#10 SymDrive: Testing Drivers without Devices

Comment notification

If selected, you will receive email when updated comments are available for this paper.

Rejected 282kB Wednesday 12 Jan 2011 5:29:27pm PST | 56c8af5f156722b7698cd2f5f691962c64622557

You are an **author** of this paper.

+ Abstract

Device driver development and testing has traditionally been a complex and error-prone undertaking. For example, dozens of patches committed to the Linux kernel include the [more]

+ Authors

M. Renzelmann, A. Kadav, M. Swift [details]

+ Topics and Options

	OveMer	RevCon	Nov	TecQua	PreQua	Rel	SuiShoPap
Review #10A	1	4	2	3	4	5	1
Review #10B	3	3	3	3	4	5	3
Review #10C	2	3	3	3	3	3	3
Review #10D	3	2	3	4	4	4	2

Edit paper

Reviews in plain text

Review #10A

Modified Tuesday 15 Mar 2011 3:14:02am PDT Plain text

Overall merit (?)

1. Reject (really no way this could be an ATC paper)

Reviewer confidence (?)

4. Very comfortable (my area)

Novelty (?)

2. One or only a few minor novel pieces

Technical quality (?)

3. About the technical depth you'd expect (typical ATC paper)

Presentation quality (?)

4. Very good

Relevance (?)

5. Excellent

Brief summary

The paper presents a system for testing Linux PCI and USB drivers, called SymDrive. Based on symbolic execution and symbolic hardware, the system can test drivers without needing access to the corresponding hardware devices. SymDrive relies on Klee to verify basic safety properties (e.g., buffer overflows) and on developer-provided checkers to verify high-level properties (e.g., memory leaks).

SymDrive uses hardware interaction traces in order to simulate concrete execution of a driver up to a certain point. The
traces are recorded while the driver runs concretely with physical hardware, and are then used on subsequent runs with the symbolic hardware to provide the driver with concrete responses (assuming the

sequence of hardware accesses stays the same in all runs). The tracing mechanism can also be used by SymDrive to verify whether changes to the driver induce changes to its hardware interaction sequence/protocol (assuming the hardware interaction is deterministic).

Main strengths

Testing of device drivers is an important problem, so progress in this area is important

Main weaknesses

Small contribution compared to existing published work

In some ways, seems to work less well than prior work

Evaluation is relatively weak (no info on achieved coverage, no systematica info on total testing time, no info on false positives, etc.)

Detailed comments for authors

This is a well built system that is clearly useful if used as part of driver development. The main concern is that the benefits over existing work are not brought out clearly.

The key idea of "symbolic hardware" that enables SymDrive to test drivers without devices appeared in the RWSet paper (TACAS '08) and then generalized in DDT (USENIX '10). This submission lists the following three advances: (1) using developer-written checkers to check high-level properties; (2) using hw interaction traces to fast-forward the execution when using symbolic hardware; and (3) using hw interaction traces to check the equivalence of a driver before and after refactoring the driver code.

On (1), it looks like SymDrive's checkers roughly correspond to DDT's OS-level checkers (like the way MSDV was used in DDT); the DDT page says one can plug any checkers into DDT, and they can operate both at the VM level and the OS kernel level. Could one write SymDrive as a collection of DDT checkers? Is there anything missing in DDT to do that?

Re. (2), SymDrive depends on hw interaction traces (as one might infer from the Evaluation, where only one of the seven drivers were able to run without the traces). I'm not sure this is an advantage. First, traces have to be obtained using a physical device each time the code of a driver or a test workload changes - doesn't this mean one basically still needs the real hw to some extent when using SymDrive?

Second, the ability to use traces (for both fast-forwarding and equivalence checking) depends on all hw interactions being in exactly the same sequence on all executions, but I'd expect the interactions to depend on other devices in the system or even on the exact timing (e.g., a polling function called from a timer interrupt could be executed either before or after some other function that also accesses the device). This

seems to be a strict disadvantage over RWSet and DDT-style symbolic hw, with which you could test similar drivers without relying on traces or any other information about the behavior of the hardware. While this kind of symbolic hw introduces non-spec-compliant hw behaviors, the SymDrive authors argued in SOSP '09 that real hw doesn't obey its spec anyway.

Comparing hw traces to establish equivalence is neither sound (as certain changes could be allowed if the resulting behavior is equivalent according to the device specification) nor complete (as bugs might be introduced without changing the hardware interactions, e.g., by returning wrong values to the user). Furthermore, this also is something that the DDT project claims to be able to do, using its execution traces.

SymDrive requires modifications to the OS kernel code and the drivers' code and build process. It would be good to discuss how much this increases the cost of testing. This could also introduce differences between behaviors of the driver during testing vs. running in production. So there ought to be a better case made for why one would opt for this route. I think that SymDrive might be able to capture some more sophisticated bugs, e.g., that rely on type information (since it has src code), but perhaps in device drivers such type information isn't leveraged much anyway?

