StateLifter

Extracting Protocol Format as State Machine via Controlled Static Loop Analysis

Qingkai Shi, Xiangzhe Xu, Xiangyu Zhang

Purdue University, West Lafayette

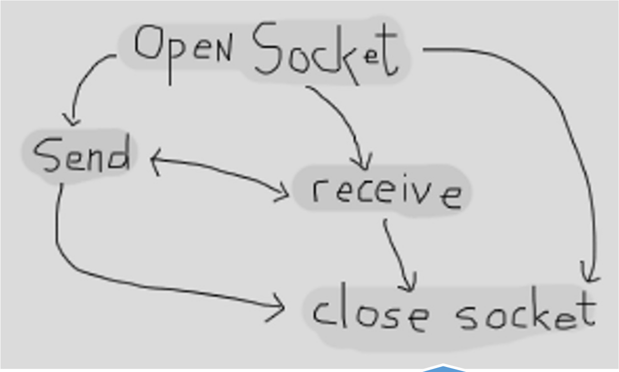
{ shi553, xu1415, xyzhang } @ purdue.edu

Outline

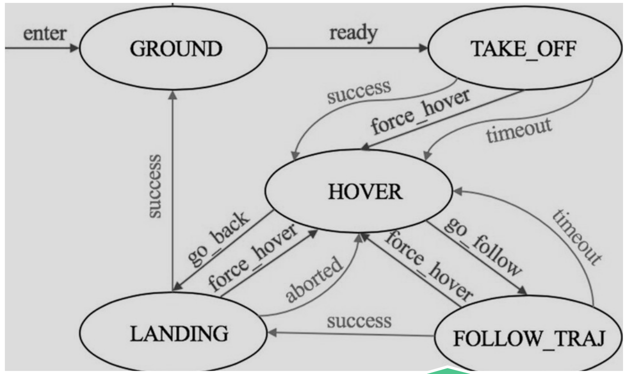
- State Machine in Practice
- Limitations of Existing Work
- Our Approach & Evaluation
- Take Away Messages

State Machine in Practice

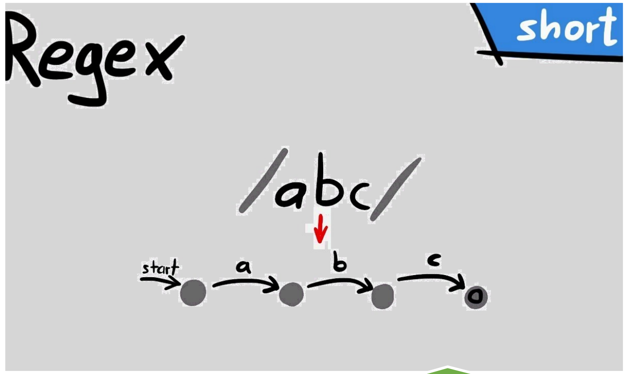
- State machines are broadly used in software applications



Networks



Robotics

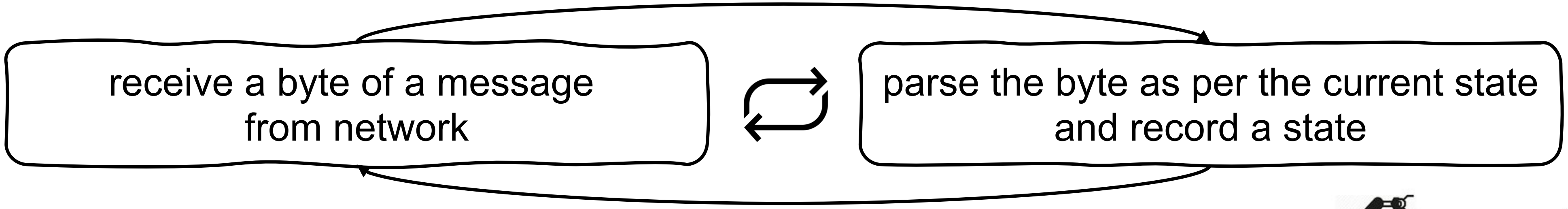


Parsers

......

