Security Testing of Network Protocol Implementations

Prof. Mathy Vanhoef - @vanhoefm

Summer School on Security Testing and Verification

20 September 2022, Leuven, Belgium

imec DistriNet KU LEUVEN

Introduction

Today we 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;
  }
}
```

Security testing of stateful protocols

A common technique for security testing is **fuzzing**:

- Give unexpected or random input to the program
- Then monitor for crashes, failed assertions, memory leaks,…

Tested program is called the **SUT (System Under Test)**

- For us, the SUT is a network protocol implementation
- The tool/component that sends (in)valid messages to the SUT will be called the **test harness**

Why fuzzing?

Why is fuzzing useful in practice?

- Programs **often only undergo functional testing**, i.e., they are tested to handle expected inputs
- Want to test how programs will react to **unexpected inputs**, since incorrectly handled input can cause vulnerabilities!

Fuzzing is frequently used in practice:

- AFL, LibFuzzer, Honggfuzz, Boofuzz, and many more…

Fuzzing recently had many successes

- Google discovered more than 25,000+ bugs in Chrome...
  …and 36,000+ bugs in more than 550 open-source projects
- Using SAGE saved Microsoft millions of dollars while creating Windows 7
- The 2016 DARPA Cyber Grand Challenge winner, Mayhem, heavily relied on white-box fuzzing to find vulnerabilities

Sources:

- https://google.github.io/clusterfuzz/#trophies
- "Automated whitebox fuzz testing" by . P. Godefroid, M. Y. Levin, and D. Molnar
- http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html
- "Unleashing Mayhem on binary code" by S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley

Well-known example: American Fuzzy Lop (AFL)

Partly a “dumb” fuzzer:

- No model of input. Uses set of seed inputs.
- Given a test input, it changes random bits.

Partly a “smart” fuzzer:

- Tracks which code in the SUT was executed.
- Adds input covering new code to the set of interesting inputs

= **coverage-guided fuzzing** = combination of dumb & smart

Example: fuzzing jpeg

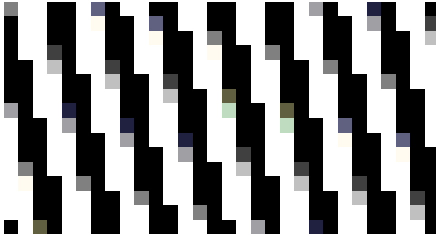
- Start with a single seed input: “hello”
- Using 7 cores for ~28 hours (i7 2.6 GHz)
- Interesting inputs are discovered, but not yet a valid jpeg file

```
$ ./djpeg id:002,op:havoc,rep:32,+cov
Premature end of JPEG file
Not a JPEG file: starts with 0xff 0xff

