Automated Testing of Stateful Network Protocol Implementations in the IoT

Sören Tempel¹, Rolf Drechsler¹,²

tempel@uni-bremen.de

¹Group of Computer Architecture, University of Bremen, Germany
²Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
Motivation

**Background:** IoT nodes exchange data via network protocols
- Protocol implementations often contain software bugs
- Some of these bugs (e.g. buffer overflows) are exploitable
- Problematic since IoT operating systems have few exploit mitigations

**Goal:** Automatically find such bugs in network modules
  ⇒ Emerging method for this purpose: *symbolic execution*
Background: Symbolic Execution

**Idea:** Enumerate reachable paths based on specific input source

- SW executed with symbolic values, represent set of concrete values
- Symbolic values are continuously constrained during execution
- Constraints on current path: *path constraints* (PC)
Background: Symbolic Execution

Idea: Enumerate reachable paths based on specific input source
- SW executed with symbolic values, represent set of concrete values
- Symbolic values are continuously constrained during execution
- Constraints on current path: *path constraints* (PC)

DSE: *Dynamic Symbolic Execution*
- Concrete execution drives symbolic execution
- Track symbolic constraints alongside concrete execution
- Branches are collected; later negated with an SMT solver
Example:

```c
void myfunc(int a) {
    if (a > 8)
        // ...
    else
        // ...

    if (a < 5)
        // ...
    else
        // ...
}
```
**Example:** Execution trace for `myfunc` with input $a = 9$

```c
void myfunc(int a) {
    if (a > 8) {
        // ...
    } else {
        // ...
    }
    if (a < 5) {
        // ...
    } else {
        // ...
    }
}
```

![Diagram of execution trace for `myfunc` with input $a = 9$.](image_url)
**Exploration:** Negate unexplored branch \( \neg (a > 8) \), solve resulting query
\[ \Rightarrow \text{Restart execution with concrete input (e.g. } a = 8) \]

```c
void myfunc(int a) {
    if (a > 8)
        // ...
    else
        // ...

    if (a < 5)
        // ...
    else
        // ...
}
```

![Decision diagram](image_url)
Dynamic Symbolic Execution (3/3)

**Goal:** Ideally discover all execution paths
⇒ Repeat until all branches have been negated

```c
void myfunc(int a) {
    if (a > 8)
        // ...
    else
        // ...
    if (a < 5)
        // ...
    else
        // ...
}
```
Input Interfaces in the IoT

Today!

CoAP
MQTT-SN
...

WSN

RIOT

SymEx

Sensor

RIOT Summit 2022
Input Interfaces in the IoT

WSN → CoAP → Sensor
MQTT-SN → RIOT → SymEx-VP

Today!
RIOT Summit 2022

Sensor
MQTT-SN: Stateful protocol for data exchange in the IoT

- Certain code can only be tested by establishing a state first
- For example, subscribing to a specific topic
- Results in a large state space for symbolic execution
  ⇒ Cannot be fully explored using symbolic execution
Goal: Discovering “interesting” execution paths first
⇒ Observation: Many inputs are rejected early on

Approach: Partially specify protocol message format
- Embedded domain specific language (EDSL)
- Based on the Scheme programming language

Figure: Message format for MQTT-SN SUBACK.

(define-input-format (suback id)
  (make-uint 'len 8 8)
  (make-uint 'type 8 MQTT-SUBACK)
  (make-symbolic 'flags 8)
  (make-symbolic 'topicid 16)
  (make-uint 'msgid 16 id)
  (make-symbolic 'code 8
    `((And
        (Uge ,code 0)
        (Ule ,code 3)))))

Figure: Message format for MQTT-SN SUBACK.
**Challenge:** MQTT-SN is a stateful protocol
⇒ Message format depends on protocol state

**Approach:** Also describe protocol state machine
- With a separate Scheme-based EDSL
- Advance protocol state based on received messages
- Return new symbolic message depending on state

---

**Figure:** Overview of message format exchange.
Specification of the MQTT-SN State Machine

**Needed:** MQTT-SN state machine description

- Described as a finite-state machine
- Transitions based on input packet
- Each transitions returns a response format

```lisp
(define-state-machine mqttt-machine
  (start pre-connected)

(define-state (pre-connected input)
  ...

(define-state (connected input)
  (switch (mqtt-msg-type input)
    ((SUBSCRIBE)
      (-> (make-resp (suback-fmt (msg-id input)))
        subscribed))
    ((DISCONNECT)
      (-> (make-resp disconn-fmt)
        disconnected)))
  ...

Figure: Excerpt of the MQTT-SN state specification.
```
Symbolic Packet Sequences

**Problem:** Need to reason about sequence of packets

⇒ Further increase of state space
Multipacket Exploration

**Simplified Algorithm:**

1. Explore program up to a sequence length of \( k \)
2. Restart execution when packet \( k \) was processed
3. When coverage is stagnant: Increment \( k \)

\[ \Rightarrow \text{Partially explored paths are re-executed continuously} \]
Research Question: Does our symbolic execution approach improve coverage?

⇒ Experiments with RIOT’s MQTT-SN implementations
Research Question: Is the approach applicable to other protocols?
⇒ Experiments with RIOT’s and Zephyr’s DHCP implementations
Bugs found in RIOT

Bugs Found:
1. #18307: out-of-bounds read in dhcpv6 module
2. #18289: missing mutex_unlock in asymcute
3. #18434: null pointer dereference in asymcute

Future Work:
- Integrate protocol rules into specification?
- Assessment of created protocol specifications
- ...
Summary

**Key Insight:** High coverage in complex network modules via symbolic execution
⇒ With comparatively little manual effort

**Contributions:**
1. Input specification language for message formats\(^1\)
2. Specification language for protocol state machines\(^2\)
3. Enhanced version of SymEx-VP with new exploration engine\(^3\)


\(^1\)https://github.com/agra-uni-bremen/sisl
\(^2\)https://github.com/agra-uni-bremen/sps
\(^3\)https://github.com/agra-uni-bremen/sps-vp