test framework module, he/she can specify any driver function to execute in this mode.

When execution enters the specified function, the test framework notifies S^2E to favor unexecuted code (the default S^2E policy) rather than favoring successful paths. The test framework terminates all paths that return to the kernel in order to focus execution within the driver. In addition, when the driver invokes a kernel function, the test framework makes the return value symbolic. This mode is similar to the local consistency mode in S^2E [15], but requires no developer-provided annotations or plugins, and supports all kernel functions that return standard error values. For example, `kmalloc` returns a symbolic value constrained to be either NULL or a valid address, which tests error handling in the driver.

For the small number of kernel functions that return non-standard values, SymGen has a list of exceptions and how to treat their return values. The full list of exceptions for Linux currently contains 100 functions across all supported drivers. Of these, 64 are hardware-access functions, such as `inb` and `readl`, that always return symbolic data. A further 14 are arithmetic operations, such as `div32`. The remaining 22 functions return negative numbers in successful cases, or are used by the compiler to trigger a compilation failure when used incorrectly, such as `__bad_percpu_size`.

SymDrive also improves code coverage by introducing additional symbolic data in order to execute code that requires specific inputs from the kernel or applications. SymDrive can automatically make a Linux driver's module parameters symbolic, executes the driver with all possible parameters. Checkers can also make parameters to the driver symbolic, such as `ioctl` command values. This allows all `ioctl` code to be tested with a single invocation of the driver, because each comparison of the command will fork execution. In addition, S^2E allows using symbolic data anywhere in the virtual machine, so a user-mode test can pass symbolic data to the driver.

3.3.3 Execution Tracing

The test framework can generate execution traces, which are helpful to compare the execution of two versions of a driver. For example, when a driver patch introduces new bugs, the traces can be used to compare its behavior against previous versions. In addition, developers can use other implementations of the driver, even from another operating system, to find discrepancies that may signify incorrect interaction with the hardware.

A developer can enable tracing via a command-line tool

that uses a custom opcode to notify SymDrive to begin recording. In this mode, an S^{2}E plugin records every driver I/O operation, including reads and writes to port, MMIO, and DMA memory, and the driver stack at the operation. The test framework passes the current stack to S^{2}E on every function call.

The traces are stored as a trie (prefix tree) to represent multiple paths through the code compactly, and can be compared using the `diff` utility. SymDrive annotates each trace entry with the driver call stack at the I/O operation. This facilitates analysis of specific functions and comparing drivers function-by-function, which is useful since traces are subject to timing variations and different thread interleavings.

3.4 SymGen

All features of the test framework that interact with code, such as favor-success scheduling, loop prioritization, and making kernel return values symbolic are handled automatically via static analysis and code generation. The SymGen tool analyzes driver code to identify code relevant to testing, such as function boundaries and loops, and instruments code with calls to the test framework and checkers. SymGen is built using CIL [32].

Stubs. SymDrive interposes on all calls into and out of the driver with stubs that call the test framework and checkers. For each function in the driver, SymGen generates two stubs: a preamble, invoked at the top of the function, and a postscript, invoked at the end. The generated code passes the function's parameters and return value to these stubs to be used by checkers. For each kernel function the driver imports, SymGen generates a stub function with the same signature that wraps the function.

To support pre- and post-condition assertions, stubs invoke checkers when the kernel calls into the driver or the driver calls into the kernel. Checkers associated with a specific function `function_x` are named `function_x_check`. On the first execution of a stub, the test framework looks for a corresponding checker in the kernel symbol table. If such a function exists, the stub records its address for future invocations. While targeted at functions in the kernel interface, this mechanism can invoke checkers for any driver function.

Stubs employ a second lookup to find checkers associated with a function pointer passed from the driver to the kernel, such as a PCI probe function. Kernel stubs, when passed a function pointer, record the function pointer and its purpose in a table. For example, the Linux `pci_register_driver` function associates the address of each function in the `pci_driver` parameter with the name of the structure and the field containing the function. The stub for the `probe` method of a `pci_driver` structure is thus named `pci_driver_probe_check`. FreeBSD drivers use a similar technique.

```
s2e_loop_before(__LINE__, loop_id);
while(work--) {
  tmp___17 = readb(cp->regs + 55);
  if(!(tmp___17 & 16)) goto return_label;
  stub_schedule_timeout_uninterruptible(10L);
  s2e_loop_body(__LINE__, loop_id);
}
s2e_loop_after(__LINE__, loop_id);
```

Figure 2: **SymGen instruments the start, end, and body of loops automatically. This code, from the `8139cp` driver, was modified slightly since SymGen produces preprocessed output.**

Stubs detect that execution enters the driver by tracking the depth of the call stack. The first function in the driver notifies the test framework at its entry that driver execution is starting, and at its exit notifies the test framework that control is returning to the kernel. Stubs also communicate this information to the VMM so that it can make path-scheduling decisions based on function return values.

