# Fuzzing Network Protocol Implementations (Lecture 6)

Prof. Mathy Vanhoef

DistriNet – KU Leuven – Belgium

# Types of protocols

1. **Stateless** protocols
   - Previous packets don’t influence response to current packet
   - Examples: ARP, ICMP, DNS, HTTP (without cookies),…

2. **Stateful** protocols
   - Code being executed depends on current & previous input
   - Examples: TLS, WPA2, IMAP, SMTP, FTP,…

# Fuzzing stateless protocols

How can we fuzz them?

- Make them behave like a ~~network service~~ program that processes ~~an incoming packet~~ a file / given input
- We can then apply the typical fuzzers like AFL

# Fuzzing stateless protocols: Preeny’s desock

Preeny’s desock module:

- Converts a network service into a command-line program
- Intercepts (= “hooks”) socket library functions:
  - Intercepts socket(), bind(), listen(), and accept()
  - Returns sockets that are synchronized to stdin and stdout
  - Use LD_PRELOAD to load the desock module when starting a program
- Popular in practice. Included in AFL++ (called desock_dup).

→ Can be combined with many fuzzers!

# Introduction: stateful protocols

We will now focus on **stateful** network protocols

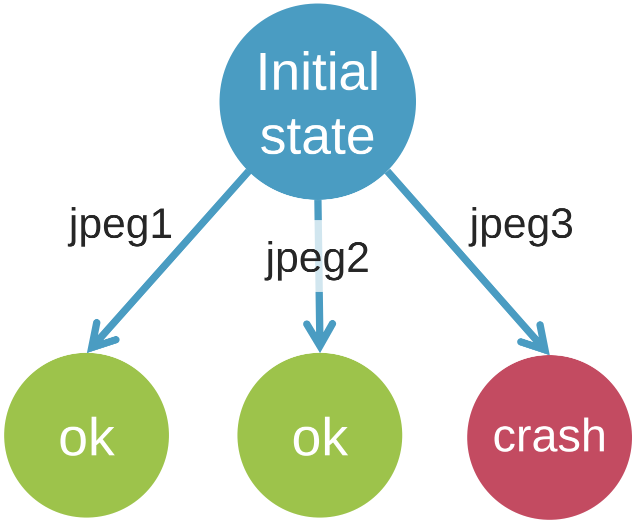
› Code being executed depends on current & previous input:

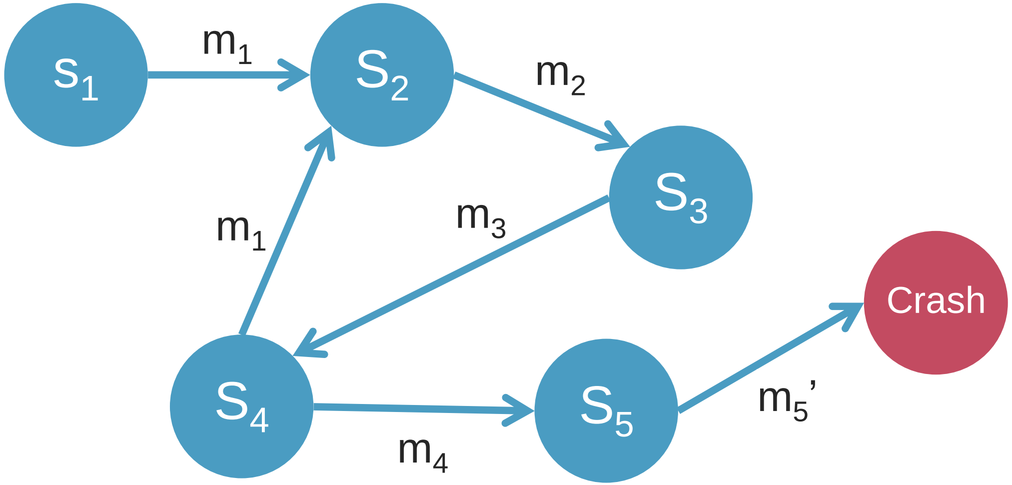
```
void handle_packet(uint8_t *p, size_t len) {
    switch (current_state) {
    case INIT: handle_init(p, len); break;
    case AUTH: handle_auth(p, len); break;
    // ... other states here ...
    case DATA: handle_data(p, len); break;
    }
}
```

# What about network protocols?

What makes fuzzing **stateful** network protocols special?

