Building a Dynamic Symbolic Execution Engine for Python Bytecode Analysis using Z3 SMT Solver and AST Manipulation

---

High-Ranking SEO Keywords Used

- Advanced Python symbolic execution
- Python bytecode analyzer with Z3
- Python static analysis tool
- Formal verification with Python
- Python program analysis using Z3 SMT solver
- Python AST transformation for vulnerability detection
- MAANG-level Python code sample
- Rare and hard Python programming projects
- Python symbolic interpreter
- Z3 constraint solver with Python AST

FULL PYTHON CODE: symbolic_executor.py

"""

Advanced Symbolic Execution Engine in Python using Z3 and AST.

Level: Advanced | Rare | MAANG-level

```
"""

import dis
import ast
import inspect
import operator
from types import FunctionType
from z3 import *

# === Symbolic Value Representation ===
class SymVal:
    def __init__(self, name, value=None):
        self.sym = Int(name) if isinstance(value, int) or value is None else Real(name)
        self.concrete = value

    def __repr__(self):
        return f"Sym({self.sym}, concrete={self.concrete})"


# === Symbolic Execution State ===
class SymbolicState:
    def __init__(self):
        self.stack = []
        self.vars = {}
        self.constraints = []
```

```python
    def push(self, val):
        self.stack.append(val)

    def pop(self):
        return self.stack.pop()

    def add_constraint(self, constraint):
        self.constraints.append(constraint)

    def get_solver(self):
        solver = Solver()
        for c in self.constraints:
            solver.add(c)
        return solver

    def __repr__(self):
        return f"Vars: {self.vars}, Stack: {self.stack}, Constraints: {self.constraints}"


# === Supported Instructions ===
INSTRUCTION_HANDLERS = {}

def instruction(opname):
    def decorator(fn):
```

```
        INSTRUCTION_HANDLERS[opname] = fn
        return fn
    return decorator


# === Core Symbolic Execution Logic ===
class SymbolicExecutor:
    def __init__(self, fn: FunctionType):
        self.fn = fn
        self.code = list(dis.get_instructions(fn))
        self.state = SymbolicState()

    def run(self):
        idx = 0
        while idx < len(self.code):
            instr = self.code[idx]
            handler = INSTRUCTION_HANDLERS.get(instr.opname, None)
            if handler:
                handler(self.state, instr)
            idx += 1
        return self.state

    def check_path_feasibility(self):
        solver = self.state.get_solver()
        return solver.check(), solver.model() if solver.check() == sat else None
```

```python
# === Instruction Handlers ===

@instruction("LOAD_CONST")
def load_const(state, instr):
    val = instr.argval
    state.push(SymVal(f"const_{val}", value=val))

@instruction("LOAD_FAST")
def load_fast(state, instr):
    name = instr.argval
    val = state.vars.get(name)
    if val is None:
        val = SymVal(name)
        state.vars[name] = val
    state.push(val)

@instruction("STORE_FAST")
def store_fast(state, instr):
    val = state.pop()
    name = instr.argval
    state.vars[name] = val

@instruction("BINARY_ADD")
```

```
def binary_add(state, instr):
    b = state.pop()
    a = state.pop()
    new_val = SymVal(f"tmp_{len(state.stack)}")
    state.add_constraint(new_val.sym == a.sym + b.sym)
    state.push(new_val)


@instruction("COMPARE_OP")
def compare_op(state, instr):
    right = state.pop()
    left = state.pop()
    op = instr.argval
    result = SymVal(f"cmp_{op}_{len(state.stack)}")

    opmap = {
        '==': operator.eq,
        '!=': operator.ne,
        '<': operator.lt,
        '<=': operator.le,
        '>': operator.gt,
        '>=': operator.ge
    }

    if op in opmap:
        state.add_constraint(opmap[op](left.sym, right.sym))
```

```
    else:
        raise NotImplementedError(f"Compare operator {op} not supported.")

    state.push(result)

@instruction("RETURN_VALUE")
def return_value(state, instr):
    # For analysis purposes, we don't do anything.
    pass


# === Sample Function to Analyze ===
def sample_fn(x, y):
    a = x + y
    if a > 10:
        return 1
    else:
        return 0


# === Usage Example ===
if __name__ == "__main__":
    executor = SymbolicExecutor(sample_fn)
    final_state = executor.run()
    print("Final Symbolic State:\n", final_state)
```

```
    status, model = executor.check_path_feasibility()
    print("\nSAT Check:", status)
    if model:
        print("Model satisfying the constraints:")
        for var in model:
            print(f"{var} = {model[var]}")
```

What This Code Does:

Feature	Explanation
Symbolic Execution Engine	Tracks symbolic values of variables instead of executing with real inputs.
Z3 SMT Solver Integration	Collects path constraints and uses Z3 to determine feasibility of execution paths.
AST + Dis + Bytecode Analysis	Operates at bytecode level using dis for rare and powerful analysis.
Constraint Generation	Captures expressions like x + y > 10 and checks if satisfiable.
Perfect for Vulnerability Analysis	This technique is used in tools like Microsoft's SAGE or Facebook's Infer.