track the state of the driver, such as whether initialization completed successfully, as well as the state of execution, such as whether blocking is allowed. The variables are copied for each execution path.

The library provides an object tracker, implemented as a hash table keyed by object address. A checker can store arbitrary data about an object, such as a lock's state or an allocation's size. The library also provides common functionality based on the object tracker to record which function allocated an object, to ensure the corresponding free function is used, and to track the state of a lock, to record it having been initialized or acquired.

Execution differencing. The test framework can generate traces to compare the execution of two versions of a driver. A developer can enable tracing via a command-line tool that notifies an S^{2}E plugin to enable tracing. 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 stack trace enables a developer to determine where execution diverges. The traces are stored as a trie (prefix tree) to represent multiple paths through the code compactly, and can be compared using the Linux diff utility. The traces are subject to timing variations and different thread interleavings. Thus, they are most useful when testing patches that modify few driver entry points, or to compare patches function-by-function.

Usage The test framework is a kernel module that supports several load-time module parameters for controlling its behavior. When loading the test framework with `insmod`, developers can direct the test framework to enable high-coverage mode (described in Section 4.4.2), tracing, and a 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.

4.4 Maximizing Code Coverage

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, by exploring many paths through a function. To execute far into the driver, SymDrive aggressively prunes paths that do not advance execution. To execute broadly, SymDrive introduces additional symbolic data and can terminate paths that leave the driver.

4.4.1 Executing 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*.

Favor-success scheduling. Executing past driver initialization is difficult because the code has many conditional branches to support multiple chips in multiple configurations. Initializing a sound driver, for example, may require branching on hardware-specific details 1,000 times. SymDrive mitigates this problem with a *favor-success* path-selection algorithm that prioritizes successfully executing paths. 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. This opcode causes the current path to be explored further before switching to another path.

At every function exit, the test framework notifies S^{2}E 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 outside the range of [-4095, -1], which are standard Linux 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, allowing packets to be sent.

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. Kernel functions that require treatment include those that use the `PTR_ERR` macro to return error values as pointers and `dma_alloc_coherent_mask`, which returns a bitmask. In these functions, the stubs do not change a path's priority based on the return value.

The test framework prunes paths when control returns to the kernel. It uses the opcodes described previously to concretize all data in the virtual machine, so the kernel executes on real values and will not fork new paths. In addition, the test framework kills all other paths still executing in the driver. This ensures that a single path runs in the kernel and allows a developer to test the system interactively.

Loop elision. Loops are challenging for symbolic execution, because each iteration may fork new paths. S^{2}E provides an "EdgeKiller" plugin that a developer may use to terminate complex loops early, but requires developers to identify each loop's offset into the driver binary [12]. Moreover, loops that produce a value, such as a checksum calculation, cannot exit early without stopping the driver's progress.

SymDrive addresses loops explicitly by prioritizing those paths that exit the loop. SymGen inserts loop opcodes into the driver, as shown in Figure 2, to tell S^{2}E

```
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.**

which paths exit the loop, and should receive a priority boost. Thus, SymDrive does not need the EdgeKiller plugin. For loops that cannot be terminated, SymDrive notifies the developer during testing if a loop creates too many paths. This notification alerts the developer to use `#ifdef` statements to disable the code during testing.

4.4.2 Executing broadly

SymDrive provides a *high-coverage* mode for testing specific functions that 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^{2}E to favor unexecuted code (the default S^{2}E policy) rather than favoring successful paths. The test framework also terminates all paths that return to the kernel in order to focus execution on the driver. Finally, when the driver invokes a kernel function, the test framework makes the return value symbolic. For example, `kmalloc` returns a symbolic value constrained to be either NULL or a valid address, which tests error handling in the driver. This mode is similar to the local consistency mode in S^{2}E [12], but requires no developer-provided annotations or plugins, and supports all kernel functions that return standard error values.

SymDrive also improves code coverage by introducing additional symbolic data beyond I/O, which allows a driver to execute code that requires specific inputs from the kernel or applications. First, SymDrive can automatically make a driver's module parameters symbolic, which allows a driver to execute with all possible parameters. Second, checkers can 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. Third, SymDrive incorporates S^{2}E's ability to use symbolic data anywhere in the virtual machine, so a user-mode test program can make system call parameters symbolic.

looks like SDV