- The code being executed depends on previous input
- This is in contrast with, e.g., image parsing tools



# How is this handled in practice?

One approach is to fuzz individual functions

- “Unit test(s)” provide mutated input to an internal function(s)
- Internal function corresponds a certain state in the protocol
- The typical fuzzers can then be used again

This is currently a common technique in practice:

- For instance, used by OpenSSL (TLS) and hostap (Wi-Fi)
- Downside is that many “unit tests” must be written

# Practical example 1: OpenSSL

The OpenSSL library:

- Implements the TLS/SSL protocol
- Commonly used in HTTPS

The TLS protocol consists of two phases:

1. Handshake protocol: negotiate keys, many crypto operations
2. Record protocol: exchange encrypted data. Records with an invalid message authentication code (“MAC”) are dropped.

# Practical example 1: OpenSSL

How is the OpenSSL library being fuzzed?

› Build-in support to fuzz using libFuzzer and AFL

```c
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    SSL *client = NULL;
    BIO *in;
    BIO *out;
    SSL_CTX *ctx;

    if (len == 0)
        return 0;

    /* This only fuzzes the initial flow from the client so far. */
    ctx = SSL_CTX_new(SSLv23_method());
    if (ctx == NULL)
        goto end;

    client = SSL_new(ctx);
    if (client == NULL)
        goto end;
    OPENSSL_assert(SSL_set_min_proto_version(client, 0) == 1);
    OPENSSL_assert(SSL_set_cipher_list(client, "ALL:eNULL:@SECLEVEL=0") == 1);
    SSL_set_tlsext_host_name(client, "localhost");
    in = BIO_new(BIO_s_mem());
    if (in == NULL)
        goto end;
    out = BIO_new(BIO_s_mem());
    if (out == NULL) {
        BIO_free(in);
        goto end;
    }
    SSL_set_bio(client, in, out);
    SSL_set_connect_state(client);
    OPENSSL_assert((size_t)BIO_write(in, buf, len) == len);
    if (SSL_do_handshake(client) == 1) {
        /* Keep reading application data until error or EOF. */
        uint8_t tmp[1024];
        for (;;) {
            if (SSL_read(client, tmp, sizeof(tmp)) <= 0) {
                break;
            }
        }
    }
 end:
    SSL_free(client);
    ERR_clear_error();
    SSL_CTX_free(ctx);

    return 0;
}
```

```c
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    SSL *client = NULL;

    ...

    SSL_set_connect_state(client);
    OPENSSL_assert((size_t)BIO_write(in, buf, len) == len);
    if (SSL_do_handshake(client) == 1) {
        /* Keep reading application data until error or EOF.
        uint8_t tmp[1024];
```

# Practical example 1: OpenSSL

How is the OpenSSL library being fuzzed?

- Build-in support to fuzz using libFuzzer and AFL
- An initial input corpus is also provided
- Handling crypto operations deterministically:
  - Uses option to disable generation of random numbers
  - Checks of correct MACs is not disabled
- Continuously being fuzzed by Google’s OSS-Fuzz

# Side note: Google ClusterFuzz and OSS-Fuzz

ClusterFuzz: a framework to fuzz on large clusters

- The OSS-Fuzz instance runs on 100,000+ machines
- Has automatic bug reporting, support for various fuzzers, etc.

OSS-Fuzz: fuzzing of open-source software

- Build on top of ClusterFuzz. Contains scripts to fuzz each project (building the project & giving it mutated input)
- Found over 40,500 bugs in 650 open-source projects

# Practical example 2: Wi-Fi hostap

Similar to OpenSSL:

› Support to fuzz using libFuzzer and AFL

› Initial input corpus is provided

› Continuously being fuzzed by Google’s OSS-Fuzz

› No special code changes to facilitate fuzzing

# Practical example 2: Wi-Fi hostap

## How does hostap tackle state?

› Fuzzes functions that handle various connection stages

path: root/tests/fuzzing

| Name |
|---|
| README |
| ap-mgmt |
| asn1 |
| build-test.sh |
| dpp-uri |
| eap-aka-peer |
| eap-mschapv2-peer |
| eap-sim-peer |
| eapol-key-auth |
| eapol-key-supp |
| eapol-supp |

