Fuzz Testing

State-of-the-Art
and
Application to Software for IoT

Bengt Jonsson
Uppsala Univ.
bengt@it.uu.se

Konstantinos Sagonas
Uppsala Univ.
kostis@it.uu.se

Nicolas Tsiftes
RISE
nicolas.tsiftes@rise.se

UPPSALA
UNIVERSITET

Outline

Overview of aSSIsT: Software Security for IoT

Fuzz Testing: Overview

Fuzz Testing: Experiences from application to IoT Software

aSSIsT

Software Security for the IoT

very short overview

UPPSALA
UNIVERSITET

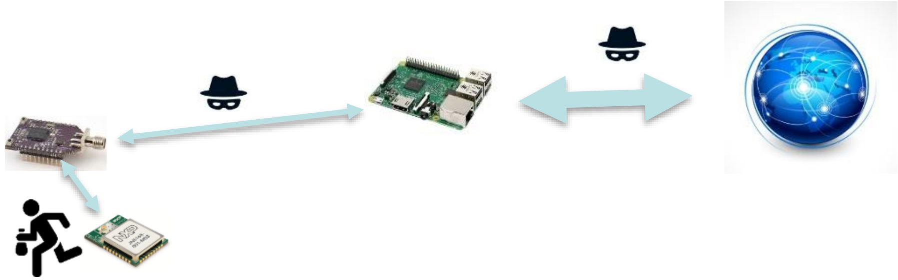
Background and Motivation

Internet of Things (IoT):

- Primary concern: **Security**

Focus of aSSIsT:

- Security of **IoT Software**
  - in platforms, communications, applications.



Challenges:

- Large attack surface
  - Internet, Wireless, Physical
- Resource-constrained platforms
  - ⇒ Lack of support (memory protection, intrusion detection, …)

aSSIsT: Secure Software for IoT

Project duration: 2018-2024, https://assist-project.github.io

Funding: Swedish Foundation for Strategic Research (SSF)

Participating Groups

Uppsala University, Dept. IT
Senior: Bengt Jonsson, Kostis Sagonas, Mohammed Faouzi Atig
PostDocs: Paul Fiterau-Brostean, Sandip Ghosal, Rémi Parrot
PhD: Hooman Asadian, Sarbojit Das, Magnus Lång, Fredrik Tåkvist

RISE CS, **Kista**
Senior: Luca Mottola, Shahid Raza, Nicolas Tsiftes, Thiemo Voigt
PostDocs: Chetna Singhal
Ph.D: Anum Khurshid

Reference Group
ASSA ABLOY, Intel Sweden, LumenRadio, Upwis, Wittra

UPPSALA UNIVERSITET

aSSIsT: Overall Goals

Challenge: Develop techniques to make IoT software resilient against security attacks, for use by developers of Software for IoT

Goals:

1. Detecting software vulnerabilities
   - Software analysis, fuzzing
2. Testing and verification of (security) protocol implementations
   - Conformance testing, security testing
3. Run-time protection mechanisms
   - Trusted execution environments
   - Low-power intermittent computing

Demonstrators:

- IoT OS: Contiki-NG
- IoT protocols: DTLS (Datagram TLS),

UPPSALA UNIVERSITET

RI. SE

Software Analysis for IoT Software

Detect bugs and vulnerabilities using

- **Fuzzing** (or **fuzz testing**)
  - fast software testing based on random inputs

- **Stateless Model**
  - for findi... cy errors

In next part of presentation

UPPSALA UNIVERSITET

Testing of Security Protocols Implementations

Challenge:

Cover all possible sequences of attacker inputs

Challenge 1:

