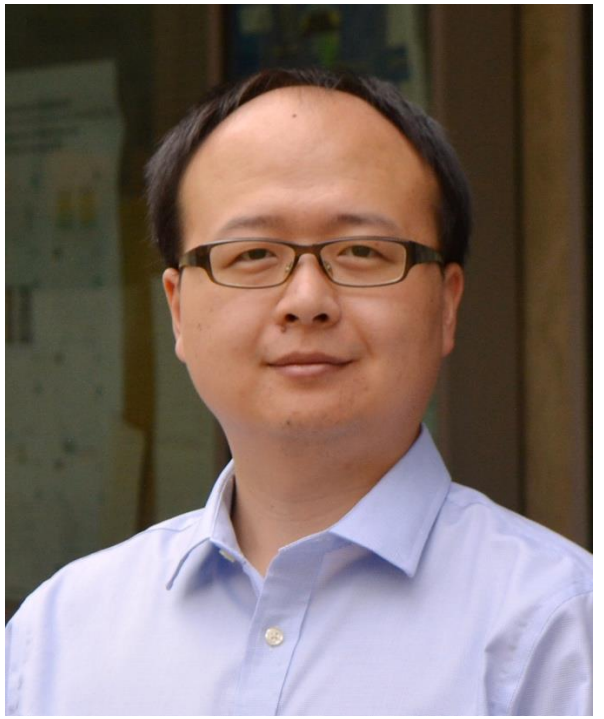
About Me



- Chengyu Song, Associate Professor, CSE
- PhD: 2016 from Georgia Tech

- Research Interests
  - Software Security: dynamic analysis, static analysis
  - System Security: compartmentalization (with new HW)
  - Hardware Security: HW-SW co-design
  - ML Security: foundation models, robotics

- NSF, DARPA, ONR, ARL, Google

UC RIVERSIDE

Software Security Research

About Vulnerabilities

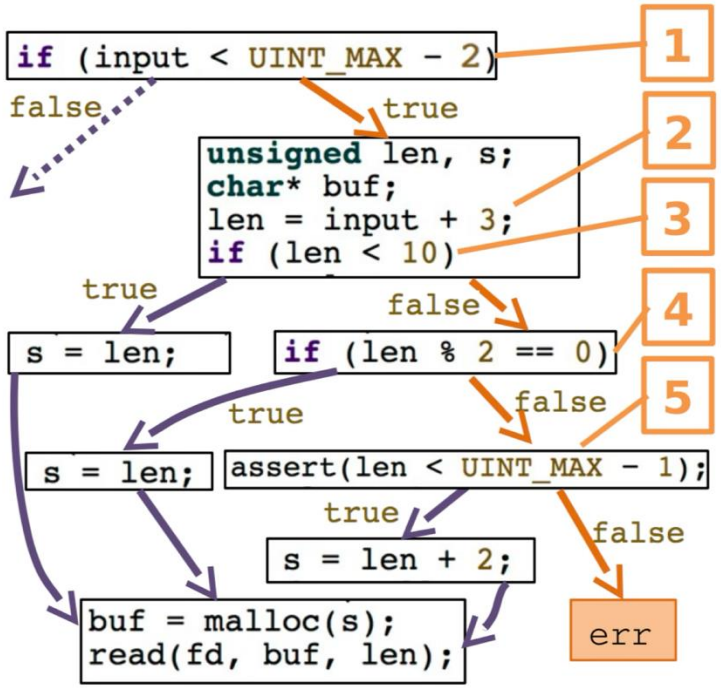
- Automated vulnerability detection via **scalable static analysis**
  - Memory errors, logic bugs
- Automated vulnerability detection via **efficient dynamic analysis**
  - Fuzzing, concolic execution, directed fuzzing
- Automated vulnerability elimination
  - Patch generation, compiler-based bug removal
- Runtime exploit prevention
  - Control-flow integrity, data-flow integrity, type checks
- Automated exploit generation

Scalable Concolic Execution

Finding vulnerabilities with concolic execution

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else {
      assert(len < UINT_MAX - 1);
      s = len + 2;
    }
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)
5	&&	!(len < UINT_MAX - 1)

Is the path predicates satisfiable?

