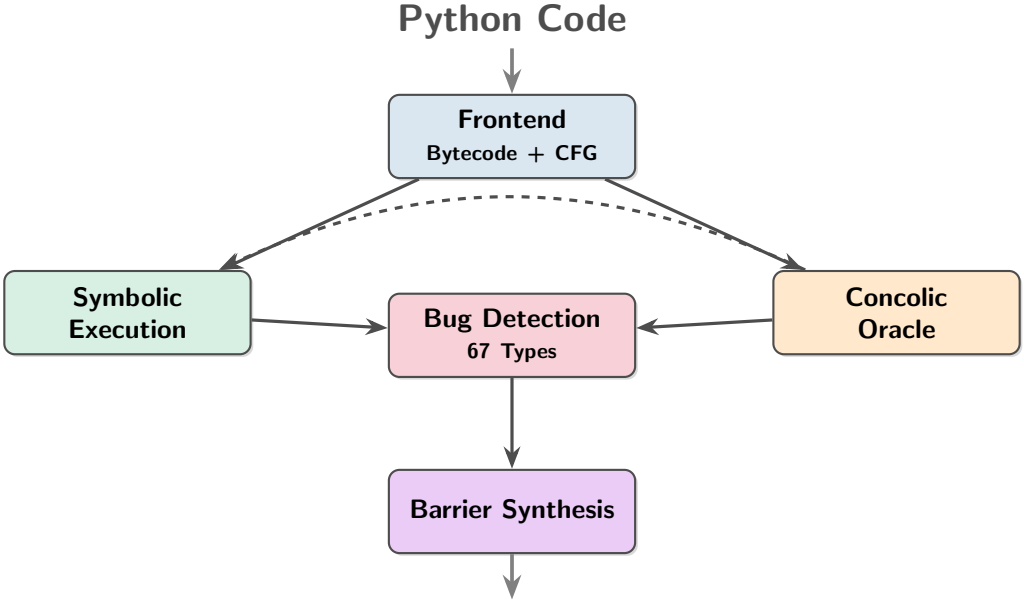
PythonFromScratch

Static Analysis with Symbolic Execution & Barrier Certificates

February 2, 2026

System Architecture Overview



Symbolic Execution & Taint Tracking

Symbolic Execution

Path Exploration:

- Z3-based constraint solving
- Path-sensitive analysis
- Loop unrolling & bounds
- Branch coverage tracking

State Management:

- Symbolic heap modeling
- Variable tracking
- Context sensitivity (k-CFA)

Taint Analysis

Taint Sources:

- User input (stdin, args)
- Network data (HTTP, sockets)
- File I/O, environment vars
- Function parameters (security mode)

Propagation:

- Intra-procedural flow
- Inter-procedural with summaries
- Container operations

Concolic Oracle: Concrete execution validates symbolic paths and refines constraints

Symbolic vs Concolic Execution: Complementary Roles

Symbolic Execution

Role: Explore *all possible* execution paths

Strengths:

- Complete path coverage (in theory)
- Discovers edge cases automatically
- Generates constraints for all branches
- No need for test inputs

Challenges:

- Path explosion in large programs
- Complex constraints (solver timeouts)
- Environmental interactions

Concolic Oracle

Role: *Validate & refine* symbolic reasoning

Strengths:

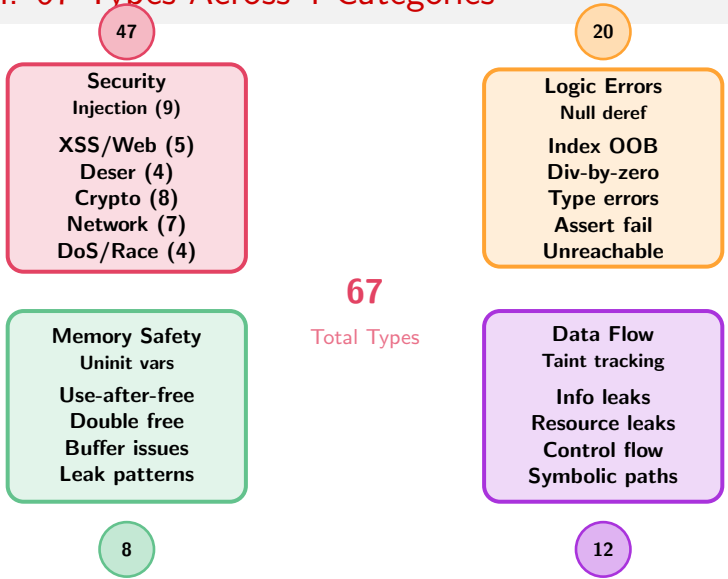
- Grounds symbolic with concrete values
- Resolves constraint ambiguities
- Handles complex operations (hashing, crypto)
- Pruning infeasible paths