Relevant example “fuzz unit tests”:

1. ap-mgmt: frames sent at start of handshake
2. eap-mschapv2-peer: authentication protocol
3. eapol-key-auth: session key negotiation stage

# Limitations

Several challenges remain:

- Must write “unit test” for every (important) function
  - May require deep knowledge of the implementation
  - Interactions between different functions may be missed
  - Not easy to assure that all states are covered
- Handling checksums or message authentication codes
- Need good initial input corpus
  - Hostap only has a single input frame for some unit tests…

# Properly fuzzing stateless protocols

There are effectively two input grammars to consider:

1. The grammar defining the allowed **format of packets**
2. The grammar defining the allowed **order of packets**

How to explore both aspects while fuzzing/testing?

- Assume one grammar is known & explore the other
  - For instance: using state inference tools
- Many other options exists as well…

# Fuzzers for stateful systems: an overview[1]

1. Grammar-based (generational)
2. Grammar learner
3. Evolutionary
4. Evolutionary grammar-based
5. Evolutionary grammar-learner
6. Machine learning-based
7. Man-in-the-middle based

Typical components:

- Test harness
- SUT
- Anomaly detector

[1] Source: "An overview of Stateful Fuzzing" by Seyed Andarzian, Cristian Daniele, and Erik Poll.

# Fuzzers for stateful systems: an overview[1]

1. **Grammar-based (generational)**
2. **Grammar learner**
3. Evolutionary
4. Evolutionary grammar-based
5. Evolutionary grammar-learner
6. Machine learning-based
7. Man-in-the-middle based

Typical components:

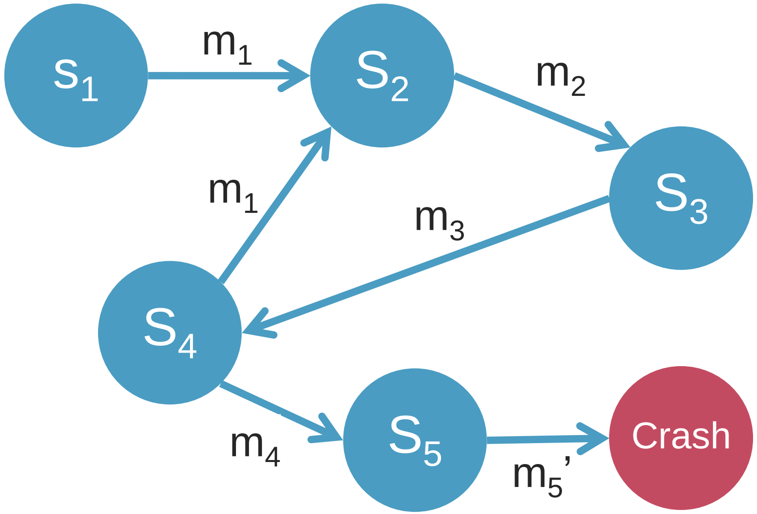
- Test harness
- SUT
- Anomaly detector

[1] Source: "[An overview of Stateful Fuzzing]" by Seyed Andarzian, Cristian Daniele, and Erik Poll.

# Grammar-based fuzzing

Define packet layout and state machine

- Fuzzer then sends valid packets to reach a target state
- When in the target state, send malformed/mutated packets




Example:

- Send packets to reach each state, will eventually test state $S_5$
- Then send mutations of $m_5$
- Will eventually detect the crash?