Correct ordering of packets received and sent

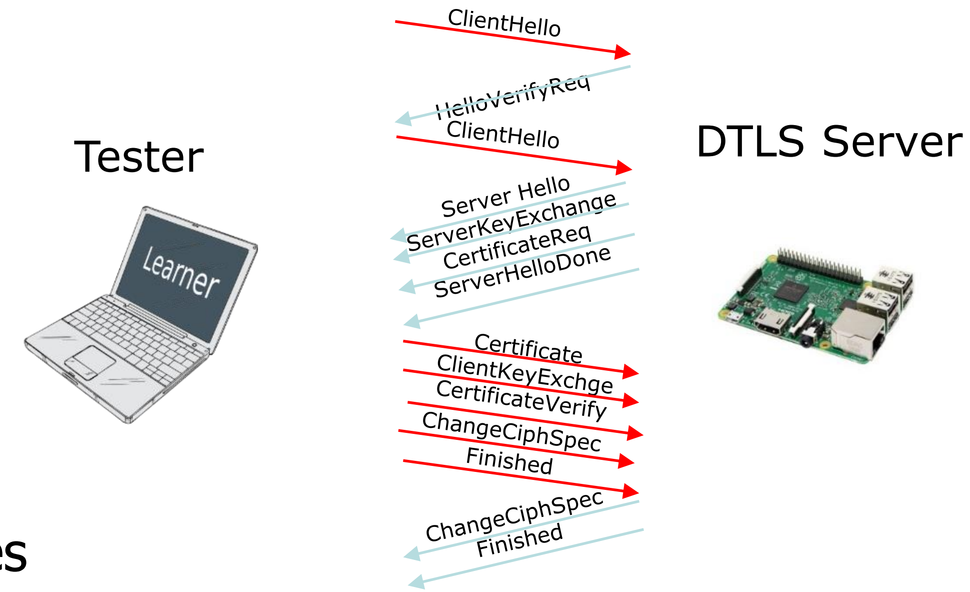
- E.g., can authentication be bypassed?

Solution:

State Fuzzing

- Systematic application of constructed input sequences
- Automated detection of packet ordering errors
- Applied to DTLS, SSH, TCP

Connection Establishment in DTLS



Testing of Security Protocols Implementations

Challenge:

Cover all possible sequences of attacker inputs

Challenge 2:

Correctness of packet data

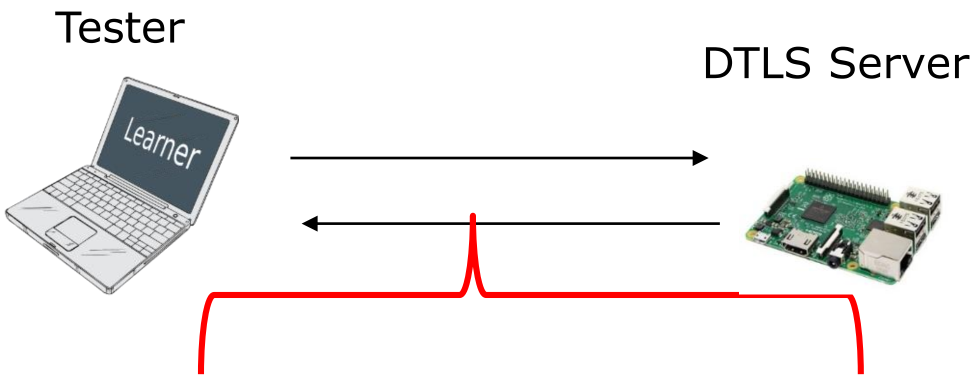
- E.g., is correctness of size fields in input packets checked?
  - Insufficient checks cause overreads/overwrites (cf. Heartbleed)

Solution:

Symbolic Execution

- Covers all values of data fields in input packets
- Detects insufficient checking of packet contents, and incorrect data in output
- Applied to DTLS