$ ./djpeg id:003,+cov
Premature end of JPEG file
JPEG datastream contains no image
```

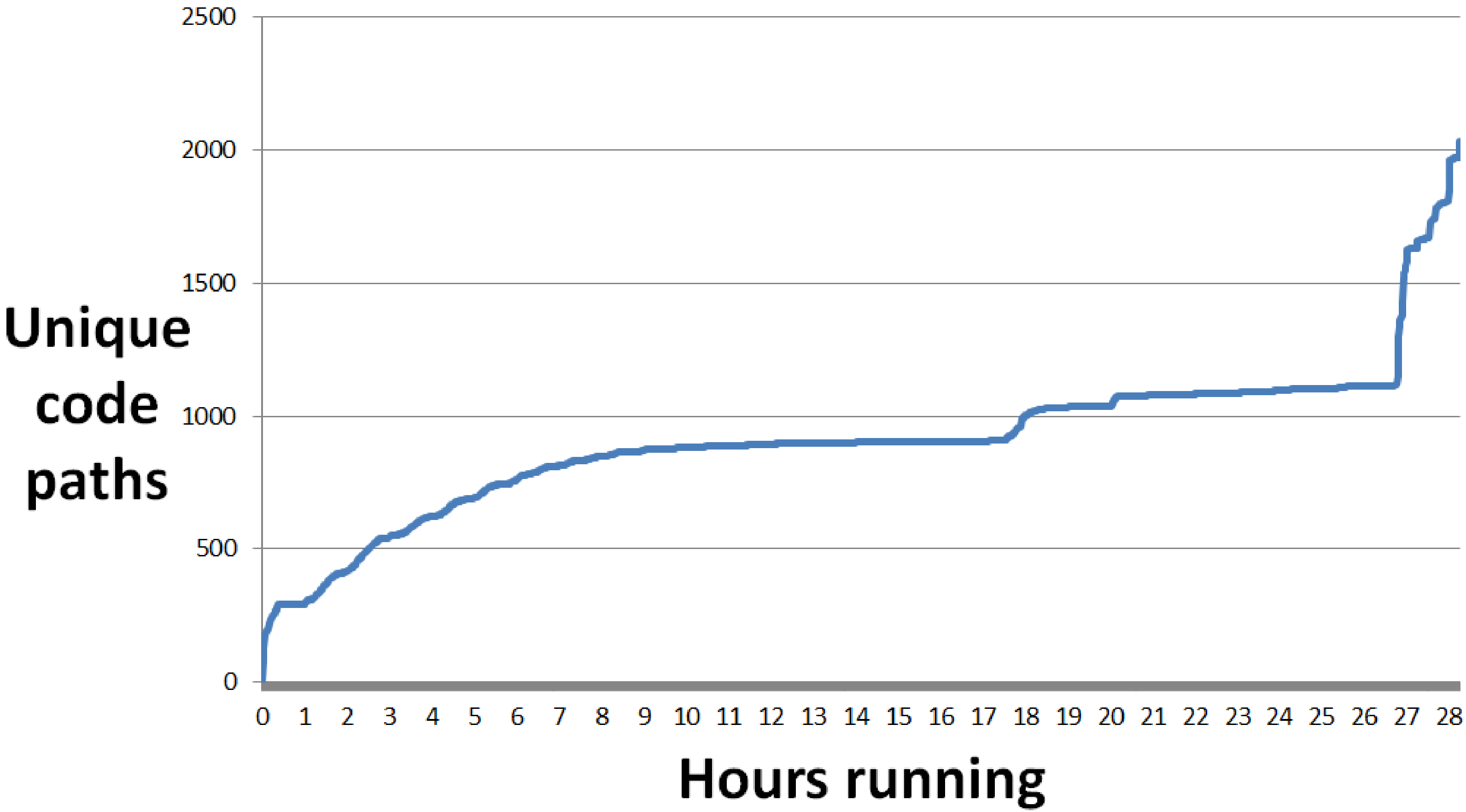
Example: fuzzing jpeg

```
$ ./djpeg id:000840,sync:fuzzer04
Corrupt JPEG data: 50 extraneous bytes before marker 0xc4
Bogus Huffman table definition