Instrumentation. The underlying principle behind SymGen's instrumentation is to inform the VMM of source level information as it executes the driver so it can make better decisions about which paths to execute. SymGen instruments the start and end of each driver function with a call into the stubs. As part of the rewriting, it converts functions to have a single exit point. It generates the same instrumentation for inline functions, which are commonly used in the Linux and FreeBSD kernel/driver interfaces.

SymGen also instruments the start, end, and body of each loop with calls to short functions that execute SymDrive-specific opcodes. These opcodes direct the VMM to prioritize and deprioritize paths depending on whether they exit the loop quickly. This instrumentation replaces most of the per-driver effort required by S^{2}E to identify loops, as well as the per-class effort of writing a consistency model for every function in the driver/kernel interface. SymGen also inserts loop opcodes into the driver, as shown in Figure 2, to tell S^{2}E which paths exit the loop, and should receive a priority boost.[2]

For complex code that slows testing, SymGen supports programmer-supplied annotations to simplify or disable the code temporarily. Short loops and those that do not generate states require no manual developer effort. Only loops that must execute for many iterations and generate new paths on each iteration need manual annotation, which we implement through C `#ifdef` statements. For example, the E1000 network driver verifies a checksum over EEPROM, and we modified it to accept any checksum value. We have found these cases to be rare.

[2]One interesting alternative is to prioritize paths that execute loops in their entirety. The problem with this approach is that it may generate many states in the process, and slow testing.

3.5 Limitations

SymDrive is neither sound nor complete. The false positives we have experienced fall into two major categories. First, symbolic execution is slow, which may cause the kernel to print timing warnings and cause driver timers to fire at the wrong time. Second, our initial checkers were imprecise and disallowed (bizarre) behavior the kernel considers legal. We have since fixed the checkers, and have not seen them generate false positives.

Although we have observed no false negatives among the checkers we wrote, SymDrive cannot check for all kinds of bugs. Of 11 common security vulnerabilities [12], SymDrive cannot detect integer overflows and data races between threads, though support for overflow detection is possible in principle because the underlying VMM interprets code rather than executing it directly. In addition, SymDrive cannot achieve full path coverage for all drivers because SymDrive's aggressive path pruning may terminate paths that lead to bugs. SymDrive may also miss race conditions, such as those requiring the interrupt handler to interleave with another thread in a specific way.

4 Checkers

SymDrive detects driver/kernel interface violations with checkers, which are functions interposing on control transfer between the driver and kernel that verify and validate driver behavior. Each function in the driver/kernel interface can, but need not, have its own checker. Drivers invoke the checkers from stubs, described above, which call separate checkers at every function in the driver/kernel interface. Since checkers run in the VM alongside the symbolically executing driver, they can verify runtime properties along each tested path.

The checkers use a *support library* that simplifies their development by providing much of their functionality. The library provides state variables to track the state of the driver and current thread, such as whether it has registered itself successfully and whether it can be rescheduled. The library also provides an object tracker to record kernel objects currently in use in the driver. This object tracker provides an easy mechanism to track whether locks have been initialized and to discover memory leaks. Finally, the library provides generic checkers for common classes of kernel objects, such as locks and allocators. The generic checkers encode the semantics of these objects, and thus do much of the work. For example, checkers for a mutex lock and a spin lock use the same generic checker, as they share semantics.