Packet structure in DTLS




```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

UPPSALA UNIVERSITET

Impact on Existing IoT Software

Fixes of bugs and vulnerabilities found in fuzzing research:

- For Contiki-NG:
  - 18 bug fixes and 11 CVEs
  - First continuous integration test suite for Contiki-NG which directly targets security
- For DTLS implementations:
  - 30+ bug fixes and 3 CVEs
  - In GnuTLS, Java SSE, OpenSSL, PionDTLS, Scandium, TinyDTLS, WolfSSL
- For QUIC implementations: 3 bug fixes

Open-source software tools:

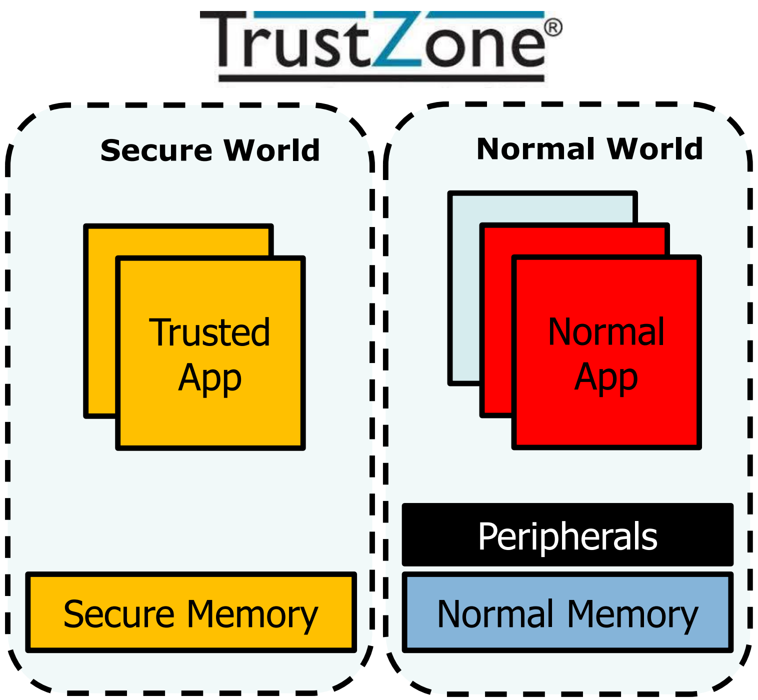
- *DTLS-Fuzzer:* Framework for state fuzzing of DTLS implementations
- *PropEr:* Property-based testing, now also for network protocols
- *Nidhugg:* Finding concurrency errors in concurrent C code

UPPSALA UNIVERSITET

Trusted Execution Environments (TEE)

TEEs provide efficient mechanisms to isolate critical software functionality

- Secure boot, digital signatures, authentication, firmware update
- Memory and peripherals partitioned into **secure** and **normal** world
- ARM supports TEE security extension in microcontrollers: TrustZone-M



UPPSALA UNIVERSITET

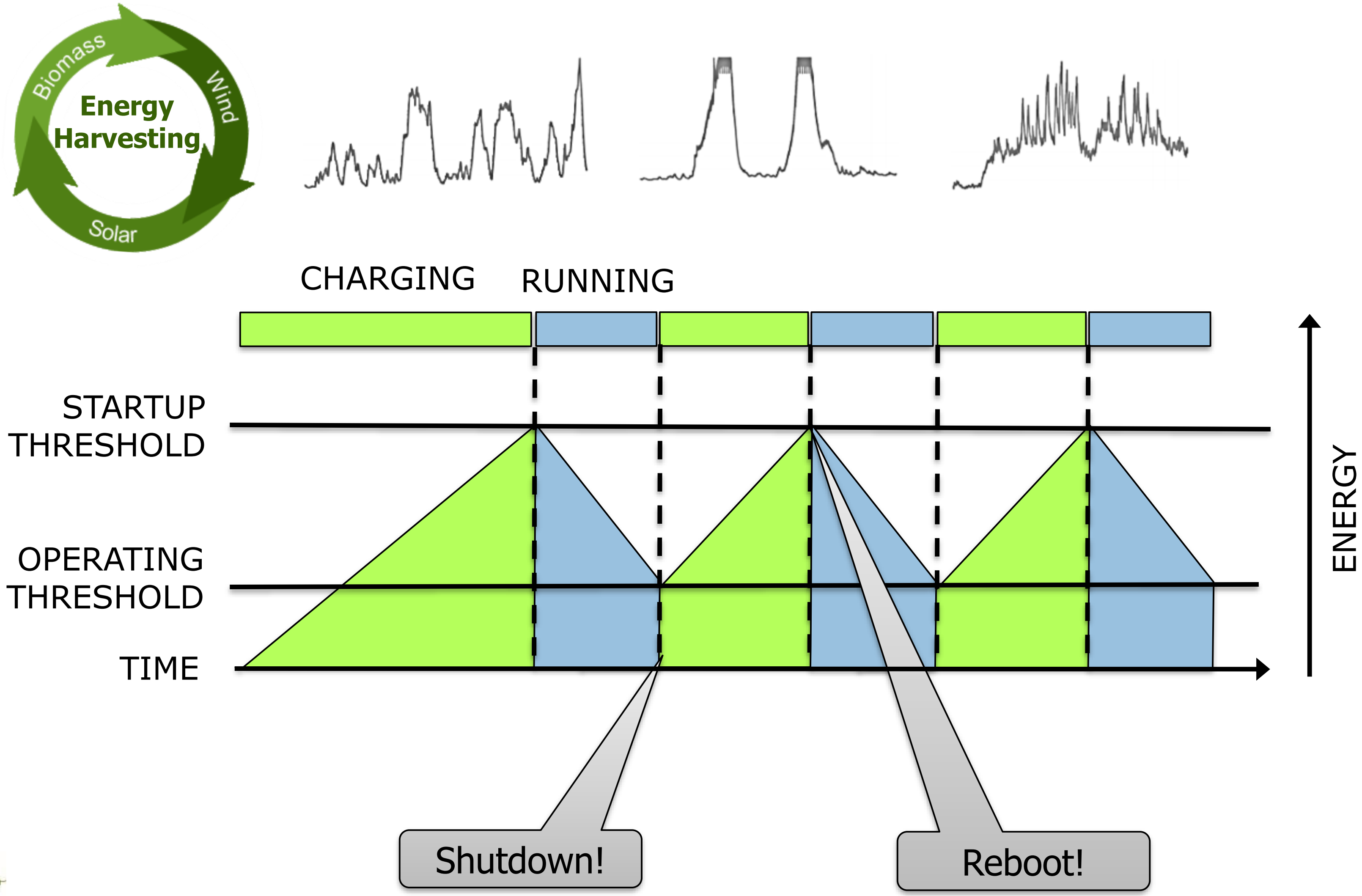
RI. SE

Trusted Execution Environments (TEE)

We have addressed several challenges:

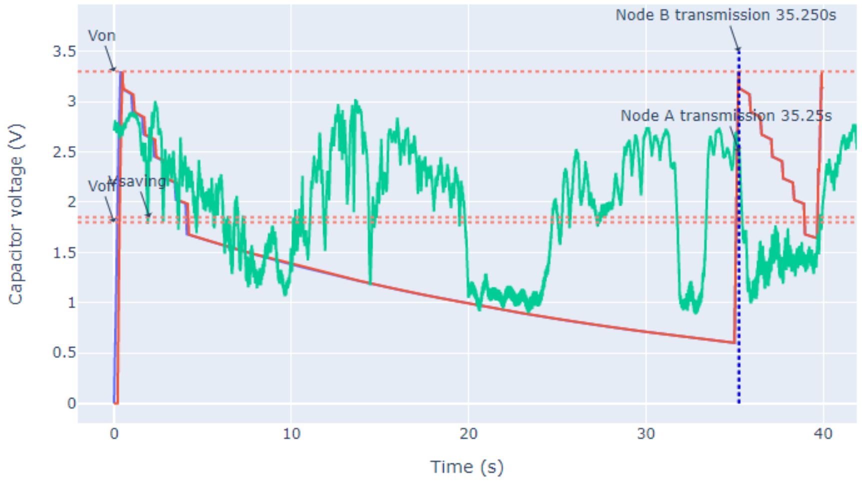
1. Authenticating communication requests from normal to secure world
   - ShieLD: Lightweight message protection scheme ensuring confidentiality and integrity, does not rely on encryption
2. Detecting if a secure application is compromised
   - TEE-watchdog: Mitigation of unauthorized activity in TEE
3. Remote attestation and Software-state certification of IoT devices
   - AutoCert: Combines Software-state certification and PKI
4. Supporting TEEs in Contiki-NG
   - Work in progress

Securing Intermittent Computing



Intermittent Computing: Results