Yes! When `input == UINT_MAX - 3`

UC RIVERSIDE

Scalable Concolic Execution

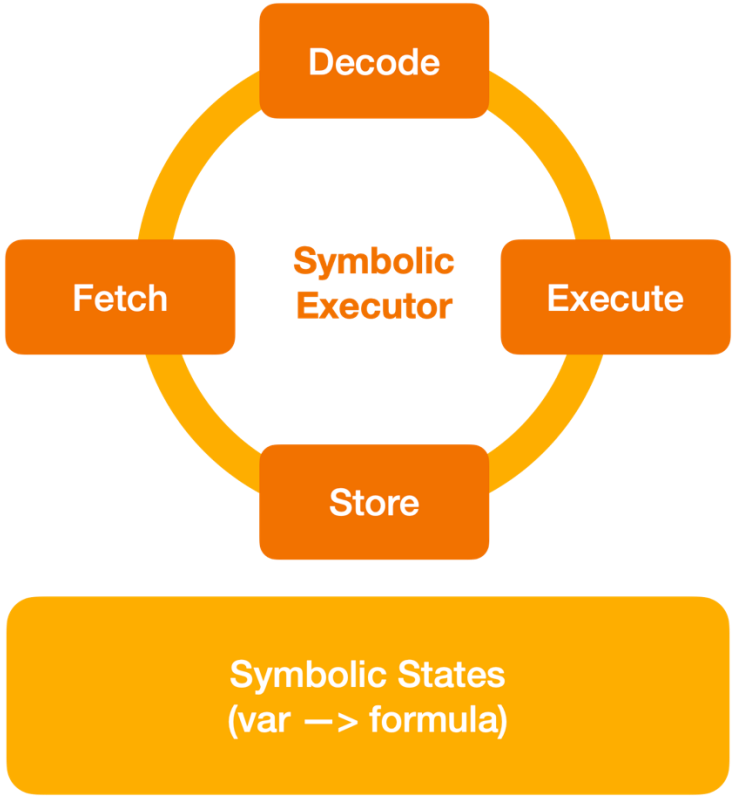
SymSan: time and space efficient concolic execution

INTERNALS OF A SYMBOLIC EXECUTION ENGINE

- Fetch: get the next instruction (LLVM IR)
- Decode: find out how to construct the symbolic formula
  – What operation, how many operands, where to find operands
- Execute: construct a new symbolic formula (AST)
  – Load formulas for symbolic operand(s), allocate a new formula
- Store: associate the new formula with the program variable

COMPARISONS

- KLEE: interpretation-based, slow runtime
- SymSan: compilation-based, more efficient runtime



UC RIVERSIDE

Scalable Concolic Execution

JIGSAW: efficient and scalable path constraint fuzzing

PATH CONSTRAINTS SOLVING

- Symbolic executors use path constraints solvers to check the feasibility of execution paths and explore different paths.
- Commonly used solvers are Satisfiability Modulo Theories (SMT) solvers like Z3.
- Path constraints solving is another major performance bottleneck.