Writing a checker requires implementing checks within a call-out function. We have implemented 49 checkers comprising 564 lines of code for a variety of common device-driver bugs using the library API. Test #1 in Figure 3 shows an example call-out for `pci_register_driver`. The driver-function stub invokes the checker function with the parameters and return value of the kernel function and sets a `precondition` flag to indicate whether the checker was called before or after the function. In addition, the library provides the global `state` variable that a checker can use to record information about the driver's activity. 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. Checkers have access to the runtime state of the driver and can store arbitrary data, so they can find interprocedural, pointer-specific bugs that span multiple driver invocations.

```
/* Test #1 */ void __pci_register_driver_check(...) {
  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);
  }
}

/* Test #2 */ void __kmalloc_check
  (..., 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);
}

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

Figure 3: **Example checkers. The first checker ensures that PCI drivers are registered exactly once. The second verifies that a driver allocates memory with the appropriate `mem_flags` parameter. The third ensures lock/unlock functions are properly matched. Unlike Static Driver Verifier checkers [31], these checkers can track any path-specific run-time state expressible in C.**

Not every behavior requirement needs a checker. Symbolic execution leverages the extensive checks already included as kernel debug options, such as for memory corruption and locking. Most of these checks execute within functions called *from* the driver, and thus will be invoked on multiple paths. In addition, any bug that causes a kernel crash or panic will be detected by the operating system and therefore requires no checker.

We next describe a few of the 49 checkers we have implemented with SymDrive.

Execution Context. Linux prohibits the calling of functions that block when executing in an interrupt handler or while holding a spinlock. 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. The support library provides a state machine to track the driver's current context using a stack. When entering the driver, the library updates the context based on the entry point. The library also supports locks and interrupt management. When the driver acquires or releases a spinlock, for example, the library pushes or pops the necessary context.

Kernel API Misuse. The kernel requires that drivers follow the proper protocol for kernel APIs, and errors can lead to a non-functioning driver or a resource leak. The support library state variables provide 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. Test #1 in Figure 3 shows a use of these states to ensure that a driver only invokes `pci_register_driver` once.

Collateral Evolutions. Collateral evolutions occur when a small change to a kernel interface necessitates changes in many drivers simultaneously. A developer can use SymDrive to verify that collateral evolutions [35] are correctly applied by ensuring that patched drivers do not regress on any tests.

SymDrive can also ensure that the desired *effect* of a patch is reflected in the driver's execution. For example, recent kernels no longer require that network drivers update the `net_device->trans_start` variable in their `start_xmit` functions. We wrote a checker to verify that `trans_start` is constant across `start_xmit` calls.

Memory Leaks. The leak checker uses the support library's object tracker to store an allocation's address and length. We implemented checkers to verify allocation and free requests from 19 pairs of functions, and ensure that an object's allocation and freeing use paired routines.

The API library simplifies writing checkers for additional allocators down to a few lines of code. Test #2 in Figure 3 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.

5 Evaluation

The purpose of the evaluation is to verify that SymDrive achieves its goals: (i) usefulness, (ii) simplicity, and (iii) efficiency.

5.1 Methodology

As shown in Table 2, we tested SymDrive on 26 drivers in 11 classes from several Linux kernel revisions (13 drivers from 2.6.29, 4 from 3.1.1, and 4 that normally run only on Android-based phones) and from FreeBSD 9 (5 drivers). Of the 26 drivers, we chose 19 as examples of a specific bus or class and the remaining 7 because we found frequent patches to them and thus expected to find bugs.

All tests took place on a machine running Ubuntu 10.10 x64 equipped with a quad-core Intel 2.50GHz Intel Q9300 CPU and 8GB of memory. All results are obtained while running SymDrive in a single-threaded mode, as SymDrive does not presently work with S^{2}E's multicore support.[3]

To test each driver, we carry out the following operations:

1. Run SymGen over the driver and compile the output.
2. Define a virtual hardware device with the desired parameters and boot the SymDrive virtual machine.
3. Load the driver with `insmod` and wait for initialization to complete successfully. Completing this step entails executing at least one successful path and returning success, though it is likely that other failed paths also run and are subsequently discarded.
