Symbolic Execution

CS390R - UMass Amherst

Course Information

- Project 6 assigned
- Presentation Recordings due 05/11 @ 2:30 pm
- Presentation Writeups due 05/20

Today's Content

- Introduction to Symbolic Execution
- Constraint Solving using SAT/SMT
- Z3 Demos
- Engine Types
- Challenges that come with Symbolic Execution
- Types of Symbolic Execution
- Practical Symbolic Execution using Triton analysis framework

Introduction

- Symbolic execution was first introduced mid 70’s to test program properties
- Main idea is to transform the program into a mathematical equation that can then be solved for various properties using a sat solver
- Work with symbolic data instead of concrete data
- Instructions are simulated using a symbolic execution engine that maintains a boolean formula describing the satisfied conditions for each path
- This formula can then be solved for satisfiability using a constraint solver

```c
int f(int x, int y) {
    char buf[0x8];

    if (x != 0) {
        if (y > 2) {
            gets(buf);
        } else {
            exit(1);
        }
    } else {
        puts("Invalid");
    }
}
```

```c
int f(int x, int y) {
    char buf[0x8];

    if (x != 0) {
                                              // 1. (x != 0)
        if (y > 2) {
            gets(buf);
        } else {
            exit(1);
        }
    } else {
                                              // 1. (x == 0)
        puts("Invalid");
    }
}
```

```c
int f(int x, int y) {
    char buf[0x8];

    if (x != 0) {
                                        // 1. (x != 0)
        if (y > 2) {
                                        // 2. ((x != 0) && (y > 2))
            gets(buf);
        } else {
                                        // 2. ((x!= 0) && (y <= 2))
            exit(1);
        }
    } else {
                                        // 1. (x == 0)
        puts("Invalid");
    }
}
```

Constraint Solving using SAT/SMT

- SAT
  - First problem that was proven to be NP-complete
  - Boolean Satisfiability/Propositional Logic
  - Operates solely on boolean formulas to find solutions to statements (Kinda like 250 truth tables)
  - eg: What boolean values for p and q satisfy the following statement: (p and !q)
- SMT
  - Depending on problem either NP-complete or undecidable
  - Predicate Logic
  - Supports same features as SAT, but can also deal with more complex structures such as integers/arrays or arithmetic operations
  - eg: Find real numbers x & y such that (x + y * 3 == 4 - x)

Z3 Demo

Types of Symbolic Execution

- **Static**:
  - Simulate execution based on source code
  - Benefit that it isn't reliant on compatible architecture
  - Very hard to reason about kernels/libraries not included with source code or tried to read environment variables
- **Dynamic/Concolic**:
  - Combine concrete state with symbolic state.
  - This approach is based on using concrete state to execute the target and maintaining symbolic information as metadata similar to taint data.
  - Exploring various paths is then done by flipping path constraints that operate on data marked as symbolic.
  - In general this scales better than purely static solutions since you don't need to maintain multiple parallel execution states

Engine Types

- Full System
  - s2e - Operates on entire vm's by integrating with hypervisors, thus allowing you to track state across system boundaries like kernels. This makes it very powerful but also hard to use
- Binary
  - Triton - Dynamic Symbolic Execution based on Pin/Taint Analysis (This is what we use)
  - Angr - Dynamic Symbolic execution based on VEX-IR
- Source Code
  - KLEE - Based on llvm, has many advantages that come with source such as an easier time distinguishing variable/pointer relations

Challenges of Symbolic Execution /1

- Memory
  - In the Z3 examples we saw integers/booleans being modelled without issues, but what about more complex data structures in programs like pointers?
  - Should all of memory me symbolized?
- Environment
  - How are library calls/syscalls handled?
  - Letting memory enter the kernel where a user-level symbex engine can’t keep track of it can mess up all of the state tracking
  - Symbolically executing massive library functions massively hurts performance by traversing complex code that is potentially rather uninteresting to our goals
  - Could attempt to just emulate syscalls in the engine and model all possible results, but eg. file system accesses can have a massive amount of possible outcomes

