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, a single driver may support dozens of devices, and a developer may not have access to any of them. As a result, many Linux driver patches include the comment "compile tested only." Furthermore, testing error-handling code is difficult, as it requires faulty inputs from the device.

SymDrive is a system for testing Linux drivers without their devices. The system uses symbolic execution to remove the need for hardware, and provides three new features beyond prior symbolic-testing tools. First, SymDrive greatly reduces the effort of testing a new driver with a static-analysis and source-to-source transformation tool. Second, SymDrive allows checkers to be written as ordinary C and execute in the kernel, where they have full access to kernel and driver state. Finally, SymDrive provides an execution-differencing tool to identify how a patch changes I/O to the device. In applying SymDrive to 21 Linux drivers in 11 classes using 5 buses, we found 37 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 program-analysis tools available for user-mode code, such as Valgrind [30]. The need for hardware can prevent testing altogether: kernel developers often do not have access to the device needed by a driver, and thus many driver 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. Furthermore, 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 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.

This paper presents *SymDrive*, a system to test Linux drivers without devices. A developer can compile the driver for SymDrive, load it into SymDrive's virtual machine, and run standard tests. SymDrive improves driver quality by enabling (i) broader testing of drivers, because anybody can test almost any driver, and (ii) deeper testing of drivers, by exploring error-handling code more thoroughly.

We built SymDrive on S^{2}E [12], a system that provides system-level *symbolic execution*. SymDrive uses the symbolic execution capability of S^{2}E to simulate all possible hardware inputs to a device driver, thereby eliminating the need for the device. In addition, S^{2}E can further enhance test coverage by making other inputs to the device 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 consists of three components: (i) a modified version of the S^{2}E symbolic-execution engine; (ii) *SymGen*, a static-analysis and code transformation tool that analyzes and prepares drivers for testing; and (iii) a *test framework* that invokes checkers for verifying and validating driver behavior. SymDrive extends S^{2}E's limited support for driver testing to include more forms of I/O, including memory-mapped I/O and DMA, and more classes of drivers. At present, SymDrive supports five buses and has been tested with eleven driver classes.

While prior systems enabled symbolic execution of drivers, they did not provide the ability to do large numbers of drivers or a wide variety of drivers, as evidenced by their testing of only a few drivers in two classes [23, 12]. These systems require extensive programmer effort to annotate program code features and to encode the program's interface. The major contribution of SymDrive is to greatly reduce this effort with SymGen, which uses static analysis to identify key features of the driver and code generation to produce an instrumented driver with callouts to test code and with hints to S^{2}E to speed execution. Many drivers can be tested without any modifications to the driver, while others require a few annotations at locations identified by SymDrive to assist symbolic execution.

The test framework complements symbolic execution by detecting incorrect driver behavior. The framework invokes *checkers*, which are short C functions with 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. Us-

ing bugs and kernel programming requirements culled from code, documentation, and mailing lists, we wrote 49 checkers comprising 312 lines of code to enforce rules that maintainers commonly check during code reviews: matched allocation/free calls across entry points, no memory leaks, and proper use of kernel APIs.

Finally, SymDrive provides an *execution-differencing* mechanism for comparing execution across different driver revisions. This facility can ensure that patches leave certain functionality unchanged such as the order of I/O operations, and can help diagnose driver failures on real devices by identifying where behavior diverges.

The key contributions of SymDrive are:

- SymDrive is the first symbolic testing system for drivers that requires very little programmer effort due to its use of static analysis and code generation.
- SymDrive is the first symbolic testing system to support many classes of drivers and many I/O buses, allowing it to be used on a much greater portion of the driver code base.
- SymDrive provides a simple and extensible mechanism for programmers to verify driver behavior, and often requires little effort beyond writing an assertion in C.

To demonstrate these contributions, we applied SymDrive to 21 drivers in 11 classes, and found 37 bugs, including two security vulnerabilities. To the best of our knowledge, no other tool besides static analysis has examined as many drivers for bugs. 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.

2 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. Most recently, the DDT and S^{2}E systems have been used for finding bugs in binary drivers [23, 12]. SymDrive is built upon S^{2}E but significantly extends its capabilities in four ways. 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. Second, SymDrive automatically detects and annotates loops, which in S^{2}E must be identified manually and specified as virtual addresses. As a result, the effort to test a driver is much reduced compared to S^{2}E. Third, SymDrive supports many (11) driver classes with little effort as well as many different buses (5). In contrast, S^{2}E only supports a single driver class, network drivers, on a single bus, PCI. Fourth, 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. In addition, when the kernel interface changes, only the checkers affected by interface changes must be modified. In contrast, checkers in S^{2}E are again 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 [8, 9, 11, 17, 23, 35, 39, 40, 36]. 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 [33].