I would have liked to see an evaluation of SymDrive's false positive rate, because it can neither control the state of the OS kernel nor propagate symbolic values through it, so SymDrive must ignore the kernel and over-approximate its behavior with an arbitrary one. I would expect that forking the VM could avoid this problem, but that's not what SymDrive does.

In hardware verification we often find that the set of "good" hw behaviors is a small subset of its possible behaviors (e.g., there are many ways in which an operation can fail, but only one in which it can succeed). This means that, for the "concrete+symbolic" strategy (Section 2.2.2) to be feasible, one absolutely requires the presence of a hardware interaction trace, or else the first explored path is most likely going to hit error-handling code very quickly and not get any further.

I disagree that exhaustive exploration in the presence of reentrancy requires inserting a possible preemption point after each instruction - in symbolic execution, preemptions on most of these points would actually produce entirely equivalent executions.

Section 2.1 says that "when symbolic values are compared, SymDrive forks execution"; this seems unnecessary, as the forking only needs to be done when a symbolic value is part of a branch predicate (i.e., a statement like "y = (x > 5)" with x symbolic should not cause any forking, just an update of y's constraints). Or maybe this statement is just a typo?

SUITABLE FOR SHORT PAPER (?)

1. Not suitable

Review #10B

Modified Monday 28 Feb 2011 9:53:39pm PST

Plain text

OVERALL MERIT (?)

3. Weak accept (acceptable, but some flaws)

REVIEWER CONFIDENCE (?)

3. Comfortable reviewing paper (close to my area)

NOVELTY (?)

3. At least one substantial new contribution (typical ATC paper)

TECHNICAL QUALITY (?)

3. About the technical depth you'd expect (typical ATC paper)

PRESENTATION QUALITY (?)

4. Very good

RELEVANCE (?)

5. Excellent

BRIEF SUMMARY

The paper describes SymDrive, a system for testing device drivers. SymDrive leverages KLEE to simulate hardware inputs, thus it can test a driver without requiring the actual device. It also provides a testing framework for developers to specify checks. The authors applied SymDrive
to seven Linux PCI and USB drivers and found six unique bugs

MAIN STRENGTHS

Driver code is buggy, and this paper presents a useful system to detect driver bugs.

One check SymDrive does is interesting: refactoring patches should not change driver behaviors.

MAIN WEAKNESSES

No false positive numbers are reported---why is this crucial information left out from the results?

Previous work (DDT) has applied symbolic execution to device drivers.

Seems that five out of six bugs (bug 1, 2, 3, 4, 6) found by SymDrive can
be found by simple static analysis [14]---is the use of symbolic execution really justified?

DETAILED COMMENTS FOR AUTHORS

I enjoyed reading the paper. SymDrive seems useful. Finding driver bugs is good. The writing is also very clear. I like the idea of checking patch equivalence, an idea that has been floating around for a while, but SymDrive is one of the first few systems that implements this idea.

The number/percentage of false positives is a crucial metric for evaluating an error detection tool. Why is it not reported? SymDrive may indeed generate a large number of false positives due to the way it handles driver-to-kernel function calls. When a driver calls a kernel function, SymDrive marks the return value and all struct fields modified by the kernel symbolic (i.e., unconstrained), which may not be the case.

I'd like to see some interesting, custom properties developers can write using your testing framework, but the properties you described seem very
generic (e.g., if you do A, then you must do B). Simple static analysis can automatically infer these properties.

Similarly, how many bugs described in sec 5.2 can be found by simple static analysis? If a static analyzer can find roughly the same set of bugs as SymDrive, why bother with symbolic execution and all the engineering efforts?

I'm surprised that your check of patch equivalence (sec 5.3) finds no bugs. Engler et al found many bugs by cross checking code equivalence (coreutils, printf, etc). This check seems one of the strengthens of symbolic execution: users specify nothing except code equivalence, and symbolic execution would happily explore many paths to check it.

How many bugs you found are previously unknown? The paper says that some
were fixed in latest versions of Linux kernel. Does this mean that you knew these bugs and applied SymDrive to find them?

The check of user-invokable allocation in sec 4.5.1 is interesting. Does your analysis have to understand quota checks though? For example, the driver code may look like

```
if(allocated < user quota)
  alloc();
```

Although the allocation is user-invokable, the quota check makes it safe.

The discussion on false positives in sec 5.6 is incomplete.

SUITABLE FOR SHORT PAPER (?)

3. Suitable

Review #10C

Modified Monday 28 Feb 2011 4:55:40pm PST

Plain text

OVERALL MERIT (?)

2. Weak reject (not quite an ATC paper, but almost)

REVIEWER CONFIDENCE (?)

3. Comfortable reviewing paper (close to my area)

NOVELTY (?)