4. Execute a workload (optional). We ensure all network drivers attempt to transmit and that sound drivers attempt to play a sound.
5. Unload the driver.

If SymDrive reports warnings about too many paths from complex loops, we annotate the driver code and repeat the operations. For most drivers, we run SymGen over only the driver code. For drivers that have fine-grained interactions with a library, such as sound drivers and the `pluto2` media driver, we run SymGen over both the library and the driver code. We annotated each driver at locations SymDrive specified, and tested each Linux driver with 49 checkers for a variety of common bugs. For FreeBSD drivers, we only used the operating system's built-in test functionality.

5.2 Bug Finding

Across the 26 drivers listed in Table 2, we found the 39 distinct bugs described in Table 3. Of these bugs, S^{2}E detected 17 via a kernel warning or crash, and the checkers caught the remaining 22. Although these bugs do not necessarily result in driver crashes, they all represent issues that need addressing and are difficult to find without visibility into driver/kernel interactions.

These results show the value of symbolic execution. Of the 39 bugs, 56% spanned multiple driver invocations. For example, the `akm8975` compass driver calls `request_irq` before it is ready to service interrupts. If an interrupt occurs immediately after this call, the driver will crash, since the interrupt handler dereferences a pointer

[3] This limitation is an engineering issue and prevents SymDrive from exploring multiple paths simultaneously. However, because SymDrive's favor-success scheduling often explores a single path deeply rather than many paths at once, S^{2}E's multi-threaded mode would have little performance benefit.

Driver	Class	Bugs	LoC	Ann	Load	Unld.
*akm8975**	Compass	4	629	0	0:22	0:08
*mmc31xx**	Compass	3	398	0	0:10	0:04
tle62x0*	Control	2	260	0	0:06	0:05
me4000	Data Ac.	1	5,394	2	1:17	1:04
phantom	Haptic	0	436	0	0:16	0:13
lp5523*	LED Ctl.	2	828	0	2:26	0:19
apds9802*	Light	0	256	1	0:31	0:21
8139cp	Net	0	1,610	1	1:51	0:37
8139too	Net	2	1,904	3	3:28	0:35
be2net	Net	7	3,352	2	4:49	1:39
dl2k	Net	1	1,985	5	2:52	0:35
e1000	Net	3	13,971	2	4:29	2:01
et131x	Net	2	8,122	7	6:14	0:47
forcedeth	Net	1	5,064	2	4:28	0:51
ks8851*	Net	3	1,229	0	2:05	0:13
pcnet32	Net	1	2,342	1	2:34	0:27
*smc91x**	Net	0	2,256	0	10:41	0:22
pluto2	Media	2	591	3	1:45	1:01
econet	Proto.	2	818	0	0:11	0:11
ens1371	Sound	0	2,112	5	27:07	4:48
*a1026**	Voice	1	1,116	1	0:34	0:03
ed	Net	0	5,014	0	0:49	0:13
re	Net	0	3,440	3	16:11	0:21
rl	Net	0	2,152	1	2:00	0:08
es137x	Sound	1	1,688	2	57:30	0:09
maestro	Sound	1	1,789	2	17:51	0:27

Table 2: **Drivers tested. Those in *italics* run on Android-based phones, those followed by an asterisk are for embedded systems and do not use the PCI bus. Drivers above the line are for Linux and below the line are for FreeBSD. Line counts come from CLOC [1]. Times are in minute:second format, and are an average of three runs.**

Bug Type	Bugs	Kernel / Checker	Cross EntPt	Paths	Ptrs
Hardware Dep.	7	6 / 1	4	6	6
API Misuse	15	7 / 8	6	5	1
Race	3	3 / 0	3	2	3
Alloc. Mismatch	3	0 / 3	3	0	3
Leak	7	0 / 7	6	1	7
Driver Interface	3	0 / 3	0	2	0
Bad pointer	1	1 / 0	0	0	1
Totals	39	17 / 22	22	16	21

Table 3: **Summary of bugs found. For each category, we present the number of bugs found by kernel crash/warning or a checker and the number that crossed driver entry points ("Cross EntPt"), occurred only on specific paths, or required tracking pointer usage.**

that is not yet initialized. In addition, 41% of the bugs occurred on a unique path through a driver other than one that returns success, and 54% involved pointers and pointer properties that may be difficult to detect statically.

Bug Validation. Of the 39 bugs found, at least 17 were fixed between the 2.6.29 and 3.1.1 kernels, which indicates they were significant enough to be addressed. We were unable to establish the current status of 7 others because of significant driver changes. We have submitted bug reports for 5 unfixed bugs in mainline Linux drivers, all of which have been confirmed as genuine by kernel developers. The remaining bugs are from drivers outside the main Linux kernel that we have not yet reported.

5.3 Developer Effort

One of the goals of SymDrive is to minimize the effort to test a driver. The effort of testing comes from three sources: (i) annotations to prepare the driver for testing, (ii) testing time, and (iii) updating code as kernel interfaces change.

To measure the effort of applying SymDrive to a new driver, we tested the `phantom` haptic driver from scratch. The total time to complete testing was 1h:45m, despite having no prior experience with the driver and not having the hardware. In this time, we configured the symbolic hardware, wrote a user-mode test program that passes symbolic data to the driver's entry points, and executed the driver four times in different configurations. Of this time, the overhead of SymDrive compared to testing with a real device was an additional pass during compilation to run SymGen, which takes less than a minute, and 38 minutes to execute. Although not a large driver, this test demonstrates SymDrive's usability from the developer's perspective.