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 [10, 18, 23, 24, 25], which is similar to SymDrive's path pruning and prioritization.

Other systems combine static analysis with symbolic execution [14, 32, 13, 16]. These systems use the analysis to identify code and paths for additional symbolic testing, while SymDrive uses it to inform the symbolic execution engine of program features. Execution Synthesis [40] combines symbolic execution with static analysis, but is designed to reproduce existing bug reports, 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 [2, 4, 31, 28, 3] or driver/device interface [22] and ignored error codes [20, 37]. Static bug-finding tools are often faster and more scalable than symbolic execution [7].

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. Second, symbolic execution has full access to driver and kernel state, making it possible to use this state in checking driver behavior. Furthermore, checkers that verify behavior can be written as ordinary C code, which enables more programmers to write checkers. 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. As a result, symbolic execution often leads to fewer false positives. Finally, static tools require a model of kernel behavior, which in Linux changes regularly [19]. In contrast, SymDrive executes checkers written in C and has no need for an operating

system model, since it executes kernel code symbolically. SymDrive can also execute existing test suites, which allows verification of functionality unique to a driver or class of drivers.

Test frameworks. Test frameworks such as the Linux Test Project (LTP) [21] and Microsoft's Driver Verifier (DV) [27, 26] 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 (also used by DDT), or in the case of LTP, as a test program.

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 [38, 34, 5]. However, specifications must be created for each driver or device. Once created, though, they could be used in SymDrive to verify driver behavior.

3 Design

The SymDrive architecture focuses on thorough testing of drivers to ensure the code does not crash, hang, or incorrectly use the kernel/driver interface. Thus, SymDrive specifically targets test situations where the driver code is available, and uses that code to simplify testing.

We have two goals for SymDrive: (i) allow the developer to verify and validate any relevant part or parts of a driver to the greatest extent possible, while (ii) minimizing the developer effort for testing each new driver. The first goal enables *deeper* testing of drivers for higher quality code, while the second goal enables *broader* testing of drivers by eliminating the requirements for hardware.

SymDrive addresses these goals using symbolic execution, static source-code analysis and transformation, and a fine-grained test framework. 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.

3.1 Symbolic Execution

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

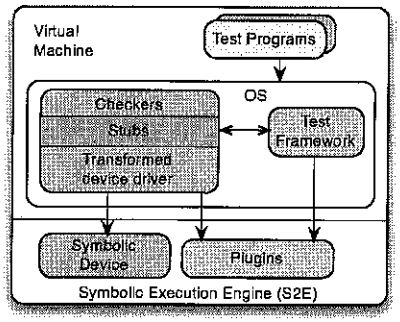

Figure 1: **The SymDrive architecture. A developer produces the transformed driver with SymGen and can write checkers and test programs to verify correctness.**

of the comparison, each with the symbolic value constrained by the chosen outcome of the comparison. For example, when used in a conditional statement, 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 $x > 5$. In places where specific values are needed, such as producing output to the screen, symbolic execution can *concretize* data, by producing a single value that satisfies all constraints over the data.

Symbolic execution with S^{2}E. SymDrive is built on a modified version of the S^{2}E symbolic execution framework. S^{2}E executes a complete virtual machine as the program under test. Thus, symbolic data can be used anywhere in the operating system, drivers, or applications. S^{2}E is a modified QEMU 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.

S^{2}E supports *plug-ins*, which it invokes to record information or to modify execution. SymDrive relies on plugins to implement symbolic PCI hardware, path selection, and minor features such as monitoring code coverage.

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

- *Simplicity.* Existing symbolic testing tools require extensive developer effort to test a single class of drivers, plus additional effort to test each individual driver because of the efficiency problem described below. For example, supporting Windows NDIS drivers in S^{2}E requires 2,230 lines of code. Thus, these tools have only been applied to a few driver classes and a small set of drivers. In addition, tests must be coded like debugger extensions, with calls

to read and write remote addresses, rather than as normal test code. Expanding testing to many more drivers requires new techniques to automate most or all of the testing effort.

- *Efficiency.* Symbolic execution creates many paths through a driver, and consequently may have difficulty executing deeply into the code. Existing path-search strategies in DDT and S^{2}E favor exploring new code, but may not execute far enough down a path to test all functionality, such as unloading a driver, because of path explosion.
- *Information.* The symbolic execution engine needs information about the code executing to test it efficiently. For example, it may be helpful to distinguish which paths through a driver successfully complete initialization and which do not. S^{2}E and DDT can test binary drivers, but at the cost of complex, manually-written annotations that depend on kernel function names and behavioral details, which are difficult for programmers to provide. Thus, little information is available to the engine.
- *Symbolic hardware.* Both S^{2}E and DDT rely on plug-and-play functionality in PCI to create symbolic hardware. Other buses, particularly those used in smartphones and embedded devices, statically configure devices and thus require additional techniques. Furthermore, S^{2}E and DDT do not support the full range of I/O behavior, including all forms of memory-mapped I/O and DMA and buses beyond PCI.