```
/* 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.**

4.5 Limitations

SymDrive is neither sound nor complete, though we are not aware of any false negatives among the checks that SymDrive supports, and we have not experienced any false positives. However, SymDrive cannot check for all kinds of bugs. First, SymDrive has no support for integer overflow. Second, SymDrive's aggressive path pruning may terminate a path that leads to a bug, causing SymDrive to miss it. Nonetheless, we find the tool to be a useful addition to the driver development and patching process.

seems likely. these would be false negatives

5 Example Checkers

We have implemented 49 checkers comprising 312 lines of code for a variety of common device-driver bugs using the test framework and library API. Writing a checker requires implementing checks within a call-out function. 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 test framework provides the global `state` variable. 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. As a result, they can find interprocedural bugs that span mul-

why wasn't this required in S2E?

unclear

how?

tiple invocations of the driver.

Not every behavior requirement needs a checker. Symbolic execution leverages the extensive checks already included as Linux debug options, such as for memory corruption and locking. Using these facilities with SymDrive can check behavior on more paths through the driver. In addition, any bug that causes a kernel crash or panic will be detected by S^{2}E and therefore requires no checker, unless the driver returns to the kernel and concretizes symbolic data in such a way as to mask the bug.

Execution Context. Linux prohibits the calling of functions that block when executing 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. For example, if the driver is executing the `start_xmit` path of a network driver, it can only allocate memory with the `GFP_ATOMIC` flag.

The test framework library provides a state machine to track the driver's current context. The test framework API tracks execution context using a stack. When entering the driver, the test framework updates the context based on the entry point. Each time the driver acquires or releases a spinlock, the test framework API pushes or pops the necessary context. In contrast to static-analysis tools, there is no need to specify which driver entry points are non-blocking.

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 test framework 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. A developer can use SymDrive to verify that collateral evolutions [31] are correctly applied by ensuring that patched drivers do not regress on any tests. In addition, SymDrive can ensure that the *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 test framework object tracker to store an allocation's address and length. We have 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 adding support for additional allocators down to adding a one-line call into the API. 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.

6 Evaluation

The purpose of the evaluation is to verify that SymDrive achieves its goals: can it execute driver code and find bugs quickly with minimal programmer effort?

As shown in Figure 2, we tested SymDrive on 21 drivers in 11 classes from several Linux kernels including 2.6.29 (13 drivers) and 3.1.1 (4 drivers). Four other drivers normally run only on Android-based phones, and were from other kernel distributions. Of the 21 drivers, we chose 14 as examples of a specific bus, while the remaining 7 were chosen because we either had the hardware or we found frequent patches to the driver 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.

6.1 Methodology

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. We ensure all network drivers attempt to transmit.
4. Optional: execute a workload.
5. Unload the driver.

If SymDrive reports warnings about too many paths, we annotate the driver code and repeat the operations. For most drivers, we run SymGen over just 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 tested each driver with 49 checkers for a variety of common bugs.

6.2 Bugs Found

We applied SymDrive to the 21 drivers listed in Table 2. Across these drivers, we found 37 distinct bugs described in Table 3.

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

Table 2: **Drivers shown here. Those in *italics* run on Android-based phones, and those followed by an asterisk are for embedded systems, and do not use the PCI bus. Line counts come from CLOC [1]. Times are in minute:second format.**

Bug Type	Bugs	Kernel / Checker	Cross EntPt	Paths	Ptrs
Hardware Dep.	5	4 / 1	2	4	4
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	37	15 / 22	20	14	19

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.**

Of these bugs, S^{2}E detected 15 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 37 bugs, 54% took place across driver entry points. For example, the akm8975 compass driver calls `request_irq` before it is ready to service interrupts. If a spurious interrupt occurs immediately after this call returns, the driver will crash, since the interrupt handler dereferences a pointer that is not yet initialized. In addition, 38% of the bugs occurred on a unique path through a driver, and 51% involved pointers and pointer properties. These capabilities are all difficult to achieve with static analysis.

6.3 Execution Differencing

We experiment with execution differencing to verify that SymDrive can distinguish between patches that change the driver/device interactions and those that do not. We test with the 8139too network driver and applied five patches from the mainline kernel that refactor the code, add a feature, or change the driver's interaction with the hardware. We use SymDrive to execute the original and patched drivers, to record the hardware interactions, and to achieve 100% coverage on the code affected by the patch. The differencing traces reported different I/O operations in the patches that added a feature or changed driver/device interactions, including which functions changed, and as expected, there were no differences in the refactoring patches.