Annotations. The only per-driver coding SymDrive requires is annotations on loops that slow testing and annotations that prioritize specific paths. Table 2 lists the number of annotation sites for each driver. Of the 26 drivers, only 6 required more than two annotations, and 9 required no annotations. In all cases, SymDrive printed a warning indicating where an annotation would benefit testing.

Testing time. Symbolic execution can be much slower than normal execution. Hence, we expect it to be used near the end of development, before submitting a patch, or on periodic scans through driver code. We report the time to load, initialize, and unload a driver (needed for detecting resource leaks) in Table 2. Initialization time is the minimum time for testing, and thus presents a lower bound.

Overall, the time to initialize a driver is roughly proportional to the size of the driver. Most drivers initialize in 5 minutes or less, although the `ens1371` sound driver required 27 minutes, and the corresponding FreeBSD `es137x` driver required 58 minutes. These two results stem from the large amount of device interaction these drivers perform during initialization. Excluding these results, execution is fast enough to be performed for every patch, and with a cluster could be performed on every driver affected by a collateral evolution [35].

Kernel evolution. Near the end of development, we upgraded SymDrive from Linux 2.6.29 to Linux 3.1.1. If much of the code in SymDrive was specific to the kernel interface, porting SymDrive would be a large effort. However, SymDrive's use of static analysis and code generation minimized the effort to maintain tests as the kernel evolves: the only changes needed were to update a

Driver	Touched Funcs.	Coverage	Time CPU	Time Latency
8139too	93%	83%	2h36m	1h00m
a1026	95%	80%	15m	13m
apds9802	85%	90%	14m	7m
econet	51%	61%	42m	26m
ens1371	74%	60%	*8h23m	*2h16m
lp5523	95%	83%	21m	5m
me4000	82%	68%	*26h57m	*10h25m
mmc31xx	100%	83%	14m	26m
phantom	86%	84%	38m	32m
pluto2	78%	90%	19m	6m
tle62x0	100%	85%	16m	12m
es137x	97%	70%	1h22m	58m
rl	84%	71%	13m	10m

Table 4: **Code coverage.**

few checkers whose corresponding kernel functions had changed. The remainder of the system, including SymGen and the test framework, were unchanged. The number of lines of code changed was less than 100.

Furthermore, porting SymDrive to a new operating system is not difficult. We also ported the SymDrive infrastructure, checkers excluded, to FreeBSD 9. The entire process took three person-weeks. The FreeBSD implementation largely shares the same code base as the Linux version, with just a few OS-specific sections. This result confirms that the techniques SymDrive uses are compatible across operating systems.

5.4 Coverage

While SymDrive primarily uses symbolic execution to simulate the device, a second benefit is higher code coverage than standard testing. Table 4 shows coverage results for one driver of each class, and gives the fraction of functions executed ("Touched Funcs.") and the fraction of basic blocks *within* those functions ("Coverage").[4] In addition, the table gives the total CPU time to run the tests on a single machine (CPU) and the latency of the longest run if multiple machines are used (Latency). In all cases, we ran drivers multiple times and merged the coverage results. We terminated each run once it reached a steady state and stopped testing the driver once coverage did not meaningfully improve between runs.

Overall, SymDrive executed a large majority (80%) of driver functions in most drivers, and had high coverage (80% of basic blocks) in those functions. These results are below 100% for two reasons. First, we could not invoke all entry points in some drivers. For example, `econet` requires user-mode software to trigger additional driver entry points that SymDrive is unable to call on its own. In other cases, we simply did not spend enough time understanding how to invoke all of a driver's code, as some functionality requires the driver to be in a specific state that is difficult to realize, even with symbolic execution.

[4] * Drivers with an asterisk ran unattended, and their total execution time is not representative of the minimum.

Driver	Touched Funcs.	Coverage	Time Serial	Time Parallel
8139too	100%	96%	9m	5m
ks8851	100%	100%	16m	8m
lp5523	100%	97%	12m	12m

Table 5: **Patched code coverage.**

Second, of the functions SymDrive did execute, additional inputs or symbolic data from the kernel were needed to test all paths. Following S^2E's relaxed consistency model by making more of the kernel API symbolic could help improve coverage.

As a comparison, we tested the `8139too` driver on a real network card using `gcov` to measure coverage with the same set of tests. We loaded and unloaded the driver, and ensured that transmit, receive, and all ethtool functions executed. Overall, these tests executed 77% of driver functions, and covered 75% of the lines in the functions that were touched, as compared to 93% of functions and 83% of code for SymDrive. Although not directly comparable to the other coverage results due to differing methodologies, this result shows that SymDrive can provide coverage better than running the driver on real hardware.

5.5 Patch Testing