Thus, our work focuses on improving the state of the art to greatly simplify the use of symbolic execution for testing, and broaden its applicability to almost any driver in any class on any bus. SymDrive provides solutions to all four open problems through its combined use of static analysis and code generation, symbolic hardware, and the test framework.

3.2 Static Analysis and Code Generation

SymDrive relies on driver source to simplify testing with the SymGen tool. SymGen analyzes driver code to identify events relevant to testing, such as function boundaries and loops. At the entry and exit of every function in the driver or imported from the kernel, SymGen generates code to invoke a *stub* function. The stub can invoke test code, for example to verify pre- and post-conditions, and can notify S^{2}E to prune paths. SymGen also identifies loops and inserts notifications to S^{2}E to facilitate faster execution by limiting the number of iterations. For complex code that slows testing, SymGen supports programmer-supplied annotations to simplify or disable the code during testing. For example, a driver that verifies a checksum over device data may need to be modified to allow any checksum value.

SymDrive uses this instrumented driver to address the open problems discussed previously in four ways. First, within the driver, SymDrive explores many code paths. However, when returning control to the kernel, SymDrive terminates all paths but one. Second, SymDrive concretizes all symbolic data when returning to the kernel, which means that the kernel will not generate additional paths. When the driver invokes the kernel, though, SymDrive does not concretize any data or terminate paths. Third, SymDrive can prioritize paths that execute successfully, to allow the developer to target driver testing within particular functions. Finally, SymGen is fully automatic. Developers do not need to write any code to test drivers in other classes or that use new kernel functionality.

3.3 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. Other driver types, such as SPI and I^{2}C, use wrapper functions for these similar primitives operations.

To provide symbolic hardware, S^{2}E provides symbolic data of the appropriate size each time the driver performs a read operation using memory or port I/O. Similarly, the VMM treats the contents of DMA memory, indicated through DMA mapping functions, as symbolic. The VMM does not insert symbolic interrupts; instead, SymDrive invokes the driver's interrupt handler each time control passes from the driver to the kernel. For buses such as I^{2}C that use I/O wrapper functions, SymDrive rewrites the wrappers to return symbolic data.

3.4 Test Framework

SymDrive relies on a *test framework*, executing within the virtual machine, to verify and validate driver behavior (see Figure 1). The test framework provides three services to support testing. First, it invokes checkers that verify driver behavior. Second, it provides a support library that simplifies the writing of checkers. Finally, it notifies the VMM of the current state of execution for pruning paths and for tracing driver execution.

A checker is a function that executes when transitioning between the driver and the kernel and that verifies assertions over driver behavior. Each function in the driver/kernel interface can, but need not, have its own checker. Drivers invoke the test framework from stubs, described above, which call separate checkers at every function in the driver/kernel interface, or even every function in the driver. For example, the precondition checker for a lock function could verify that the lock has been initialized.

Component	LoC
Changes to S^2E	1,954
SymGen	3,442
Test framework	3,760
Checkers	312
Linux kernel changes	153

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

The test framework provides a *support library* that simplifies authoring checkers by providing much of the functionality needed. First, it provides state variables to track the state of the driver, such as whether it has completed execution, and the state of the thread, such as whether it can be rescheduled. The library also provides an object tracker to record kernel objects currently in use by 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.

Finally, the test framework notifies the VMM of every function entry and exit. This information is used for path pruning and prioritization, and to provide additional symbolic data to increase coverage. It also provides a stack trace for *execution differencing*, a feature that compares the I/O behavior of two versions of a driver, similar to delta execution [15]. On every function call and return, the test framework notifies S^2E of the current call stack. A plugin within S^2E records each I/O and the call stack of the operation. These traces can then be compared to see how execution differs across patches.

4 Implementation

We implemented SymDrive for the Linux kernel, as it provides the largest number of drivers to test. The design applies to any operating system supported by S^2E, and only the test framework code is specialized to Linux. We made small changes to Linux under conditional compilation to print failures and panics to the S^2E log, as well as to register the module under test with S^2E. The breakdown of SymDrive code is shown in Table 1.

We next describe the implementation of SymDrive's three major components: the modified S^2E virtual machine, the SymGen tool, and the test framework.