6.4 Developer Effort

One of the goals of SymDrive is to minimize the effort to test a driver. As an example, we tested the `phantom` haptic driver from scratch in 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 only extra time spent due to SymDrive was an additional pass during compilation to run SymGen, which takes less than a minute, and 38 minutes of execution time.

Annotations. The only coding SymDrive requires are 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 21 drivers, only 5 required more than two annotations, and 8 required no annotations. In all cases, SymDrive printed a warning indicating where testing would benefit from an annotation.

Testing time. Symbolic execution is much slower than normal execution, but still faster than buying the device. We report the time to load and initialize a driver, which is required for any testing, in Table 2. The table also reports the time to unload the driver, which is necessary to detect resource leaks. 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 because we included the entire sound library (25,000 lines of code). Thus, 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 [31].

Kernel evolution. Near the end of development, we upgraded the test framework to support Linux 3.1.1 in-

stead of 2.6.29. The only changes we made were to update a few checkers because their corresponding kernel functions had changed. As the 49 checkers comprise 312 lines of code, these changes took little effort. The remainder of the test framework was unchanged. Thus, SymDrive's use of static analysis and code generation minimized the effort to maintain tests as the kernel evolves.

Comparison to other tools. As a comparison against the capabilities of S^{2}E without SymDrive's additions, we executed the 8139too driver with only annotations to the driver source guiding path exploration but without the test framework. In this configuration, S^{2}E executes using *strict consistency*, wherein the only source of symbolic data is the hardware, and it attempts to maximize coverage through use of its MaxTbSearcher plugin. We ran it until it started thrashing the page file, with a commensurate drop in CPU utilization, for a total of 23 minutes.

During this test, only 33% of the total functions in the driver executed at all, with an average coverage of 69%. The driver did not complete initialization successfully and did not attempt to transmit packets. Even this test is beyond the capabilities of S^{2}E, as it relied on programmer annotations in the source. With S^{2}E, the annotations must be made on the binary rather than the driver source, which requires regenerating annotations every time a driver is compiled. This result demonstrates the value of the test framework mechanisms to improve code coverage.

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, and many of these would require corresponding functions in the plugin. For example, the consistency-model plugin Windows NDIS driver in S^{2}E is 2230 lines. Since SymDrive relies on SymGen to generate code to implement a consistency model automatically, this effort is not necessary. Furthermore, with S^{2}E consistency plugins, much of this code would need to be updated when the kernel/driver interface changes.

Some of the test framework checkers are similar to debug functionality built into Linux. Compared to the Linux leak checker, kmemleak, the test framework provides a report specific to the driver under test, rather than for the entire kernel, which simplifies diagnosing the source of leaks. As a point of comparison, 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.

Driver	Touched Funcs.	Coverage	Time Serial	Time Parallel
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

Table 4: **Code coverage. Entries with an asterisk ran overnight or are otherwise not representative of the minimum time required.**

6.5 Coverage

One potential benefit of symbolic execution is high code coverage. 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"). In addition, the table gives the total time to run the tests on a single machine (serial) or if multiple machines are used (parallel), in which case the time is that of the longest execution run. 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 meaningfully improve from one run to the next.

Overall, SymDrive was able to execute over 80% of the functions in most drivers, and more than 80% of the code in those functions. We were limited for two reasons. For some drivers, we could not test all entry points. For example, econet requires additional software that we did not have, and other drivers required kernel-mode tests for entry points not accessible via system calls. Second, of the functions SymDrive did execute, additional inputs or symbolic data from the kernel was needed to test all paths. Encoding more of the kernel API semantics in checkers, which allows more data to be made symbolic, could help here.

To estimate the value of symbolic execution in improving test coverage, we tested the 8139too driver on a real network card using `gcov` to measure coverage. We loaded and unloaded the driver, and ensured that transmit, receive, and all ethtool functions executed. 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 this result is not directly comparable to the other coverage results in this section due to differing methodologies, it demonstrates that SymDrive can provide coverage similar to or better than that of running the

still seems not completely trivial to use. Perhaps better to target maintainers for whom the work quickly amortized

driver on real hardware.

7 Conclusions

SymDrive uses symbolic execution combined with a test framework and static analysis to test 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 use their existing tests with symbolic execution to exercise driver code. 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.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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*.

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

[10] 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.

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

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS 2011*.

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

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

[15] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA '07*.

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

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

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

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

[20] 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.

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

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

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

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

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

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

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

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

[29] 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.

[30] N. Nethercode and J. Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *PLDI*, 2007.

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

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

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

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

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

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

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

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

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

[40] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Eurosys 2010*.