The second major use of SymDrive is to verify driver patches similar to a code reviewer. For this use, we seek high coverage in every function modified by the patch in addition to the testing described previously. We evaluate SymDrive's support for patch testing by applying all the patches between the 3.1.1 and 3.4-rc6 kernel releases that applied to the `8139too` (net), `ks8851` (net) and `lp5523` (LED controller) drivers, of which there were 4, 2, and 6, respectively. The other drivers lacked recent patches, had only trivial patches, or required upgrading the kernel, so we did not consider them.

In order to test the functions affected by a patch, we used favor-success scheduling to fast-forward execution to a patched function and then enabled high coverage mode. The results, shown in Table 5, demonstrate that SymDrive is able to quickly test patches as they are applied to the kernel, by allowing developers to test nearly all the changed code without any device hardware. SymDrive was able to execute 100% of the functions touched by all 12 patches across the 3 drivers, and an average of 98% of the code in each function touched by the patch. In addition, tests took an average of only 12 minutes to complete.

Execution tracing. Execution tracing provides an alternate means to verify patches by comparing the behavior of a driver before and after applying the patch. We used tracing to verify that SymDrive can distinguish between patches that change the driver/device interactions and those that do not, such as a collateral evolution. We tested five patches to the `8139too` network driver that

refactor the code, add a feature, or change the driver's interaction with the hardware. We executed the original and patched drivers and record the hardware interactions. Comparing the traces of the before and after-patch drivers, differing I/O operations clearly identify the patches that added a feature or changed driver/device interactions, including which functions changed. As expected, there were no differences in the refactoring patches.

We also apply tracing to compare the behavior of drivers for the same device across operating systems. Traces of the Linux `8139too` driver and the FreeBSD `rl` driver show differences in how these devices interact with the same hardware that could lead to incorrect behavior. In one case, the Linux `8139too` driver incorrectly treats one register as 4 bytes instead of 1 byte, while in the other, the `rl` FreeBSD driver uses incorrect register offsets for a particular supported chipset. Developers fixed the Linux bug independently after we discovered it, and we validated the FreeBSD bug with a FreeBSD kernel developer. We do not include these bugs in the previous results as they were not identified automatically by SymDrive.

These bugs demonstrate a new capability to find hardware-specific bugs by comparing independent driver implementations. While we manually compared the traces, it may be possible to automate this process.

5.6 Comparison to other tools.

We compare SymDrive against other driver testing/bug-finding tools to demonstrate its usefulness, simplicity, and efficiency.

S^{2}E. In order to demonstrate the value of SymDrive's additions to S^{2}E, we executed the `8139too` driver with only annotations to the driver source guiding path exploration but without the test framework or SymGen to prioritize relevant paths. In this configuration, S^{2}E executes using *strict consistency*, wherein the only source of symbolic data is the hardware, and maximizes coverage with the MaxTbSearcher plugin. This mode is the default when a developer does not write API-specific plugins; results improve greatly when these plugins are available [15]. We ran S^{2}E until it ran out of memory to store paths and started thrashing after 23 minutes.

During this test, only 33% of the functions in the driver were executed, with an average coverage of 69%. In comparison, SymDrive executed 93% of functions with an average coverage of 83% in 2½ hours. With S^{2}E alone, the driver did not complete initialization and did not attempt to transmit packets. In addition, S^{2}E's annotations could not be made on the driver source, but must be made on the binary instead. Thus, annotations must be regenerated every time a driver is compiled.

Adding more RAM and running the driver longer would likely have allowed the driver to finish executing the initialization routine. However, many uninteresting paths would remain, as S^{2}E has no automatic way to prune them. Thus, the developer would still have considerable difficulty invoking other driver entry points, since S^{2}E would continue to execute failing execution paths.

In order for S^{2}E to achieve higher coverage in this driver, we would need a plugin to implement a relaxed consistency model. However, the `8139too` driver (v3.1.1) calls 73 distinct kernel functions, which would require developer effort to code corresponding functions in the plugin.

Static-analysis tools. Static analysis tools are able to find many driver bugs, but require a large effort to implement a model of operating system behavior. For example, Microsoft's Static Driver Verifier (SDV) requires 39,170 lines of C code to implement an operating system model [31]. SymDrive instead relies on models only for the I/O bus implementations, which together account for 715 lines of code for 5 buses. SymDrive supports FreeBSD with only 491 lines of OS-specific code, primarily for the test framework, and can check drivers with the debugging facilities already included in the OS.