IMPROVING PATH CONSTRAINTS SOLVING

- Constraints solving as a local search problem: finding an input assignment that would evaluate the constraints to true
  - Accuracy
  - **Throughput**

$$\frac{\text{constraints}}{\text{\#inputs}} \times \frac{\text{\#inputs}}{\text{second}} = \frac{\text{constraints}}{\text{second}}$$

Accuracy: how many inputs are expected to try

Throughput: how many inputs can be tried per unit time

UC RIVERSIDE

Scalable Concolic Execution

https://github.com/R-Fuzz

Performance improvements over previous state-of-the-art

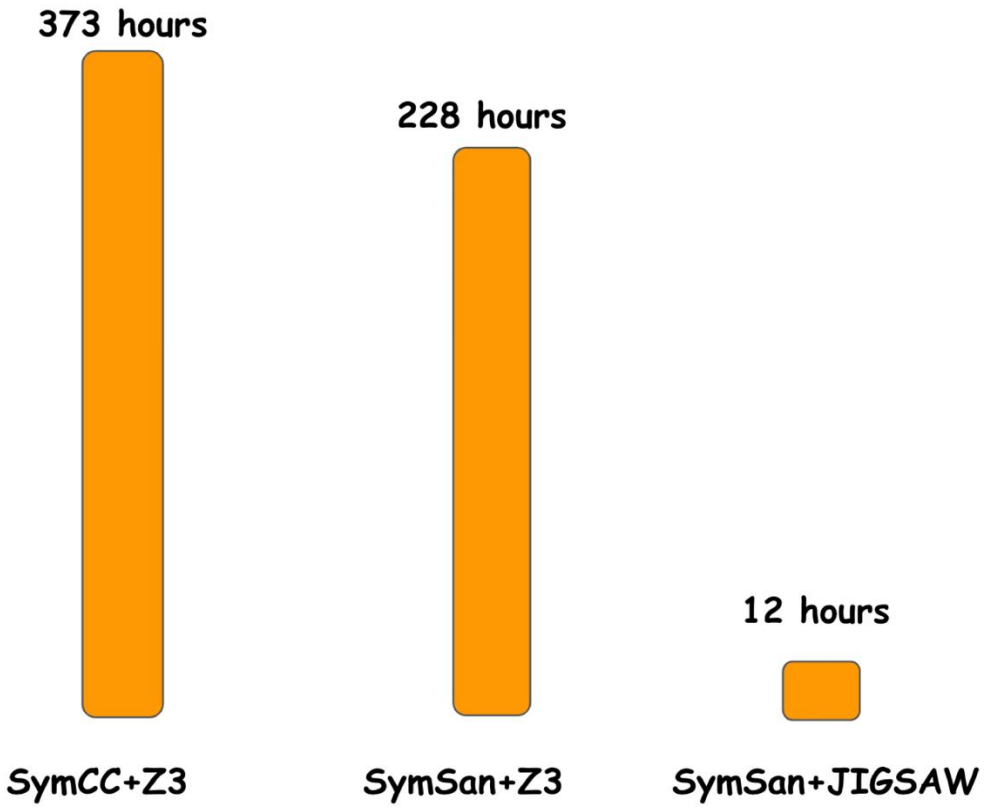


TABLE IX: Comparison of concolic execution engines on flipping all symbolic branches along a single execution trace. The top half shows the execution time, the bottom half shows the basic-block coverage measured by SanitizerCoverage.

Programs	JIGSAW	Z3-10s	Z3-50ms	Angora	SymCC	Fuzzolic
readelf	2.2h	51.3h	12.6h	89.5h	546.6h	48.2h
objdump	12.3h	227.5h	29.6h	411.5h	373.5h	52.2h
nm	0.3h	18.1h	3.2h	72.3h	29.3h	48.2h
size	0.1h	8.4h	1.4h	16.8h	12.6h	5.2h
libxml2	0.2h	9.3h	3.6h	58.0h	52.3h	20.9h
readelf	7923	7957	7423	8287	6410	5843
objdump	4926	4926	4865	4846	4929	4689
nm	3347	3347	3329	3339	3122	3123
size	2453	2457	2449	2406	2229	2259
libxml2	6038	6233	6034	5952	6012	6022