State Machine in Practice

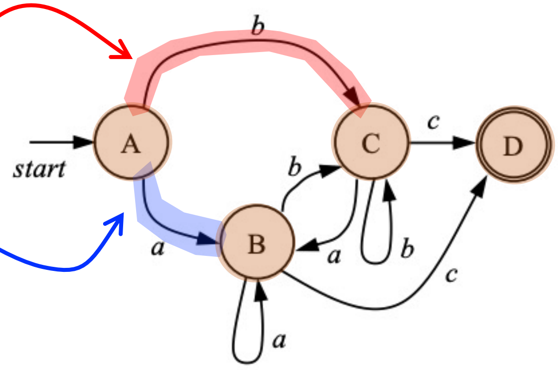
- When used to parse network messages, state machines enable high performance and **low latency**.
  - It does not have to wait for the entire message.



State Machine in Practice

- How are state machines coded in software?

```c
void read_message_and_parse() {
  char state = 'A';
  while (1) {
    switch(state) {
    case 'A':
        char in = read_next_msg_byte();
        if (in == 'a') { state = 'B'; }
        else { assert(in == 'b'); state = 'C'; }
        break;
    case 'B':
        …
    case 'C':
        …
    case 'D': …
    }}}
```



Regex: (a|b)+c

1. Use a loop to encode a state machine
2. Use **state variables** to record the state
   a) Referred in one iteration to control the path to execute
   b) Revised in one iteration to transition from one state to the other
3. Control which path to execute as per the state and the input
4. **There may be > 1 state variables**
5. **State value may not be enumerable**

State Machine in Practice

- How are state machines coded in software?

```
enum State { TOK, OK, ERR };
void parse(char *msg) {
    State state = TOK;
    string tok = "";
    while (1) {
        switch(state) {
        case TOK:
            char in = read_next_byte(msg);
            if (in == ':')
                if (iskey(tok)) state = OK;
                else state = ERR;
            else if (in == '^') { tok = ""; }
            else { assert('a'≤in≤'z'); tok+=in; }
            break;
        case OK: exit(0);
        case ERR: exit(1);
    }}}
```

The variable `state` with three possible values is not enough to parse the input!

Recognize non-empty token between ^ and :

⇩

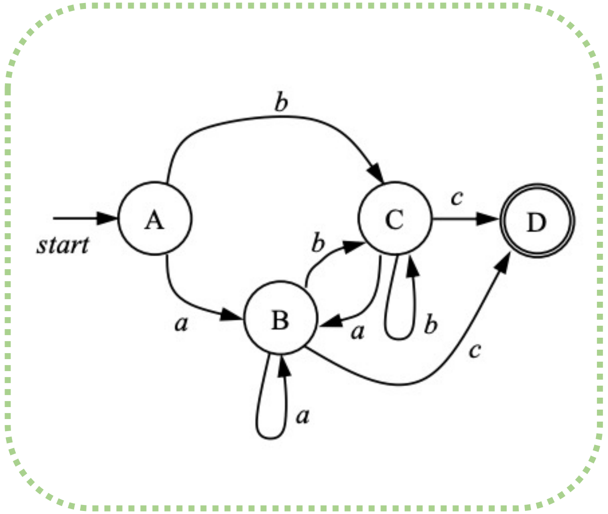
^^^<u>xyzabc</u>:

1. Use a loop to encode a state machine
2. Use **<u>state variables</u>** to record the state
   a) Referred in one iteration to control the path to execute
   b) Revised in one iteration to transition from one state to the other
3. Control which path to execute as per the state and the input
4. **<u>There may be > 1 state variables</u>**
5. **<u>State value may not be enumerable</u>**