- **Problem:** Securing persistent state
  - **Results**: Comparing different schemes
- **Problem:** Energy attacks
  - How to detect the attacker is messing with the source?
  - How to mitigate the effects?
- **Findings:**
  - Energy attacks may cause priority inversion, livelocks, and unwanted synchronization
- **Outcomes:**
  - A monitoring system with 95%+ accuracy and little overhead
  - A mitigation architecture to let programmers deal with it



Opportunities for Future Work and Collaboration

Software analysis

- Test effectiveness of fuzzing techniques on other IoT software
- Fuzzing IoT software on target platforms
  - E.g., by supplying fuzzing infrastructure on emulation platforms

Testing of protocol implementations

- Applying test techniques to other IoT protocols
  - Include EDHOC, OSCORE, QUIC

TEEs

- Realization on open-source hardware

Intermittent computing

- Low-power reconfigurable hardware

Fuzz Testing (Fuzzing)
An Introduction

Kostis Sagonas

kostis@it.uu.se

Dynamic Program Analysis

- Run program in instrumented execution environment
  - Static instrumentation
  - Binary translator
  - Emulator
- Look for bad stuff
  - Assertion violations
  - Exceptions (e.g., null pointer dereferences)
  - Use of invalid (out of bounds, freed, etc.) memory
  - Undefined behavior (e.g., arithmetic overflows)
  - etc.

Regression vs. Fuzzing

Regression: Run program on many “expected” inputs, look whether bugs were introduced.

Goal: Check that normal program uses are OK.

Fuzzing: Run program on many unexpected “random” inputs, look for errors.

Goal: Prevent attackers from encountering exploitable errors.

Fuzzing Basics

- Automatically generate test cases
  - typically given some valid inputs as “seeds”.
- Many slightly anomalous test cases are input into a target interface.
- Application is monitored for errors.

Fuzzing Example

- Standard HTTP GET request

