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?

1. **void** read_message_and_parse() {
2.   **char** state = ‘A’;
3.   while (1) {
4.     switch(state) {
5.       case ‘A’:
6.         **char** in = read_next_msg_byte();
7.         if (in == ‘a’) { state = ‘B’; }
8.         else { assert(in == ‘b’); state = ‘C’; }
9.         break;
10.        case ‘B’:
11.          …
12.        case ‘C’:
13.          …
14.        case ‘D’:
15.         }}}

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**

Regex: (a|b)+c
State Machine in Practice

• How are state machines coded in software?

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

```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':  ...
        }
    }
}
```

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

xyzabc:

Recognize non-empty token between ^ and :
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 $\rightarrow$ Path and state explosion

• State Machine Inference by **Dynamic Analysis**
  • Relying on inputs, Suffering from low coverage
Limitations of Existing Work

Proteus

```
1. void read_message_and_parse() {
2.     char state = 'A';
3.     while (1) {
4.         switch(state) {
5.             case 'A': char in = read_next_msg_byte();
6.                 if (in == 'a') { state = 'B'; }
7.                 else { assert(in == 'b'); state = 'C'; }
8.                 break;
9.             case 'B': char in = read_next_msg_byte();
10.                if (in == 'a') { /*do nothing*/ }
11.                else if (in == 'b') { state = 'C'; }
12.                else { assert(in == 'c'); state = 'D'; }
13.                break;
14.             case 'C': char in = read_next_msg_byte();
15.                 if (in == 'a') { state = 'D'; }
16.                 else if (in == 'b') { /*do nothing*/ }
17.                 else { assert(in == 'c'); state = 'D';}
18.                 break;
19.             case 'D': exit(0);
20.         }
21.     }
22. }
```

Groundtruth State Machine

Regex: (a|b)+c

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.

<table>
<thead>
<tr>
<th>Protocols</th>
<th>StateLifter</th>
<th>Proteus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># states</td>
<td># transitions</td>
</tr>
<tr>
<td>ORP[11]</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>MAVLINK[12]</td>
<td>42</td>
<td>197</td>
</tr>
<tr>
<td>IHEX[5]</td>
<td>15</td>
<td>63</td>
</tr>
<tr>
<td>BITSTR[8]</td>
<td>22</td>
<td>75</td>
</tr>
<tr>
<td>TINY[16]</td>
<td>14</td>
<td>54</td>
</tr>
<tr>
<td>SML[7]</td>
<td>32</td>
<td>89</td>
</tr>
<tr>
<td>MIDI[17]</td>
<td>19</td>
<td>81</td>
</tr>
<tr>
<td>MQTT[18]</td>
<td>28</td>
<td>87</td>
</tr>
<tr>
<td>RDB[15]</td>
<td>22</td>
<td>57</td>
</tr>
<tr>
<td>KISS[6]</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

Size of the Inferred FSMs

Time Cost in Seconds

40x simpler
4x simpler
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!