Limitations of Existing Work

- State machines enable many security applications
  - Fuzzing, model checking, verification, …
- State Machine Inference by **Static Analysis**
  - Only work for simple cases that follow the pattern below
    - Only a **single** state variable and state value is **enumerable**
  - Relying on symbolic execution → Path and state explosion
- State Machine Inference by **Dynamic Analysis**
  - Relying on inputs, Suffering from low coverage

Limitations of Existing Work



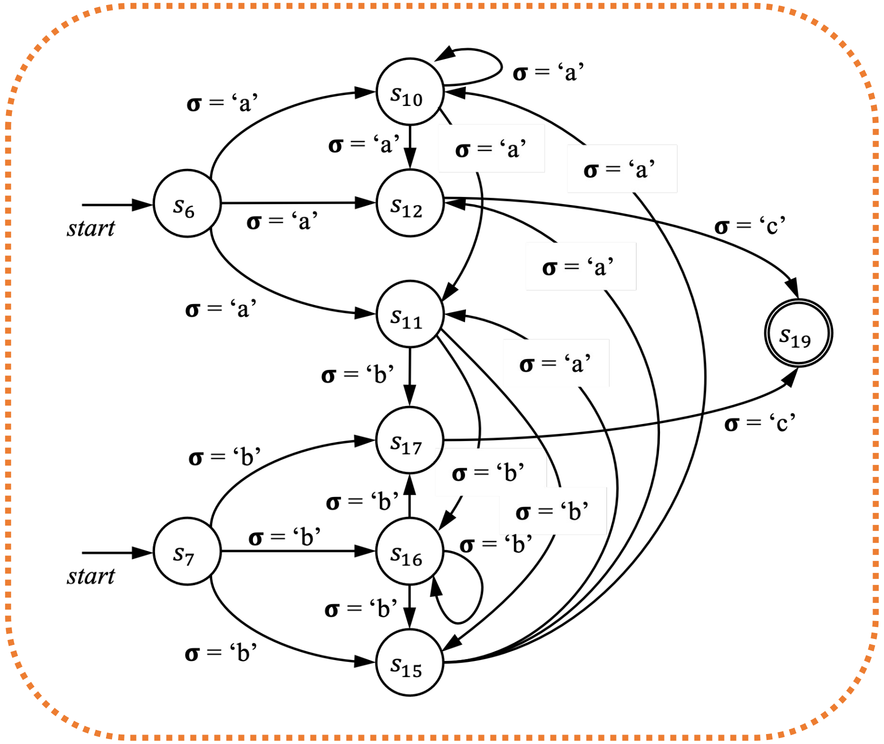
Groundtruth

State Machine

```
void read_message_and_parse() {
    char state = 'A';
    while (1) {
      switch(state) {
      case 'A': char in = read_next_msg_byte();
            if (in == 'a') { state = 'B'; }
            else { assert(in == 'b'); state = 'C'; }
            break;
      case 'B': char in = read_next_msg_byte();
            if (in == 'a') { /*do nothing*/ }
            else if (in == 'b') { state = 'C'; }
            else { assert(in == 'c'); state = 'D'; }
            break;
      case 'C': char in = read_next_msg_byte();
            if (in == 'a') { state = 'B'; }
            else if (in == 'b') { /*do nothing*/ }
            else { assert(in == 'c'); state = 'D';}
            break;
      case 'D': exit(0);
      }}}
```