Challenges of Symbolic Execution /2

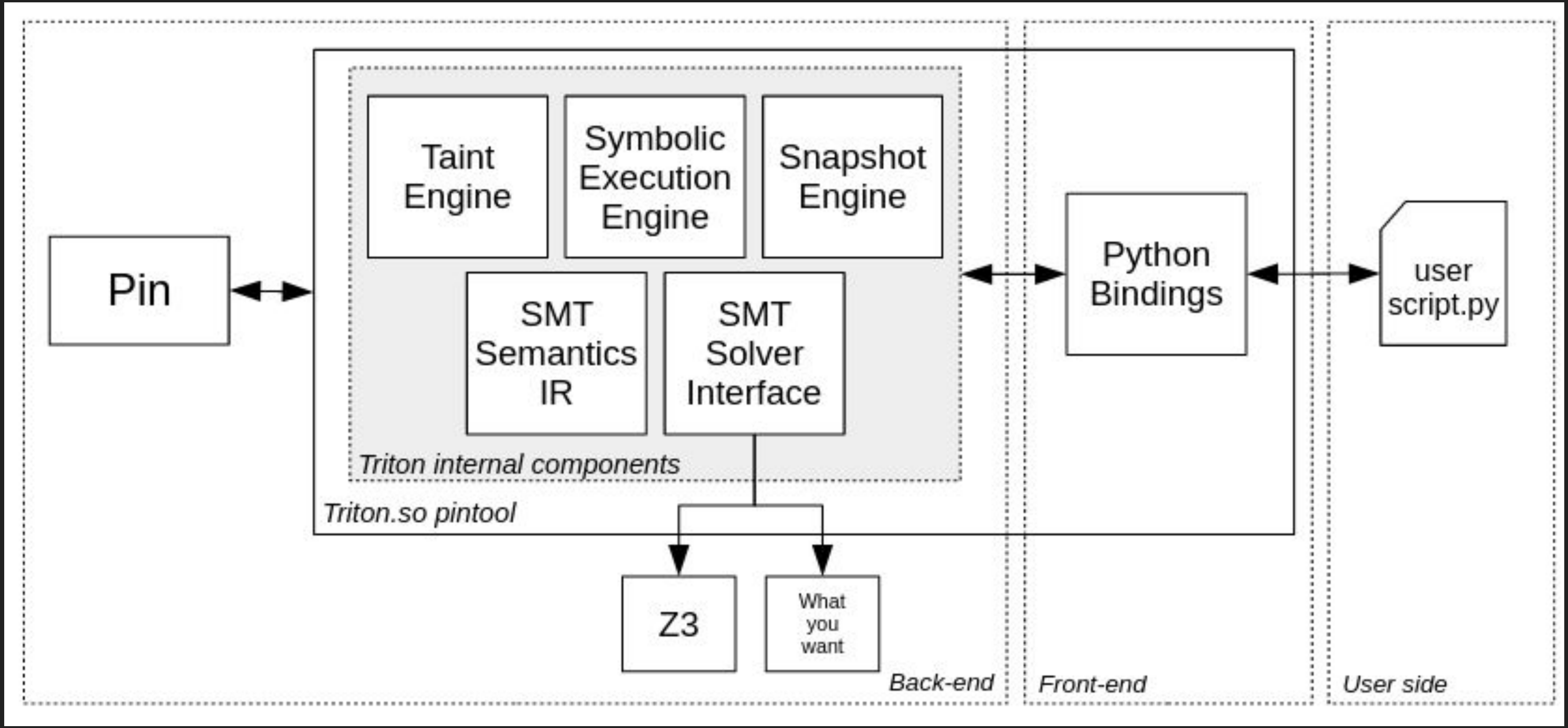
- State/Path Explosion
    - Constructs like loops/nested conditionals exponentially increases execution state
        - Each loop iteration can be viewed as an if-else branch, thus requiring state to be forked
        - In practice many engines resolve this by putting a cap on how often a loop will be modeled
    - This includes both memory & execution paths that have to be taken
    - This means that an exhaustive investigation of every possible trace is not possible on real world applications
    - Heuristics are often used to determine with paths to follow (eg. DFS/BFS)
- Constraint Solving
    - This is an NP-Complete/Undecidable problem
    - Once the formula gets large enough/contains enough hard to solve constraints this can massively impact performance thus slowing down progress

Symbolic Execution + Fuzzing

- Chose specific points when to invoke Symbolic Execution to find new paths
  - Eg. After running out of new coverage for a certain timeframe
- Angr + AFL = Driller
  - Very promising approach in theory, in reality fuzzing alone found 3x more bugs
  - Often as the path gets too long, symbolic execution can no longer keep up to find more coverage

Reasons to use Triton

- Written in C++ with its concolic mode being based on Pin so it integrates well with what we have been learning thus far
- Very easy to integrate with other tools/frameworks
- Completely free/open source and under active development
- Popular enough so that there's plenty of usage-examples out there
- Since it's written in C++ it has performance advantages in certain areas and provides users more fine grained control over execution at time's
- Good documentation
- Very customizable/provides a lot of additional functionality in terms of eg. exchanging the constraint solvers or using taint analysis


Pin
Taint Engine
Symbolic Execution Engine
Snapshot Engine
SMT Semantics IR
SMT Solver Interface
Triton internal components
Triton.so pintool
Z3
What you want
Python Bindings
user script.py
Back-end
Front-end
User side

What can you do with Triton?

- Symbolic execution
- Fuzzing/code coverage
- Scriptable debugging
- Access to Pin through higher level languages
- Converting binary code to llvm-ir and back
- Taint Analysis
- …

(List taken from https://blog.quarkslab.com/triton-under-the-hood.html)

Internals

- Every instruction is modelled by its AST and remains in SSA form
    - The AST nodes are maintained in the SMT2-LIB language so it can interact with Z3
- All parts of memory have either concrete values or symbolic values
    - The more memory we initialize as symbolic the more complex state we need to track
- Analysis is done at program runtime using pin
- Supports snapshots so multiple paths can be explored simultaneously even with its concolic nature

Triton Demo