# Other grammar-based fuzzers

**BooFuzz** protocol fuzzer (fork/successor of Sulley)

1. The structure of each message is first defined

Name of the message

```
user = Request("user", children=(
    String("key", "USER"),
    Delim("space", " "),
    String("val", "anonymous"),
    Static("end", "\r\n")
))
```

Name of the field

Default field value

Field type: impacts mutation during fuzzing

# Other grammar-based fuzzers

**BooFuzz** protocol fuzzer (fork/successor of Sulley)

1. The structure of each message is first defined

```
passw = Request("pass", children=(
    String("key", "PASS"),
    Delim("space", " "),
    String("val", "james"),
    Static("end", "\r\n"),
))
```

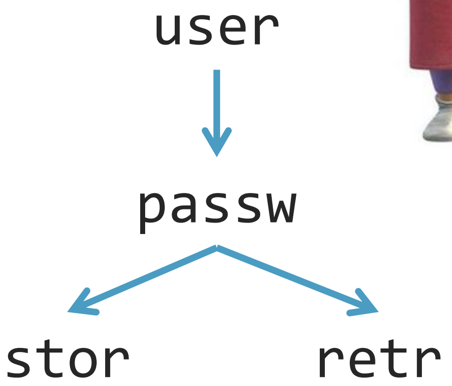
**Note: these block-based grammars are inspired by the (underdocumented?) SPIKE fuzzer**

# Other grammar-based fuzzers

**BooFuzz** protocol fuzzer (fork/successor of Sulley)

2. State machine is defined by connecting messages

```
session.connect(user)
session.connect(user, passw)
session.connect(passw, stor)
session.connect(passw, retr)
```




→ `user` is sent before fuzzing `passw`, `user` and `passw` is sent before fuzzing `stor` or `retr`. Doesn't fuzz order of messages.

# Other grammar-based fuzzers

**Peach** network protocol fuzzer

- Like Sulley/BooFuzz, but uses XML for the grammar
- Initially an open-source project.

Commercial edition received updates and features

- There (was) a community edition, but it lacked such updates
- GitLab open-sourced core engine of commercial Peach (2021)
  - Known as the GitLab Protocol Fuzzer Community Edition
  - The commercial version is no longer available…?

# Other grammar-based fuzzers

**Peach** network protocol fuzzer

- Like Sulley/BooFuzz, but uses XML for the grammar

Main mutation strategies of Peach:

1. Random: selects n fields from the data model. These fields are modified using a random mutator function.
2. Sequential: all fields are mutated in order using all possible mutator functions.

Limitation: the order of messages isn't fuzzed.

# Grammar learning: state machine inference

Other downsides of BooFuzz and Peach:

- Must manually specify state machine *and* packet format
- Can’t detect new interesting inputs based on code coverage

First point can be improved by **inferring the state machine**

- Will still need to specify packet formats, but the state machine of the implementation is automatically inferred.
- Can manually or (semi-)automatically inspect the inferred state machine and then use it in BooFuzz or Peach.

# Black-box state inference

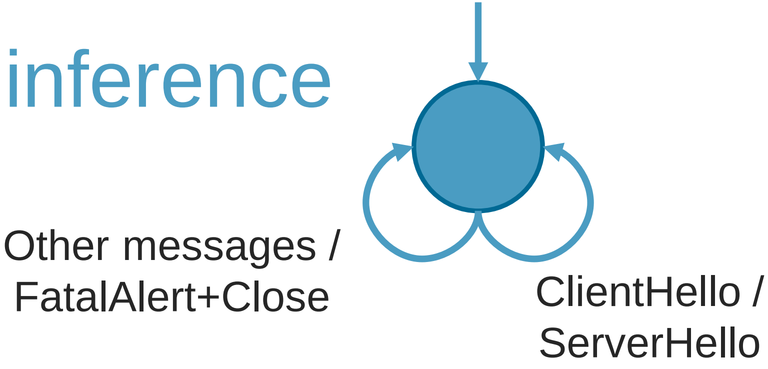
Common method is to use algorithms for automata learning

- **Actively** interact with the SUT to learn its behavior
- Send packets in a random order and inspect the responses

Infer the state machine based on the responses

- We can then inspect & use the state machine
- Successfully applied to discover bugs in TLS, SSH, WPA2,…

# Intuitive intro to state machine inference




Start with traces of length one:

