Symbolic Execution

- Goal: automating the search for security vulnerabilities / finding bugs
    - We assume that bugs ~ exploit
    - It might be very hard to use this bug, but we assume it is possible
- Protocol for disclosing exploits:
    - Contact vendor
    - After some time (i.e. 90 days) disclose to public
    - CVE: Common Vulnerabilities and Exposures
        - Community effort to record bugs
- Many companies have bounty programs

Testing

- Application has inputs which the attacker can provide (i.e. web requests)
- We add checks for bad situations that might arise
- Paper provides method to find inputs the attacker could provide that would cause the code to fail the checks
- Types of bugs:
- Divide by zero, null-pointer dereference, out-of-bounds array access
- Application-specific bugs (through asserts)
    - Bank transaction changed sum of other accounts
    - X's transaction decreased Y's balance
    - Program opened a file with name determined b yinput
    - Request sets cookie to impersonate another user
- We only turn these asserts / checks on during testing
- Alternative:
- Directly prove that a program is correct
    - A lot of work
    - Not practical for big programs

- You could write a suite of tests
- Good for:
    - Intended functionality
    - Known bugs
- Bad for:
    - Unintended functionality (vulnerabilities)
    - Unknown bugs
    - Hard to write for complete coverage
    - Takes effort
- We wish to automate test-case generation / search for as-yet-unknown bugs that achieve good coverage

Fuzzers

- Execute program on lots of randomly-generated inputs
    - Find input sources (CLI / HTTP requests)
    - Write input generation code
        - Should be smart to generate syntatically correct input and only put random content where freedom is allowed
- Continue to execute until you get bored / see if random inputs trigger an assert
- Effectiveness:
    - Widely used and have found lots of bugs
        - Particularly good at buffer overflow that do not need specific input to trigger them

        - Many out-of-bound values are likely to generate a crash
    - Doesn't need source code
    - Better then programmer at testing for unexpected behavior
- Problems:
    - May use lots of CPU time
    - Hard to cover everything
        - Hard to test with random input

Symbolic Execution

- A more sophisticated testing scheme (i.e. EXE)
- Goal: find deep bugs by driving program along all paths in program
- Ideas:
    - Compute on symbolic values
    - Branch on each if statement
    - Use constraint solver to see if branch is possible
        - SMT queries (satisfiability modulo theories
        - STP: simple theorem prover: constraint solver

EXE

- At compile time:
    - C input is passed into a translator that instruments the ifs, assignments, and expressions
    - Also adds a table that goes from each memory range to a symbolic value
- At runtime:
    - Runs application, which reaches a path condition
    - Constraint solver checks whether this path condition is satisfiable
        - If it is, we fork to explore it
        - Otherwise, we don't explore
    - Scheduler process selects which application process to explore
- Each variable / memory holds an expression in terms of inputs
    - They hold symbolic values, not concrete values since they are dependent on inputs
    - EXE remembers which memory locations hold symbolic values and what each location's current symbolic value is
    - Executed if statements imposes constraints known as "path constraints"
- When EXE gets to an error call, it checks whether the current path constraints can be satisfied
    - If possible, EXE reports an assert failure with the inputs
- How to use:
    - User marks which memory ranges are symbolic
    - Whenever we have an assignment that uses at least one symbolic type, the result becomes symbolic as well with some constraint now in place
    - Symbolic types just stored as bytes / bits so it is independent of what the C representation is
- Each time we have a branch, the process forks and then waits for the search server to tell it to continue
    - The search server uses the best-first heuristic to determine which process to send out:
        - Tries to execute the line of code that's been run the fewest timese
        - Runs on DFS on that process and children "for a little while"
    - Infeasible to run through every possible branch
    - Loops are handled as just branches, then each process spawned will hit the branch again later (but with more constraints now so this should terminate eventually hopefully)
- Cannot handle floating point / interaction with the OS (e.g. calling open)

Constraint Solver

- Solves setes of equations

- Limitations:
  - Treats many C constructs as arrays (i.e. strings / ptrs / structs)
  - If we are indexing into them with a constant index, this is generally fine since we can just treat it as a specific symbolic value
  - When our index is symbolic, we have a big disjunction
  - If we are dereferencing a pointer that is symbolic, this could be literally anywhere
- If the solver times out, we effectively treat this branch as unsatisfiable
- This is very slow, so optimizations are critical
  - Prune if branches if no solution
  - Cache + share constraint solutions / fragments
  - Try as much as possible to use your concrete operations / operand values
- Takes tens of minutes for small UNIX utility code programs