Integration:

- Symbolic finds paths $\rightarrow$ Concolic validates
- Concolic provides counterexamples
- Hybrid: symbolic breadth + concrete depth

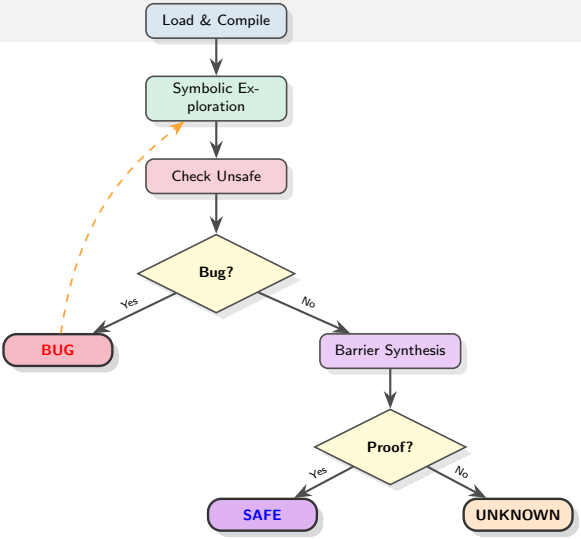
Best of Both: Symbolic exploration + Concrete validation = Robust static analysis

Bug Detection: 67 Types Across 4 Categories



Detection: Symbolic execution + Taint analysis + Barrier verification

Analysis Workflow



Input: Module • Functions • Interprocedural **Output:** Trace or Certificate

Barrier Synthesis: Core Concept

What is a Barrier Certificate?

A **barrier function** $B(x)$ separates reachable from unsafe states:

- $B(x) \geq 0$ for initial; $B(x) < 0$ for unsafe
- $B(x)$ non-negative along executions (inductive)

If such B exists, program is provably **SAFE**.

Synthesis Challenge

Goal: Find $B(x)$ automatically
4 Approaches (20 Papers):

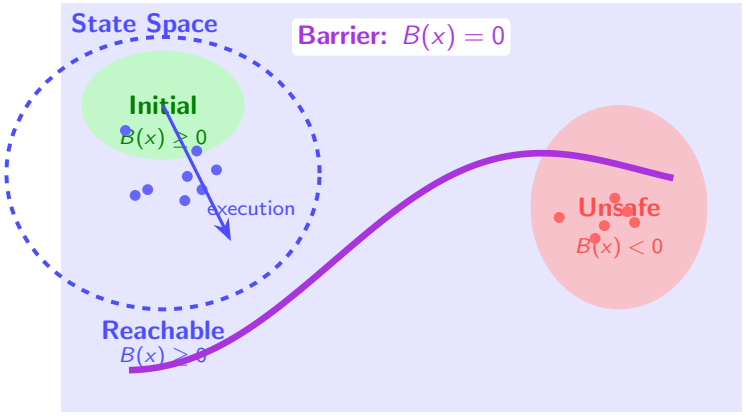
- **Direct:** Polynomial + SOS/SDP
- **Abstraction:** Simplify, refine
- **Learning:** From examples
- **Symbolic:** IC3/PDR, CHC

5-Layer Integration

L1-2: Math foundations + certificate types
L3: CEGAR abstracts $\rightarrow$ feeds L1-2
L4: Learning generates templates $\rightarrow$ L1-2
L5: IC3/CHC provides alternatives

All layers **synthesize or validate barrier functions** that prove safety

Barrier Theory: Visual Explanation



Key Insight: The barrier $B(x)$ acts as a *mathematical fence*
If no execution can cross from $B(x) \geq 0$ to $B(x) < 0$, the system is **safe**