In addition, SDV achieves much of its speed through simplifying its analysis, and consequently its checkers are unable to represent arbitrary state. Thus, it is difficult to check complex properties such as whether a variable has matched allocation/free calls across different entry points.

Kernel debug support. Most kernels provide debugging to aid kernel developers, such as tools to detect deadlock, track memory leaks, or uncover memory corruption. Some of the test framework checkers are similar to debug functionality built into Linux. Compared to the Linux leak checker, kmemleak, the test framework allows testing a single driver for leaks, which can be drowned out when looking at a list of leaks across the entire kernel. Furthermore, writing checkers for SymDrive is much simpler: the Linux 3.1.1 kmemleak module is 1,113 lines, while, the test framework object tracker, including a complete hash table implementation, is only 722 lines yet provides more precise results.

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.

DDT and S^{2}E. The DDT and S^{2}E systems have been used for finding bugs in binary drivers [14, 15, 26]. SymDrive is built upon S^{2}E but significantly extends its capabilities in three ways by leveraging driver source code. First and most important, SymDrive automatically detects the driver/kernel interface and generates code to interpose checkers at that interface. In contrast, S^{2}E requires programmers to identify the interface manually and write plugins that execute *outside the kernel*, where kernel symbols

are not available, though S^2E and SymDrive both support re-using existing testing tools. Second, SymDrive automatically detects and annotates loops, which in S^2E must be identified manually and specified as virtual addresses. As a result, the effort to test a driver is much reduced compared to S^2E. Third, checkers in SymDrive are implemented as standard C code executing in the kernel, making them easy to write, and are only necessary for kernel functions of interest. When the kernel interface changes, only the checkers affected by interface changes must be modified. In contrast, checkers in S^2E are written as plugins outside the kernel, and the consistency model plugins must be updated for all changed functions in the driver interface, not just those relevant to checks.

Symbolic testing. There are numerous prior approaches to symbolic execution [9, 10, 13, 20, 26, 39, 40, 43, 45]. However, most apply to standalone programs with limited environmental interaction. Drivers, in contrast, execute as a library and make frequent calls into the kernel. BitBlaze supports environment interaction but not I/O or drivers [37].

To limit symbolic execution to a manageable amount of state, previous work limited the set of symbolically executed paths by applying smarter search heuristics and/or by limiting program inputs [11, 21, 26, 27, 28, 44], which is similar to SymDrive's path pruning and prioritization.

Other systems combine static analysis with symbolic execution [16, 18, 19, 36]. SymDrive uses static analysis to insert checkers and to dynamically guide the path selection policy from code features such as loops and return values. In contrast, these systems use the output of static analysis directly within the symbolic execution engine to select paths. Execution Synthesis [45] combines symbolic execution with static analysis, but is designed to reproduce existing bug reports with stack traces, and is thus complementary to SymDrive.

Static analysis tools. Static analysis tools can find specific kinds of bugs common to large classes of drivers, such as misuses of the driver/kernel [3, 4, 5, 31, 35] or driver/device interface [25] and ignored error codes [23, 41]. Static bug-finding tools are often faster and more scalable than symbolic execution [8].

We see three key advantages of testing drivers with symbolic execution. First, symbolic execution is better able to find bugs that arise from multiple invocations of the driver, such as when state is corrupted during one call and accessed during another. It also has a low false-positive rate because it makes few approximations. Second, symbolic execution has full access to driver and kernel state, which facilitates checking driver behavior. Furthermore, checkers that verify behavior can be written as ordinary C, which simplifies their development, and can track arbitrary runtime state such as pointers and driver data. Symbolic execution also supports the full functionality of C including pointer arithmetic, aliasing, inline assembly code, and casts. In contrast, most static analysis tools operate on a restricted subset of the language. Thus, symbolic execution often leads to fewer false positives. Finally, static tools require a model of kernel behavior, which in Linux changes regularly [22]. In contrast, SymDrive executes checkers written in C and has no need for an operating system model, since it executes kernel code symbolically. Instead, SymDrive relies only on models for each I/O bus, which are much simpler and shorter to write.