$ ./djpeg id:001032,sync:fuzzer06
Corrupt JPEG data: 2 extraneous bytes before marker 0xc9
Quantization table 0x31 was not defined
```

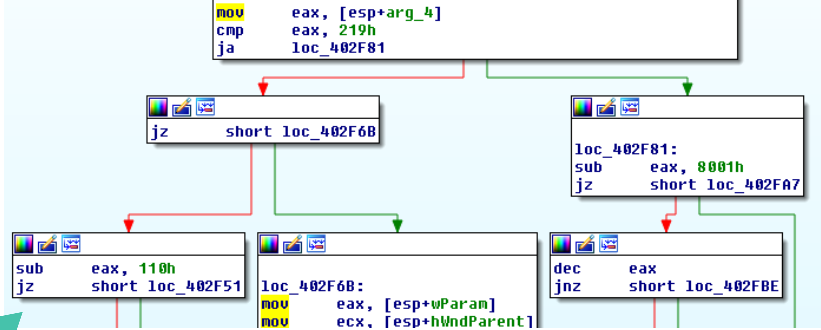
Suddenly valid jpeg's are generated!



Downside: this can take a long time



How does AFL achieve this?



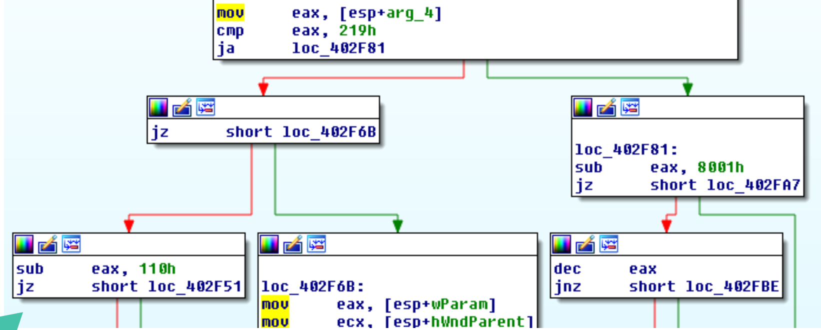
It monitors which code is executed

- Doesn't track the actual code path
- Tracks how many times a branch was taken

Example 1:

Path:	**Branches:**
A → B → C → D → E	AB, BC, CD, DE
A → B → D → C → E	AB, BD, DC, CE

How does AFL achieve this?



It monitors which code is executed

- Doesn't track the actual code path
- Tracks how many times a branch was taken

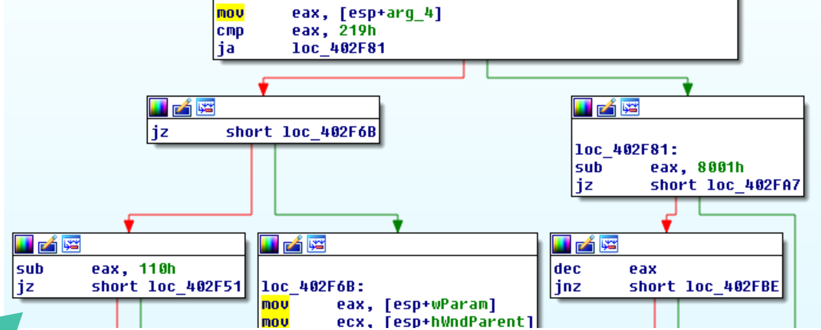
Example 2:

More hits = different path

Path:	**Branches:**
A → B → A → C	AB, BA, AC
A → B → A → B → A → C	AB, BA, AC

KU LEUVEN

How does AFL achieve this?



It monitors which code is executed

- Doesn't track the actual code path
- Tracks how many times a branch was taken

Queue of "interesting" inputs:

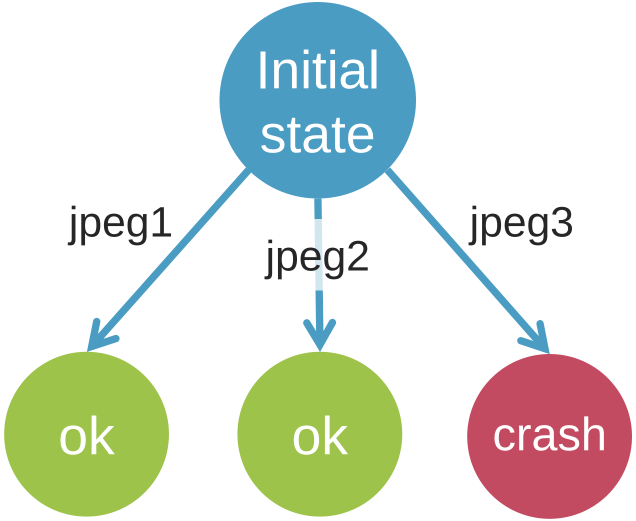
- Take an input from this queue and mutate it
- If new path is taken, add mutation to the queue

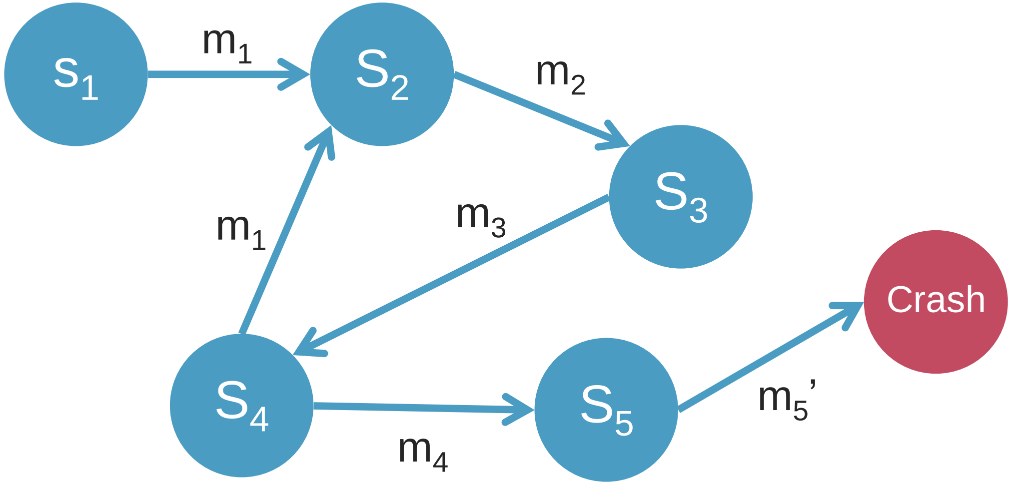
→ AFL slowly "explores" functionality of program

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