3. At least one substantial new

TECHNICAL QUALITY (?)

3. About the technical depth you'd

contribution (typical ATC paper)

expect (typical ATC paper)

PRESENTATION QUALITY (?)

3. Good

RELEVANCE (?)

3. Good

BRIEF SUMMARY

combine existing ideas (KLEE symbolic execution and SLAM style device driver checkers) to check device drivers.

MAIN STRENGTHS

works on real code, found real bugs, important problem.

MAIN WEAKNESSES

few bugs, few drivers, no measurement of how thorough the testing was (e.g., not even statement coverage).

DETAILED COMMENTS FOR AUTHORS

device drivers control their devices by reading and writing memory. previous work by cadar et al (which they should probably cite: RWSet analysis, in ISSTA) showed you could run devices at user level by making this device memory symbolic. the gamble you are making is that real hardware may only return a limited number of values (say, 5 and 6) while unconstrained symbolic memory can return any value (all integer values expressible with the given number of bits), possibly giving false positives. in my opinion, it's interesting if the gamble works for real devices, so found the paper's results cool, though (IMHO) too light.

swift usually has experimentally thorough papers, so i was disappointed at the meager evaluation in this one. you'd like to see many more devices
since the open question w/ symbolic execution is whether it can handle real code or instead chokes on some NP issue.

also, the point of symbdriver is to check drivers thoroughly, but this paper does not demonstrate that it can do so --- there is not even any coverage results, which is sort of the bare minimum you'd like. can it hit all statements? only 20%?

sec 3.1: their work looking through "2,000" linux messages seems useful --
it'd be nice if they made these results available.

4.3: they seem to be claiming an innovation in state space searching by stating that by default klee uses DFS. it does, but since DFS is retarded (though simple) it has a bunch of other search heuristics controlled by command flags. IMHO switching b/n DFS and BFS isn't particularly novel --- and in any case if you do search tricks you have to show they work (e.g., coverage, or something similar).

they make a point about ptrintk allowing users to do DoS --- will

linux hackers actually fix such "bugs"? i kind of doubt it, given the broad array of DoS attacks users have at their disposal.

they state that symbolic execution can find more bugs with fewer false positives than static. while we can stipulate that this is true in theory, their results do not convincingly show it is true here --- e.g., the bugs they discuss seem largely within the realm of static. further, their bug counts are miniscule comparedto static since they were only able to
check a few drivers (if they could easily check more i assume they would have).

they state: are not aware of any specific false negatives in among the checks that symdrive supports --- klee has a broad range of false negatives,
which they therefore share. probably good to discuss these.

SUITABLE FOR SHORT PAPER (?)

3. Suitable

Review #10D

Modified Sunday 6 Mar 2011 12:38:20pm PST

Plain text

OVERALL MERIT (?)

3. Weak accept (acceptable, but some flaws)

REVIEWER CONFIDENCE (?)

2. Somewhat comfortable reviewing (but not my area)

NOVELTY (?)

3. At least one substantial new contribution (typical ATC paper)

TECHNICAL QUALITY (?)

4. Very well done and quite deep (good ATC paper)

PRESENTATION QUALITY (?)

4. Very good

RELEVANCE (?)

4. Very good

BRIEF SUMMARY

The paper describes a system for PCI and USB driver testing and validation through symbolic execution. The system can achieve achieve 100% coverage even without presence of the test hardware.

MAIN STRENGTHS

The paper addresses a very important issue -- increasing the quality of device drivers, and covering error paths, which are known to be frequently mishandled. Its main contribution is getting rid of the need to actually have hardware for the drivers, which can subtantially increase the number of people able to do driver work. Overall, their approach appears to work.

MAIN WEAKNESSES

Some issues are not thoroughly studied: the amount of false-positives,

and the coverage of known bugs in existing drivers. Since the authors claim they address the most urgent problems in drivers development, some sort of quantification should have been presented. Moreover, it is not clear kernel stability can be guaranteed without significant effort in annotation and checkers, which may match the effort required for formal specification.

DETAILED COMMENTS FOR AUTHORS

The main issue is that the system is not throughly evaluated, and the number of found bugs is not compared to the known bugs (even if the system only covers part of them, it is claimed to be a substantial part). In other words, how useful is this in practice? What percent of bugs does it find, and how many of those are *important* bugs?

The amount of false-positives is not shown and the quantity of required annotations is not mentioned in the paper.

The authors mention they abort the driver after 10 minutes (Section 5.1). in this case, how is 100% coverage ensured?

The effort required for writing checkers that cover a substantial number of bugs seem high. Without data regarding the coverage of the existing checkers, I suspect this is the case.

Can the existing Linux correctness checkers (e.g., the lock validation code) be augmented via symbolic execution?

SUITABLE FOR SHORT PAPER (?)

2. Can't tell