Regex: (a|b)+c

Proteus

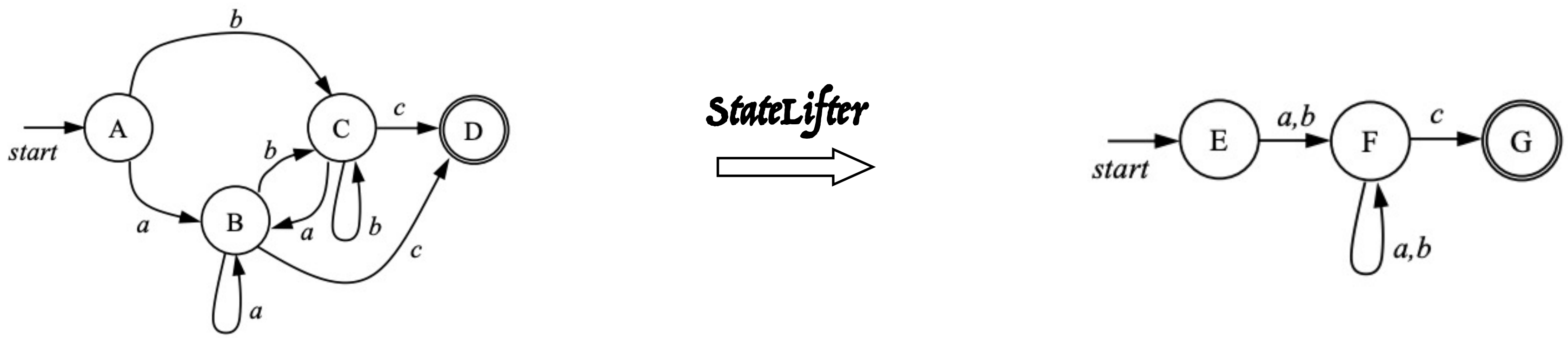


State Machine

Generated by Proteus

StateLifter in a Nutshell

- **Feature 1**: Inferring a **compressed state machine** even from the code that implements a complex but equivalent state machine.



- **Feature 2**: An abstract interpretation framework supporting **multiple and non-enumerable state variables and is proved to be sound**.

refer to our paper for details

Evaluation: Compared to Static Analyzers

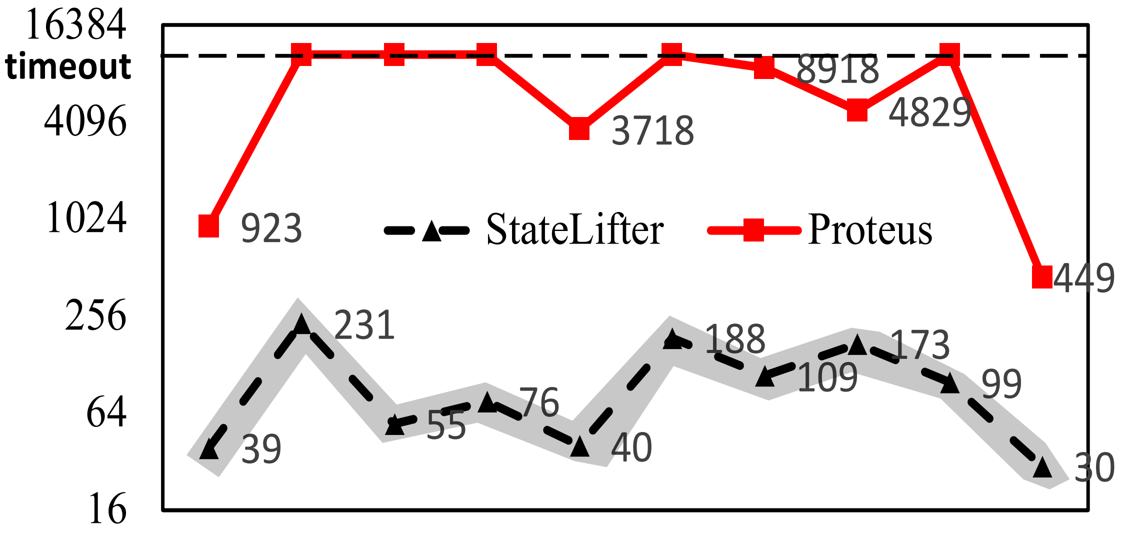
- We run both tools on 10 real-world parsers, and record the complexity of the resulting state machines.
- We record the time consumption of both tools.