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:

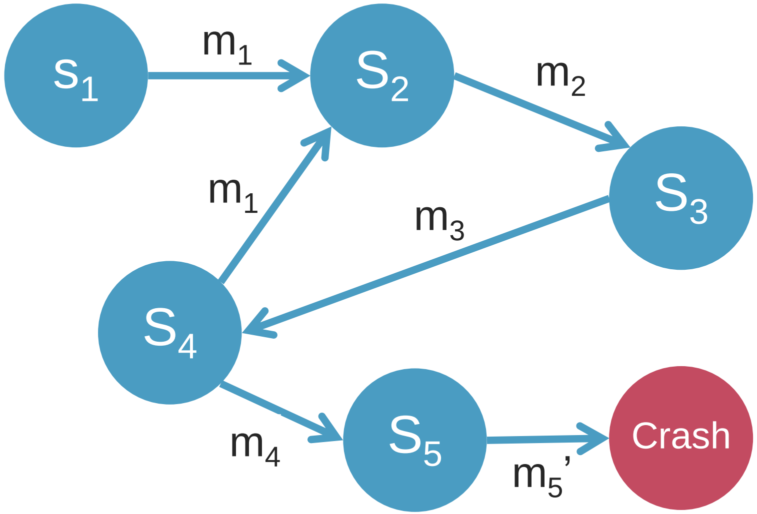
- Test harness
- SUT
- Anomaly detector

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

Grammar-based fuzzing

Define packet layout and state machine

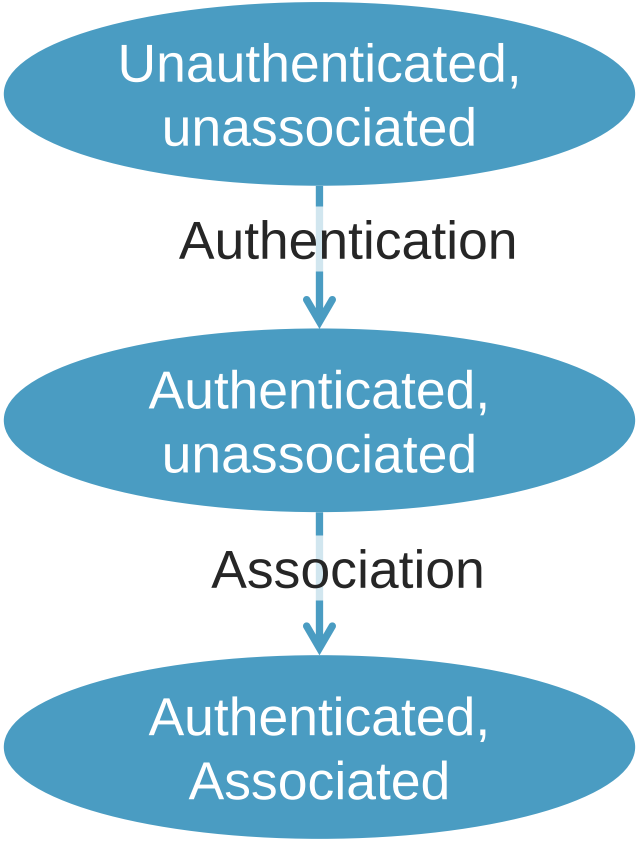
› Fuzzer then sends valid packets to reach a target state
› When in the target state, send malformed/mutated packets



Example:

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

Simple example: early Wi-Fi fuzzing



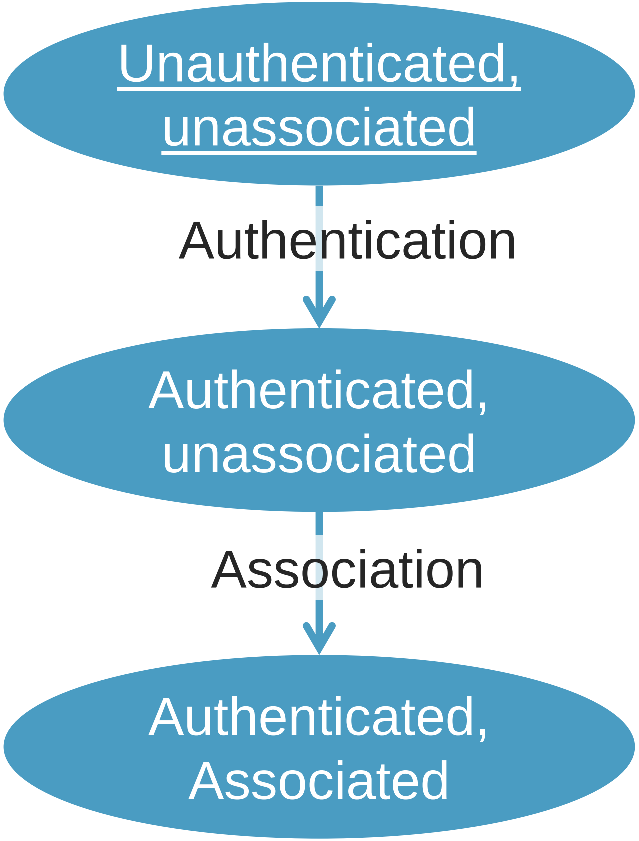
Option 1: fuzz each state

- Must define state machine and packet formats to mutate

Option 2: only fuzz the first state

- 1st state can be reached by attackers
- Makes fuzzing easier: only need to define packet formats to mutate…
- …but doesn't cover all the code

Simple example: early Wi-Fi fuzzing



Butti and Tinnès fuzzed the first state

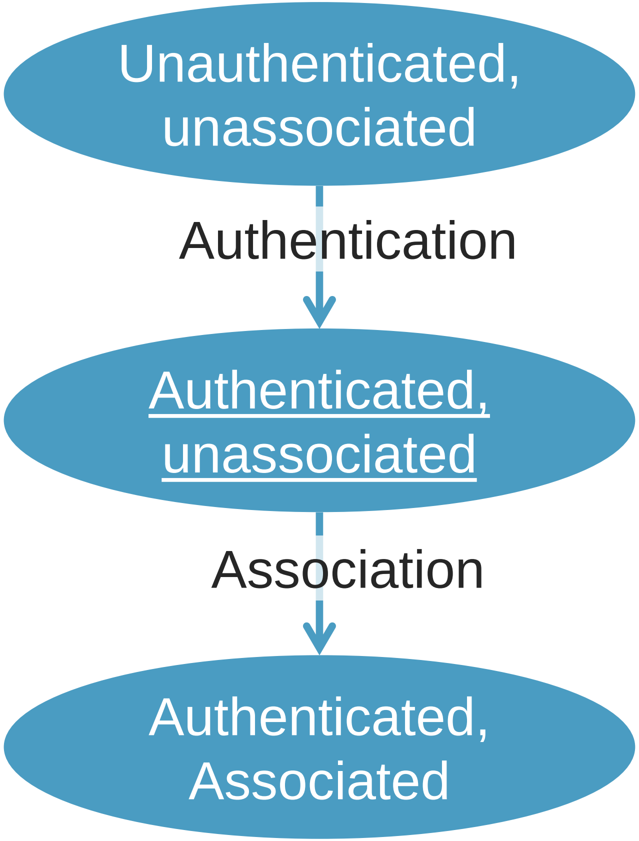
- Fuzzed **probe responses & beacons**: these are processed while scanning
- Basic packet layout is defined in Scapy
- Random fields are mutated

Discovered multiple crashes

- Remote code execution in the kernel!

Source: “Discovering and exploiting 802.11 wireless driver vulnerabilities” by Laurent Butti and Julien Tinnès.

Simple example: early Wi-Fi fuzzing



Keil and Kolbitsch fuzzed the 2^{nd} state

