“Notes” – extra comments not in paper

* Titles
  + SymDrive: Validating and Verifying Driver Code without a Device
  + SymDrive: Who needs a Device to Test a Driver?
  + SymDrive: Removing the Hardware Requirement in Driver Testing
  + SymDrive: Finding and Preventing Bugs in Drivers without Devices
* Potential test framework (checkers excluded) names
  + SymRuntime
  + SymLib

Outline

* Intro
  + Symbolic Execution allows executing code that could not otherwise execute, not primarily for coverage
    - want to execute the code of interest, rather than all code
* Background and open problems
  + Overall goals:
    - Useful – finds bugs hard to find via other mechanisms
      * what do other tools lack
    - Simplicity – new devices/buses/operating systems/kernel versions
    - Efficiency – fairly quick to test
  + Symbolic execution is promising
    - What is symbolic execution?
    - Why useful?
      * no hardware
      * better coverage
    - Why is it not used today
      * Information – the instrumented driver supplies runtime information to the path selection algorithm.
      * Specification
        + (answer is use kernel + hardware + checkers for unchecked kernel issues + liveness)
* Design
  + Goals ½-2/3 column
    - Any motivated kernel programmer can run the tool to find bugs or to test a patch.
    - Bug finding: Broad. Symdrive must be able to test many drivers with minimum effort.
    - Patch testing: Deep. SymDrive must be able to test arbitrary driver functions thoroughly with minimum effort.
  + Overview ½-2/3 column
    - Outline four components: VM environment, Symbolic Hardware, SymGen, Checkers, Test Framework. This section should be shorter than the existing design section.
    - Rationalize using symbolic data to test driver
  + Virtual Machine –1/2 column – use S2E + extend generality
    - Notes – the VMM implementation includes S2E, I/O memory and Port I/O support + tracing, path selection, and opcodes. We can probably defer path selection discussion to the section on the TF.
    - Opcodes
      * Inform VMM of the driver’s state.
  + Symbolic devices 1 column
    - semantics of i/o memory, io ports, interrupts,
    - device discovery/creation
    - Notes – we should put this section before SymGen since SymGen seems to depend on the symbolic hardware for it to make much sense
    - Driver provides model of device- knows what relevant parts of device state change
      * Generate the model as execution proceeds through the driver, but ignore anything that the driver also ignores. Relationship with H/W dependence bugs.
  + Test Framework (renaming seems like good idea)
    - Notes – this section should tie it all together. With the instrumentation, checkers, and VM we have the capacity to do broad and deep testing. “Test framework” means only the part that controls prioritizing/deprioritizing and informing the VMM of the driver’s state.
    - Broad testing: many drivers
      * Favor-success scheduling: fast forward to relevant parts.
      * Loop elision
    - Deep testing: thorough coverage of specific functions
      * High coverage mode for patch testing
      * Entry point testing, e.g. ethtool, ioctl
        + Notes – this seems similar to a checker but it fits well here.
        + Notes – mention that drivers are like libraries with many entry points and lots of interaction between them, especially interrupt handlers
    - Limitations
  + SymGen
    - Motivation: information, interposition
    - Tool
      * CIL module
      * Most does code generation – no sophisticated static analysis
    - Generated code
      * Stubs, which include code to:
        + Callouts to the checkers
        + Callouts to the test framework
      * Loop instrumentation
        + break/continue/goto handled
        + instrument before and after loop, and end of loop body
      * Driver entrance/exit instrumentation
        + only one return statement per function, goto’s handled etc
  + Checkers
    - Library API in the TF
    - Code for finding checkers
    - Examples:
      * Execution context
      * Kernel API misuse
      * Collateral evolutions
      * Memory leaks – we track actual pointer values which provides strong guarantees, in contrast to SDV
      * Execution Differencing: recording I/O memory operations etc. This is not a checker per-se but it functions similarly in that it interposes on driver operations.
* Evaluation
  + Methodology
  + Broad Testing
    - Bugs Found
    - Bug validation (perhaps some other word?)
    - Whole-driver Coverage
    - Developer effort
      * Using SymDrive
      * Brief section on TF usage load module, specify any interesting functions etc.
  + Deep Testing
    - Patch Testing
      * Coverage of patched functions
    - Execution differencing
      * Linux/FreeBSD
* Related Work
  + DDT and S2E
  + Symbolic testing
  + Static analysis tools
  + Test frameworks
  + Formal specifications for drivers
* Conclusion