```
GET /index.html HTTP/1.1
```

- Anomalous requests generated by fuzzing

```
AAAAAA...AAAA /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1
```

How To Generate Inputs?

- Mutation Based
- Generation Based
  - e.g., Grammar-Based Fuzzing
- Feedback Based
  - e.g., Coverage-Guided Fuzzing
- Hybrid Fuzzing
  - e.g., Fuzzing Guided by Symbolic Execution

Mutation-Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed.
- Anomalies are added to existing valid inputs.
- Mutations may be completely random or follow some heuristics (e.g., remove a bit, add a byte, flip two characters, etc.).

Example: Fuzzing a pdf Viewer

- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script to)
  1. Grab a file
  2. Mutate that file
  3. Feed it to the program
  4. Record if the program crashed/hanged/etc.

     (and remember the input that crashed it)

Mutation-Based Fuzzing

- Strengths
  - Super easy to setup and automate
  - Little to no program knowledge required
- Weaknesses
  - Limited by initial corpus
  - May fail for protocols with checksums, those which depend on challenge response, etc.

Generation-Based Fuzzing

- Test cases are generated from some description of the format: protocol RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs.
- Knowledge of protocol should give better results than random fuzzing.

Generation-Based Fuzzing

- Strengths
  - Completeness
  - Can deal with complex dependencies e.g. checksums
- Weaknesses
  - Have to have spec of protocol
    - Often can find good tools for some protocols e.g. http, SNMP
  - Writing generator can be labor intensive for complex protocols
  - The spec is not the code

How Much Fuzz Is Enough?

- Mutation-based fuzzers can generate an infinite number of test cases…
  - When has the fuzzer run long enough?
- Generation-based fuzzers generate a finite number of test cases.
  - What happens when they're all run and no bugs are found?

Code Coverage

- Some of the answers to these questions lie in code coverage.
- Code coverage is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools (e.g., gcov).

Types of Code Coverage

- Line coverage
  - Measures how many lines of source code have been executed.
- Branch coverage
  - Measures how many branches in code have been taken (conditional jumps)
- Path coverage
  - Measures how many paths have been taken

Example