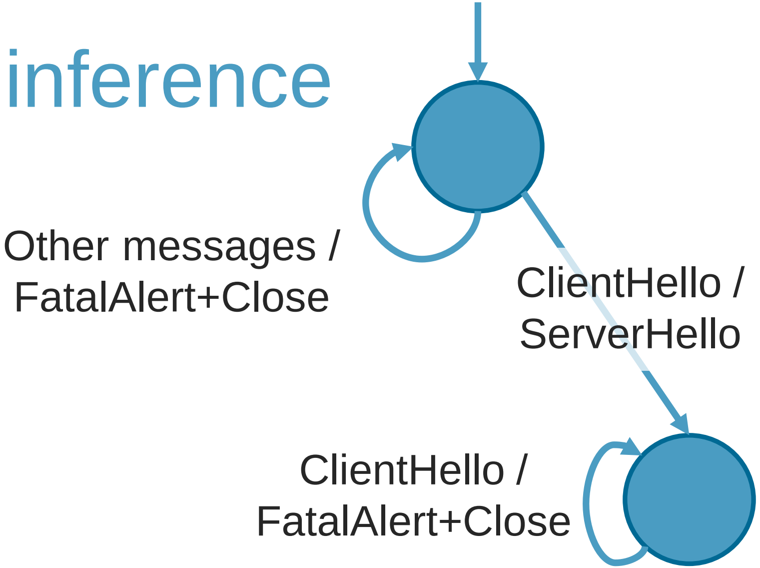
- ClientHello / ServerHello
- Update state machine
- Other packets / FatalAlert+Close
- Update state machine

Traces of length two:

- ClientHello / Server Helo, ClientHello / FatalAlert+Close
- Update state machine to handle this case

This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

# Intuitive intro to state machine inference




Start with traces of length one:

- ClientHello / ServerHello
- Update state machine
- Other packets / FatalAlert+Close
- Update state machine

Traces of length two:

- ClientHello / Server Helo, ClientHello / FatalAlert+Close
- Update state machine to handle this case

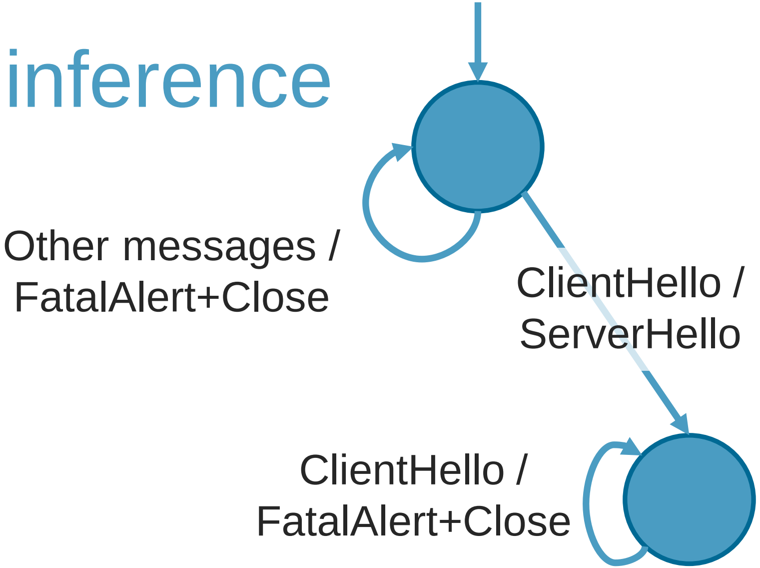


This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

# Intuitive intro to state machine inference

Continue with traces of length two:

- Other messages / FatalAlert+Close, Any message / empty

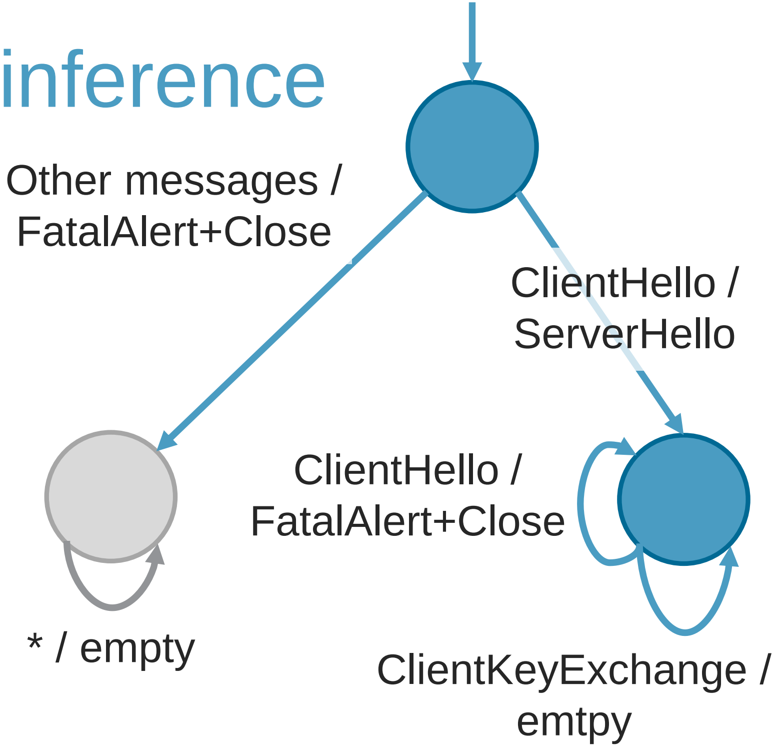



This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

# Intuitive intro to state machine inference

Continue with traces of length two:

- Other messages / FatalAlert+Close, Any message / empty
- ClientHello / ServerHello, ClientKeyExchange / empty
- ClientHello / ServerHello, Other messages / FatalAlert+Close

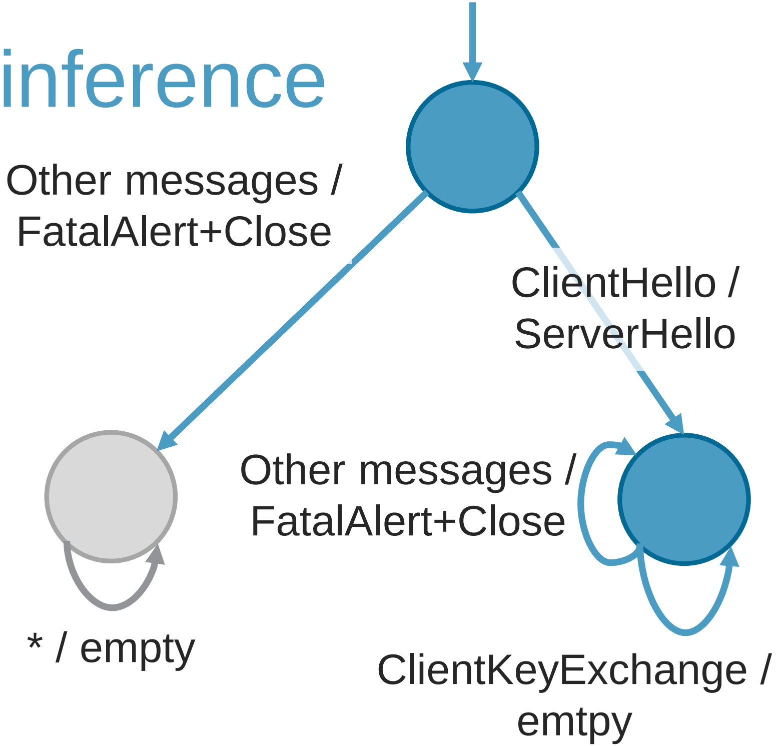



This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

# Intuitive intro to state machine inference

Continue with traces of length two:

- Other messages / FatalAlert+Close, Any message / empty
- ClientHello / ServerHello, ClientKeyExchange / empty
- ClientHello / ServerHello, Other messages / FatalAlert+Close




