~~SymDrive: Testing Drivers without Devices~~SymDrive: Who Needs a Device to Test a Driv

~~Matthew J. Renzelmann, Asim Kadav and Michael M. Swift Computer Sciences Department, University of Wisc~~

Abstract

Device-driver development and testing is a complex and error-prone undertaking. For example, ~~testing error-handling code is difficult, because it requires faulty inputs from the device. In addition,~~ a single driver may support dozens of devices, and a developer may not have access to any of them. ~~Consequently~~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 and FreeBSD drivers without their devices~~present~~. 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 ~~checkers are ordinary C code~~ 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-tracing tool to identify how a patch changes I/O to the device and to compare ~~device-driver~~ 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 [**?**]. The need for hardware can prevent testing altogether: ~~Linux and FreeBSD~~ kernel developers often do not have access to the device needed by a driver, and thus over two dozen 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. 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. In addition, thorough testing of failure-handling code is time consuming and requires exhaustive fault-injection tests with a range of faulty inputs.

~~Static~~ While static analysis tools such as Coverity [**?**] and Microsoft's Static Driver Verifier [**?**] can find many bugs~~quickly. However~~, these tools are tuned for fast analysis of large amounts of code and miss large aspects of driver behavior, such as bugs that propagate across multiple invocations of the driver.

We address these challenges using *symbolic execution* to test device drivers. This approach executes driver code on all possible device inputs, and allows (i) driver code to execute without the device present, and (ii) more thorough coverage of driver code, including error handling code. While DDT [**?**] and S^2E [**?**] applied symbolic execution to driver testing~~previously~~, these systems did not address the many complexities of symbolic execution as evidenced by ~~their application to~~ testing only a few drivers in two classes ~~. Testing a wider variety of drivers~~on one bus. Testing additional drivers, classes of drivers, or drivers on other buses requires substantial developer effort to encode the driver/bus interfaces into the testing tool.

This paper presents *SymDrive*, a system to test Linux and FreeBSD drivers without devices. Compared with prior symbolic execution tools, SymDrive greatly reduces developer effort. SymDrive uses static analysis to identify key features of the driver code, such as entry-point functions and loops. ~~With~~ Based on this analysis, SymDrive produces an instrumented driver with callouts to test code~~that allows many drivers to~~ , and hints to improve testing performance. As a result, many drivers can be tested with no modifications. The remainder require a few annotations ~~to assist symbolic execution at locations that~~ at locations identified by SymDrive ~~identifies~~to assist symbolic execution.

We designed SymDrive for three purposes. First, a driver developer can use SymDrive to test ~~driver patches by thoroughly executing all branches through~~ a patch and target thorough exploration of the changed code. 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, similar to static analysis tools, and perform broad testing of an entire driver

with little developer input.

SymDrive is built with the S^{2}E system by Chipounov et al. [**?**] , 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^{2}E enables SymDrive to further enhance code coverage by making 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^{2}E 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~~a developer, such as identifying the driver/kernel interface, ~~and also provides~~ as well as providing hints to S^{2}E to speed testing. ~~Consequently~~As a result, little effort is needed to apply SymDrive to additional drivers, ~~driver classes~~ classes of drivers, 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: matched allocation/free calls across entry points, no memory leaks, and proper use of kernel APIs.

In addition, SymDrive provides an *execution-tracing* mechanism for logging the path of driver execution, including the instruction pointer and stack trace ~~of~~ for 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 different operating systems.

We demonstrate SymDrive's value by applying it to 26 drivers, and ~~find~~ found 39 bugs, including two security vulnerabilities. We also ~~find~~ found 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, which is largely limited by the ability of user-mode tests to invoke driver entry points. When ~~we use to execute code changed by~~ applied to driver patches, SymDrive ~~achieves~~ achieved over 95% coverage on 12 patches in 3 drivers.

2 Motivation

The goal of our work is to improve driver quality through more 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, which consist both of other tools as well as testing on real hardware. Second, SymDrive must minimize developer effort to test a new driver and therefore support many device classes, buses, and operating systems. Finally, SymDrive must be fast enough that developers can apply it to each of their patches.

We have two use cases for SymDrive: *deeper* testing of drivers, by providing high coverage of individual patches, and *broader* testing of drivers by allowing more people to test drivers and find bugs.

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~~ program inputs 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 comparison 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 shows ~~the state of the executing path~~ a stack trace at the failure site before resuming execution of another path.

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, including drivers and applications. S^{2}E 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 logically similar to a thread executing a different outcome of a branch, and the scheduler () periodically switches execution to a different path.~~

S^2E supports *plug-ins*, which it invokes to record information or to modify execution. SymDrive relies on 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 benefit: 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 a driver.

Symbolic execution eliminates the hardware requirement, because it can use symbolic data for all device input. Alternative testing approaches rely on a programmer to create a device model [**?**]. Models allow driver/device interface verification, but require ~~a large~~ much more effort to faithfully replicate device behavior.

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. As SymDrive executes the driver code, it effectively builds a model of the device's behavior by tracking successful and failing execution paths, and finds bugs as it does so.

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

In ~~comparison~~ contrast 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 invokes many driver entry points in series, allowing it to find bugs that span invocations, such as resource leaks. In contrast, static analysis tools tend to focus on bugs 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*. It is useful to distinguish and prioritize paths that successfully complete ~~successfully. For example, if driver initialization fails, the operating system could not otherwise invoke most driver entry points~~initialization and allow the remainder of the driver to execute. 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, such as unloading a driver, because of path explosion.

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~~ 2,230 lines of code~~specific to this driver class~~. 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 most or all of the testing effort.

Specification. Finally, symbolic execution by itself does not provide any specification of correct behavior. A "hello world" driver is correct but does not initialize a network device. 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 specification.

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.

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. ~~We target~~ Thus, SymDrive specifically targets test situations where the driver code is available, and uses that code to simplify testing by combining symbolic execution, static source-code analysis and transformation, and a fine-grained test framework.