```
if (a > 1) x = 1;
if (b > 1) y = 2;
```

Requires:

- 1 test case for line coverage
- 2 test cases for branch coverage
- 4 test cases for path coverage

  `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`

Fuzzing Rules of Thumb

- More fuzzers is better
  - Different fuzzers often find different bugs.
- The longer you run, the more bugs you find.
- Best results come from guiding the process.
- Code coverage can be very useful for guiding the process.

Grey-box Fuzzing

- Select mutations based on fitness metrics
- Prefer mutations that give
  - Better code coverage
  - Modify inputs to potentially dangerous functions (e.g. memcpy)

Fuzzing IoT Software

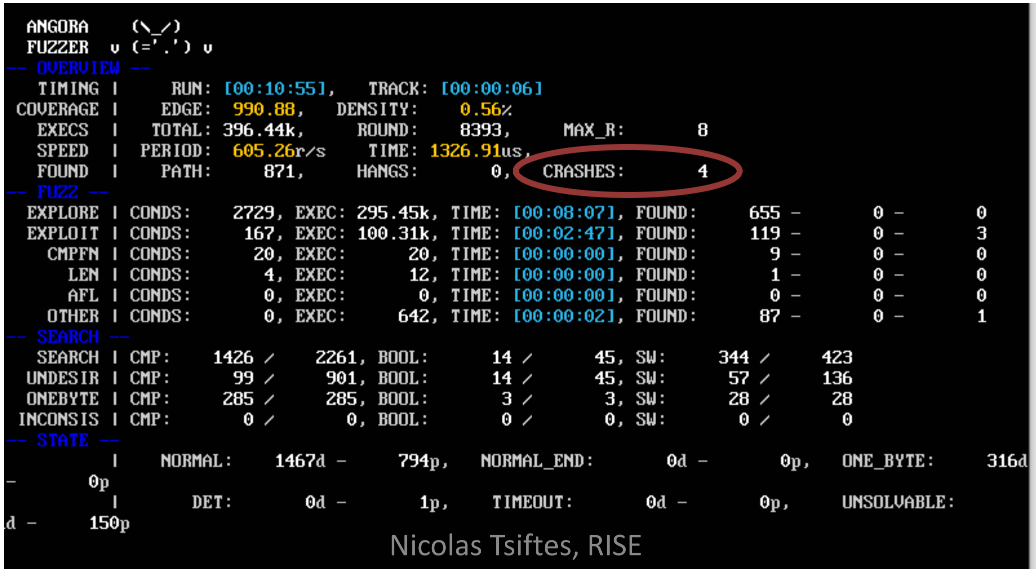
Technical Overview

Setting Up Fuzzing

- Create a fuzzing harness
  - Passes input data from fuzzer to target app
  - Typically a small module or shell script
- Generate or collect a test seed
  - Example 1: pre-recorded protocol message sessions for fuzzing a protocol implementation
  - Example 2: different types of binaries when fuzzing a dynamic loader

Fuzzing Output

- Input data leading to new code execution paths in the target application
- Input data causing crashes or hangs
  - Re-run application with GDB or Valgrind to debug