Continue with traces of length 3 & update state machine



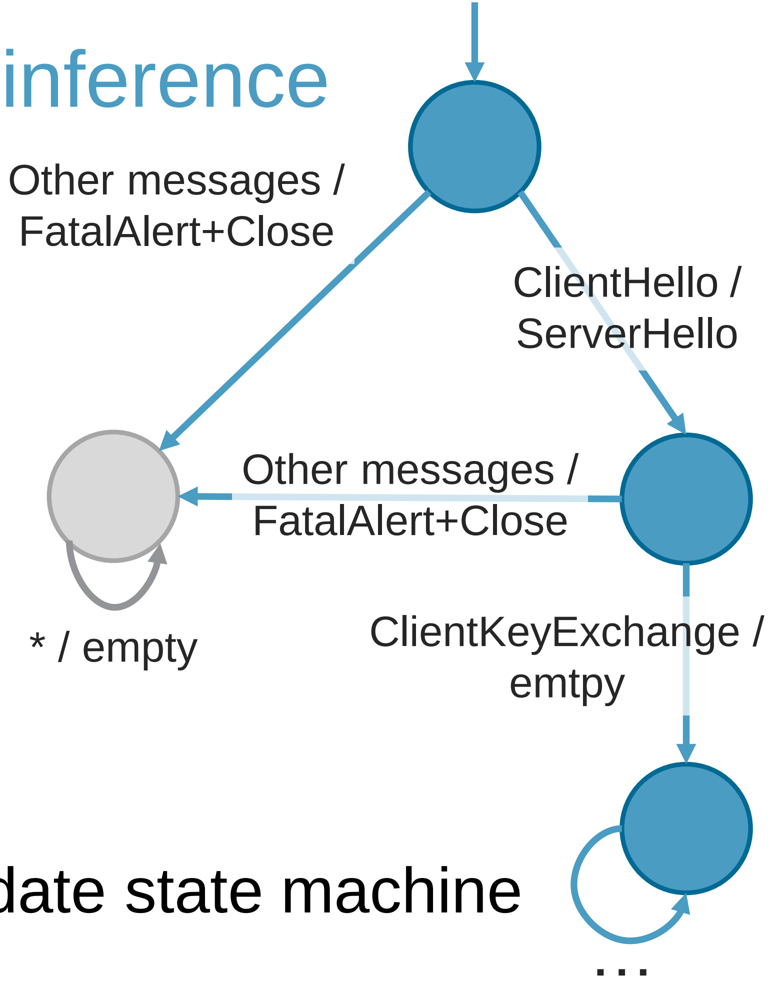
This example is based on “Protocol State Fuzzing of TLS Implementations” by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

# Intuitive intro to state machine inference

Continue with traces of length two:

- Other messages / FatalAlert+Close, Any message / empty
- ClientHello / ServerHello, ClientKeyExchange / empty
- ClientHello / ServerHello, Other messages / FatalAlert+Close

Continue with traces of length 3 & update state machine






This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

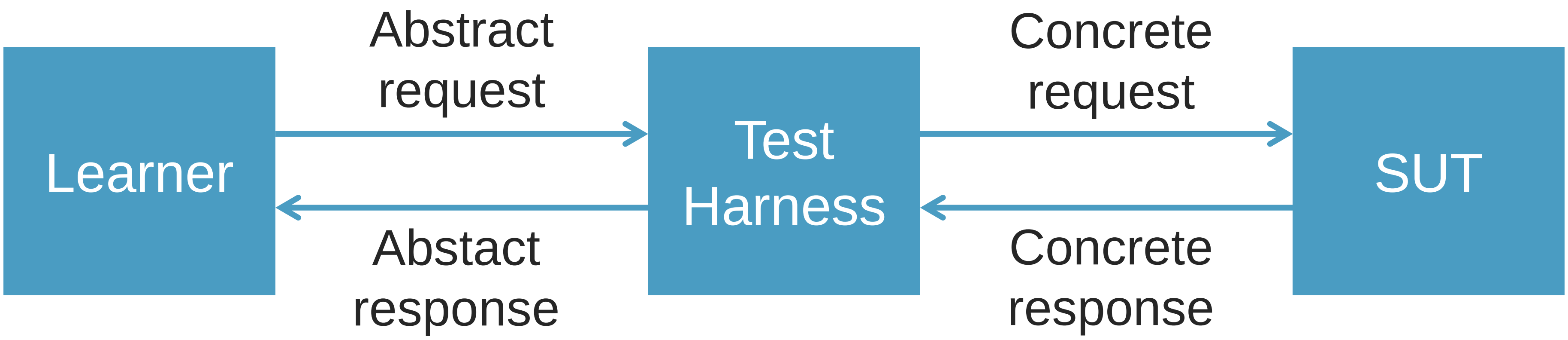
# The real learning setup

- What do we learn? A deterministic Mealy machine.
- How to learn? Use libraries such as LearnLib.
  - They use L* or TTT to from a hypothesis for the state machine.
- Must be able to perform three actions:
  1. Reset the SUT
  2. Send message to SUT & get the output
  3. Check whether the hypothesis (= current state machine) is correct
- Performing action 2 & 3 is non-trivial!