Barrier Synthesis & Verification

5-Layer SOTA Architecture

Layer 1: Mathematical Foundations

- Positivstellensatz, SOS/SDP, Lasserre hierarchy

Layer 2: Certificate Types

- Hybrid, stochastic, exponential & logarithmic barriers

Layer 3: Abstraction & Refinement

- CEGAR, Predicate abstraction (IMPACT), Boolean programs

Layers build progressively: L1-2 provide core math, L3 adds abstraction

Barrier Synthesis: Advanced Techniques

Layers 4-5: Learning & Verification

Layer 4: Learning-Based Synthesis

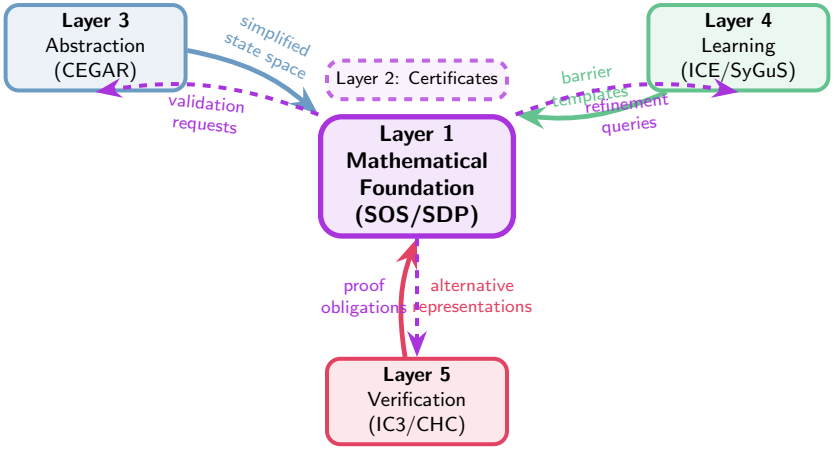
- ICE learning, Houdini inference
- SyGuS for template instantiation
- Data-driven candidate generation

Layer 5: Advanced Verification

- IC3/PDR (Incremental Inductive Clauses)
- CHC solving with Spacer
- DSOS/SDSOS for scalability
- Assume-Guarantee reasoning

Feedback Loop: L3 simplifies $\rightarrow$ L4 suggests templates $\rightarrow$ L1-2 validates $\rightarrow$ L5 provides fallback

Layer Feedback Architecture



Iterative refinement: Each layer enhances L1's barrier synthesis capability

Case Study: DeepSpeed Analysis

Target: Microsoft DeepSpeed

Production deep learning optimization library
Scale: 700 Python files, 6,208 functions, 300K+ LOC

Initial Results

Raw Analysis:

- 16,049 total bugs found
- 989 HIGH severity ($\geq$0.8 conf)
- 38 seconds, 18 files/sec

Problem:

- 82% false positive rate
- 75% duplicates
- Test file noise

After FP Filtering

Improved Analysis:

- 1,553 unique bugs (dedup)
- 31 HIGH severity bugs
- **96.9% reduction** workload
- Same 20-second runtime

Key Improvements:

- Deduplicate reports
- Downgrade test files
- Recognize safe patterns

Manual verification: Sampled bugs confirm high TP rate in final 31

Real Bugs Found in DeepSpeed

Verified True Positives

1. Division by Zero (`runtime/utils.py:partition_uniform`)

```
def partition_uniform(num_items, num_parts):
    parts = [0] * (num_parts + 1)
    if num_items <= num_parts:
        ...
    chunksize = num_items // num_parts  # No check for num_parts==0!
```

Bug: Can divide by zero if `num_parts=0`
Confidence: 0.90 (HIGH severity)

Other HIGH Severity

- `partition_parameters.py`
  BOUNDS in gradient reduction
- `stage_1_and_2.py`
  BOUNDS in checkpoint restore
- `data_analyzer.py`
  BOUNDS in map operation

Impact

- All in production runtime code
- Core optimizer & checkpoint logic
- Could cause training failures
- High confidence (0.90-0.95)