4.1 Virtual Machine

SymDrive uses S^2E [12] version 1.1-10.09.2011, itself based on QEMU [6] and KLEE [9], for symbolic execution. S^2E provides the execution environment and constraint solving capability necessary for symbolic execution. It also implements the path forking that takes place as symbolic values are compared against each other. All driver and kernel code, including the test framework, executes within the S^2E VM.

We augment S^2E with new opcodes for the test framework that pass information into the VMM and its plugins. S^2E uses invalid x86 opcodes for communication between code within the virtual machine and the VMM. SymDrive adds opcodes to provide additional control over the executing code. These opcodes are either inserted into driver code by SymGen or invoked by the test framework. The new opcodes fall into three categories. First, a set of opcodes control whether memory regions are symbolic, and are used when mapping data for DMA and when entering the kernel, in which case all data becomes concrete. Second, a set of opcodes control path scheduling by adjusting priority, search strategy, or killing other paths. We discuss their use in the following sections. Finally, a set of opcodes provide tracing for execution differencing.

4.1.1 Symbolic Hardware

SymDrive provides symbolic devices for the driver under test, while at the same time emulating the other devices in the system. A symbolic device provides three major behaviors. First, it must be discovered, so the kernel loads the appropriate driver. Second, it must provide methods to read and write data to the device. Third, it must support interrupts and DMA when needed. SymDrive currently supports 5 buses: PCI (and its variants), I^2C (including SMBus), Serial Peripheral Interface (SPI), General Purpose I/O (GPIO), and Platform, although not USB.

Discovering devices. When possible, SymDrive creates symbolic devices in the S^2E virtual machine and lets existing bus code discover the new device and load the appropriate driver. For example, SymDrive uses S^2E's symbolic hardware plugin to create a virtual PCI device with the desired I/O memory configuration and PCI identifiers, both ~~easily located~~ in driver source code. This device triggers the PCI code in Linux to load the driver.

Not all buses support this plug-and-play functionality. For some buses, such as I^2C, the kernel or another driver normally creates a statically configured device object during initialization. For such devices, we created a small kernel module that invokes the kernel to create the desired symbolic device.

SymDrive can also make the device configuration space symbolic after loading the driver by returning symbolic data from PCI bus functions with the test framework. PCI devices use this region of I/O memory for plug-and-play information, such as the vendor and device identifiers. If this memory is symbolic, the driver will 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 drivers do a mix of programmed I/O and DMA (described below). 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 and return symbolic data from reads. For drivers that invoke a bus function to perform I/O, such as I^2C, the test framework overrides the bus function to return symbolic data on reads and to silently drop writes.

Symbolic interrupts and DMA. The test framework provides additional symbolic interrupt and DMA support. When a driver requests an interrupt, the test framework invokes the interrupt handler on every subsequent transition from the driver into the kernel. This ensures the interrupt handler is called often enough to keep the driver executing successfully. Each stub generates at most 5 interrupts, to prevent loops from slowing testing.

When a driver invokes a DMA mapping function, such as `dma_alloc_coherent`, the test framework directs S^2E to make the memory act like a memory-mapped I/O region: each read returns a new symbolic value, and writing to the memory has no effect. This approach reflects the ability of the device to write the data via DMA at any time. When the driver unmaps the memory, the test framework directs S^2E to revert the region to normal symbolic data, so data written is available to subsequent reads and constraints persist across writes.

4.2 SymGen

SymDrive employs SymGen, based on CIL [29], to analyze and transform driver code for testing. SymGen generates stubs and instruments driver functions to interpose stubs on all function calls.

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. In addition, for each kernel function the driver imports, SymGen generates a stub function with the same signature that wraps the function. These stubs ensure that the test framework can interpose on all calls within the driver as well as calls into the kernel.

To support pre- and post-condition assertions, stubs invoke test framework 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 by creating a function with the appropriate name and exporting it to the symbol table with the `EXPORT_SYMBOL` directive.

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 `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`.

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.

Instrumentation. 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 kernel/driver interface. SymGen also instruments the start, end, and body of each loop with opcodes. These direct the VMM to prioritize and deprioritize paths depending on whether they exit the loop quickly. This instrumentation replaces much of per-driver effort required by S^2E to identify loops, as well as the per-class effort of writing a consistency model for every function in the driver/kernel interface.

4.3 Test Framework

The test framework is a library invoked from instrumented drivers, and serves two purposes. First, the test framework provides an API library that provides common test functions to simplify checkers. We describe checkers more in Section 5. Second, the test framework implements the logic for deciding which paths through the driver and kernel to execute, which is described in Section 4.4.

Library API. The library API provides support routines to simplify checkers, as well as a set of checkers for common kernel-interface functions. The library provides a global state variable that a checker can use to store information across invocations or to pass information to another checker. Checkers use state variables to