Test frameworks. Test frameworks such as the Linux Test Project (LTP) [24] and Microsoft's Driver Verifier (DV) [29, 30] can invoke drivers and verify their behavior, but require the device be present. In addition, LTP tests at the system-call level and thus cannot verify properties of individual driver entry points. SymDrive can use these frameworks, either as checkers, in the case of DV, or as a test program, in the case of LTP.

Formal specifications for drivers. Formal specifications express a device's or a driver's operational requirements. Once specified, other parts of the system can verify that a driver operates correctly [6, 38, 42]. However, specifications must be created for each driver or device. Amani et al. argue that the existing driver architecture is too complicated to be formally specified, and propose a new architecture to simplify verification [2]. Many of the challenges to static verification also complicate symbolic testing, and hence their architecture would address many of the issues solved by SymDrive.

7 Conclusions

SymDrive uses symbolic execution combined with a test framework and static analysis to test Linux and FreeBSD driver code without access to the corresponding device. Our results show that SymDrive can find bugs in mature driver code of a variety of types, and allow developers to test driver patches deeply. Hopefully, SymDrive will enable more developers to patch driver code by lowering the barriers to testing. In the future, we plan to implement an automated testing service for driver patches that supplements manual code reviews, and investigate applying SymDrive's techniques to other kernel subsystems.

Acknowledgments

This work is supported by the National Science Foundation grants CNS-0745517 and CNS-0915363 and by a gift from Google. We would like to thank the many reviewers who provided detailed feedback on our work, and for early feedback from Gilles Muller and Julia Lawall. We would also like to thank the S^2E developers, who provided us a great platform to build on. Swift has a significant fi-

nancial interest in Microsoft.

References

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

[2] S. Amani, L. Ryzhyk, A. Donaldson, G. Heiser, A. Legg, and Y. Zhu. Static analysis of device drivers: We can do better! In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, et al. Thorough static analysis of device drivers. In *EuroSys*, 2006.

[4] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. In *Commun. of the ACM*, volume 54, July 2011.

[5] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. In *Commun. of the ACM*, volume 54, June 2011.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software*, 1975.

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

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

[12] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

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

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[15] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[16] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.

[17] Coverity. Analysis of the Linux kernel, 2004. Available at http://www.coverity.com.

[18] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *EuroSys*, 2011.

[19] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[21] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, 2008.

[22] Greg Kroah-Hartman. The Linux kernel driver interface. http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt, 2011.

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

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

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

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

[27] E. Larson and T. Austin. High coverage detection of input-related security facults. In *USENIX Security*, 2003.

[28] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[29] Microsoft. Windows device testing framework design guide. http://msdn.microsoft.com/en-us/library/windows/hardware/ff539645%28v=vs.85%29.aspx, 2011.

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

[31] Microsoft Corporation. Static Driver Verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, May 2010.

[32] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.*, 2002.

[33] S. Nelson and P. Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. www.linuxplumbersconf.org/2011/ocw/sessions/243. In Linux Plumbers Conference, 2011.

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

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

[36] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI 2011*, 2011.

[37] C. S. Păsăreanu et al. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

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

[39] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, 2005.

[40] D. Song et al. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[41] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *DSN*, 2006.

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

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

[44] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symp. on Security and Privacy*, 2006. IEEE Computer Society.

[45] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.