UC RIVERSIDE

Scalable Concolic Execution

Reinforcement learning directed concolic execution

(a) AFLGo (b) AFL++ (c) DAFL (d) Beacon (e) MazeRunner

Systems Security Research

Compartmentalization and principle of least privilege

- Preventing memory-corruption-based privilege escalation attacks in kernel
  - Automated identification of critical metadata for kernel access control subsystems
  - Efficient integrity protection of access control metadata with hardware features
- Isolating JavaScript JIT engine
- Automated isolating vulnerable kernel modules

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

UC RIVERSIDE

Hardware Security Research

Software-Hardware co-design

- Preventing **speculative execution attacks**
  - Violation of security contracts during speculative execution
  - Communicate and enforce the contracts: SpecCFI

❖Target class-aware call

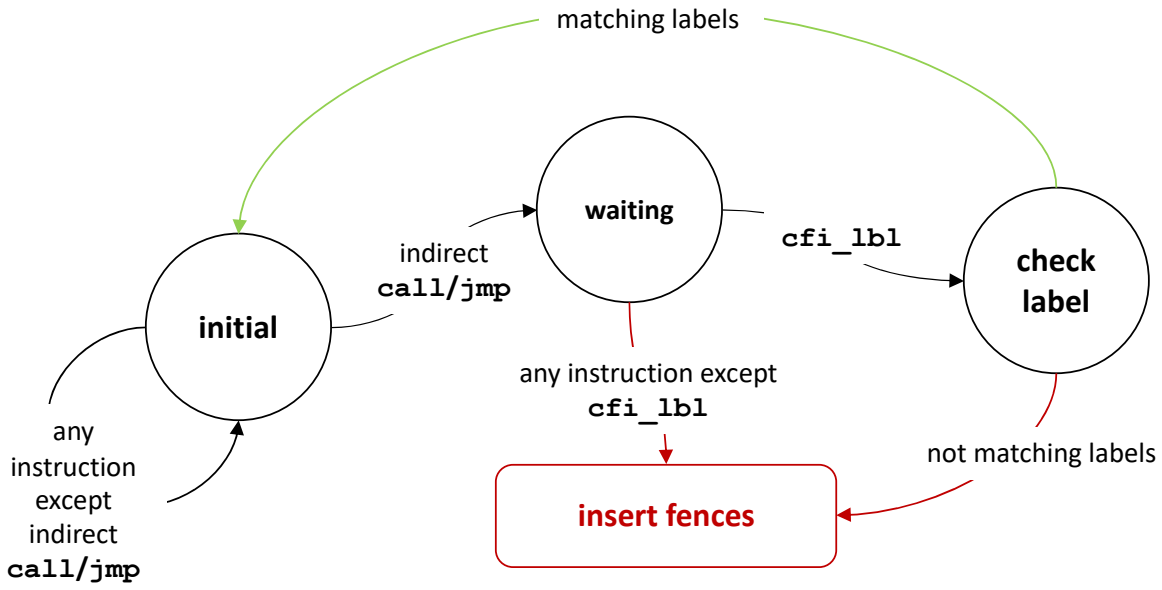
`call [dest], label`

Extending the `call` instruction to include CFI labels

❖CFI Integrity Check

`cfi_lbl`

A new instruction to mark legitimate targets with corresponding labels



UC RIVERSIDE

Hardware Security Research

Software-Hardware co-design

- Enforcing memory safety efficiently and thoroughly
  - Hardware supported memory safety enforcement
  - Respect memory safety properties during speculative executions
  - Performance vs. security?

```
// Spectre v1
int array[256];
int x, y = 0;
x = get_input();  // assume x = 65535
if (x < 256) {    // assume prediction is wrong
    y = array[x]; // array[x] is in cache!
    // cache leaks address
    z = array2[y * 4096];
}
```

Machine Learning Security Research

Adversarial attacks and defenses

- Attacks against computer vision models
  - Image classification, object detection, video classification
  - Context-base defenses and transfer attacks

- Attacks against foundation models (LLMs, VLMs)
  - Unlearning-based safety alignment
  - Structural queries
  - Robotics

Questions