# Challenge: sending messages to the SUT (1)

State machine uses abstract messages, e.g., "`ClientHello`"

- Must be converted to concrete messages = actual bytes!
- A test harness is used for this conversion:



# Challenge: sending messages to the SUT (2)

The **state harness** must be able to send packets in any order

- Must consider previous messages that were sent/received
  - Example: random nonces that were part of the handshake
  - Example: currently negotiated session key
- In certain cases, it's unclear which values to use in requests
  - Example: how to send an encrypted TLS record before a key was negotiated? Use a random key? Use an all-zero key?
- And we must be able to **receive packets in any order**
  - Example edge case: we receive an encrypted packet before a key was negotiated. Do we try to decrypt it? With which key?

# Challenge: is the state machine correct?

Learning algorithms need a way to check if their current hypothesis for the sate machine is correct

- But we don’t know the state machine…

Two typical solutions:

1. **Random traces**: send some fixed number of random traces and see if the responses match the state machine
2. **Chow’s W-method**: guarantees correctness of the state machine given an upper bound on the number of states

# Why is state inference useful?

Use the inferred state machine in **BooFuzz or similar**

- You will now fuzz the *actual* states of the implementation
- This may be more/other/different states than in the standard!

**Manually inspect** the state machine for flaws

- Identified flaws in TLS, DTLS, WPA2, and 4G/LTE

State machine may form a **fingerprint** of the implementation

- Use unique behavior to detect implementation being used

# Last but not least: frameworks for manual tests

Sometimes you want to manually test specific behavior

- To reimplement a known attack quickly…
- You think a library has a specific vulnerability…
- You want to confirm a flaw in the standard…

This may require implementing large parts of a protocol

- E.g., might require implementing tedious crypto algorithms
- Or requires implementing the full handshake if you want to test behavior after authenticating

# TLS testing framework

Use TLS-Attacker framework for TLS tests:

- Java-based framework for analyzing TLS libraries
- Easily define custom TLS protocol flows to test libraries

```java
Config config = Config.createConfig();
WorkflowTrace trace = new WorkflowTrace();
trace.addTlsAction(new SendAction(new ClientHelloMessage()));
trace.addTlsAction(new ReceiveAction(new ServerHelloMessage()));
State state = new State(config, trace);
DefaultWorkflowExecutor ex = new DefaultWorkflowExecutor(state);
ex.executeWorkflow();
```

# Testing tools in practice: Wi-Fi security

Wi-Fi Alliance **tests for KRACK** during device certification (2017):

> industry. There is no evidence of the vulnerability being used against Wi-Fi users maliciously, and Wi-Fi Alliance has taken immediate steps to ensure users can continue to count on Wi-Fi to deliver strong security protections.
>
> - Wi-Fi Alliance now requires testing for this vulnerability within our global certification lab network
> - Wi-Fi Alliance has provided a **vulnerability detection tool** for use by any Wi-Fi Alliance member

Source: https://web.archive.org/web/20220306005730/https://www.wi-fi.org/security-update-october-2017

# Testing tools in practice: Wi-Fi security

Wi-Fi Alliance also ***seems* to test for FragAttacks** flaws (2021):

> “Wi-Fi Alliance has taken immediate steps to ensure users can remain confident in the strong security protections provided by Wi-Fi.
>
> - Wi-Fi CERTIFIED now includes additional testing within our global certification lab network to encourage greater adoption of recommended practices
> - Wi-Fi Alliance is broadly communicating implementation guidance to device vendors [..] ”

Source: https://web.archive.org/web/20220504234741/https://www.wi-fi.org/security-update-fragmentation

# Optional reading

- “Protocol State Fuzzing of TLS Implementations” by Joeri de Ruiter and Erik Poll, USENIX Security 2015