Protocols	StateLifter		Proteus	
	#states	#transitions	#states	#transitions
ORP [11]	5	8	42	92
MAVLINK [12]	42	197	-	-
IHEX [5]	15	63	-	-
BITSTR [8]	22	75	-	-
TINY [16]	14	54	151	[illegible]
SML [7]	32	89	-	-
MIDI [17]	19	81	765	3812
MQTT [18]	28	87	105	581
RDB [15]	22	57	-	-
KISS [6]	6	12	24	142

40x simpler

4x simpler

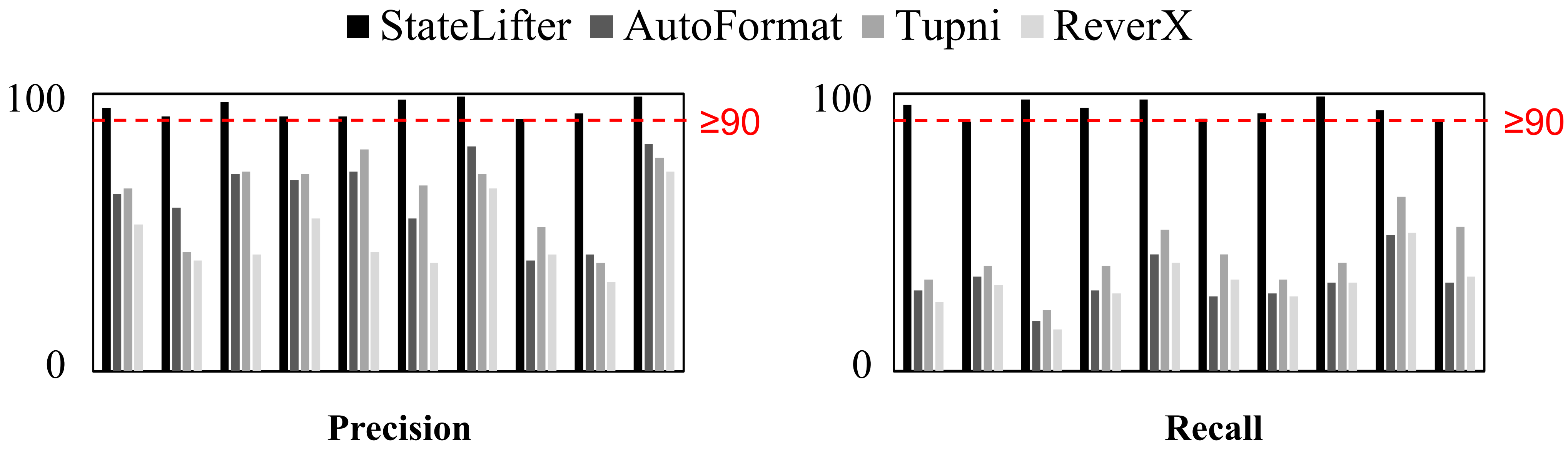
Size of the Inferred FSMs



Time Cost in Seconds

Evaluation: Compared to Dynamic Analyzers

- To drive dynamic analyzers, we randomly generate 1000 valid input messages for each protocol.



Evaluation

- Security Application: Fuzzing Network Protocol Parsers

1. Both mutation- and generation-based fuzzing
    a) For mutation-based fuzzer, generate seed corpus
    b) For generation-based fuzzer, directly generate input formats
2. Coverage is improved by **20% to 230%**
3. Detect 12 zero-day bugs, **10 more** than baselines

- Security Application: Fuzzing Cyber-Physical System (with PGFuzz)
    - We discover bugs in both Ardupilot and the fuzzer, PGFuzz
    - See an extended version of our paper (in arxiv)

Take Away Messages

- *StateLifter* is a static code analyzer that can infer precise state machine with high recall from the source code

- *StateLifter* is an abstract interpreter for state machine inference, with proof of soundness and completeness

- *StateLifter* enables many security analyses in different domains, considering the broad use of state machines in practice

THANKS FOR YOUR TIME!