Detecting Vulnerabilities

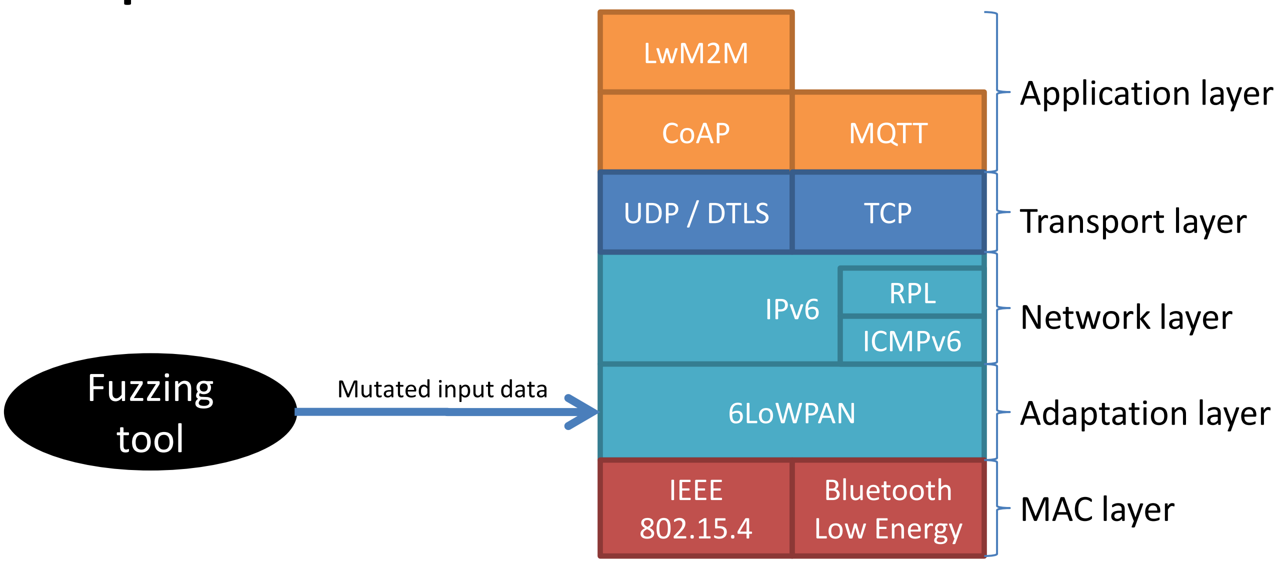
- Crashes
  - E.g., out-of-bounds memory accesses, NULL pointer dereferences
- Hangs
  - E.g., infinite loops, thread deadlocks
  - Set fuzzer timeout depending on target app
- Enhanced bug detection with sanitizers
  - E.g., undefined behavior not causing a crash
  - Address Sanitizer, Undefined Behavior Sanitizer

Fuzzing in Atypical Environments

- Challenges
  - Many state-of-the-art fuzzers require Linux env.
  - Fuzz software on IoT devices?
  - No access to source code
- Solutions
  - Emulator-based fuzzing of binaries
    - AFL QEMU mode
  - Adapted fuzzing target setup
    - Run IoT OS as a Linux application
  - Specialized tools
    - FIRM-AFL, IoTFuzzer

Experiences with Contiki-NG

- OS for resource-constrained IoT devices
  - Open-source development
  - Used in research and industry
- Low-power IPv6 stack



Contiki-NG Network Stack Fuzzing

- Multiple protocol layers
- Must pass many field validity checks to reach upper layers
  - 6LoWPAN → IPv6 → UDP → CoAP → LwM2M
- Alternative entry points for fuzzed input packets
  - 6LoWPAN, IPv6, CoAP, DNS resolver

Which fuzzing method is most effective when applied on a codebase of Contiki-NG's characteristics?