- Let the client authenticate but not yet associate
- Then transmitted fuzzed frames, i.e., send fuzzed association responses

Discovered one new vulnerability

- No remote code execution though

Source: “Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment” at Black Hat Japan (2007).

Side-note: these issues were very common (2007)

CVE-2007-1218 (PARSER)	Off-by-one buffer overflow in the parse_elements function in the 802.11 printer code (print-802_11.c) for tcpdump 3.9.5 and earlier allows remote attackers to cause a denial of service (crash) via a crafted 802.11 frame. NOTE: this was originally referred to as heap-based, but it might be stack-based.
CVE-2007-0933 (DRIVER/WIN)	Will be released today
CVE-2007-0686 (DRIVER/WIN)	The Intel 2200BG 802.11 Wireless Mini-PCI driver 9.0.3.9 (w29n51.sys) allows remote attackers to cause a denial of service (system crash) via crafted disassociation packets, which triggers memory corruption of "internal kernel structures," a different vulnerability than CVE-2006-6651. NOTE: this issue might overlap CVE-2006-3992.
CVE-2007-0457 (PARSER)	Unspecified vulnerability in the IEEE 802.11 dissector in Wireshark (formerly Ethereal) 0.10.14 through 0.99.4 allows remote attackers to cause a denial of service (application crash) via unspecified vectors.
CVE-2006-6651 (DRIVER/WIN)	Race condition in W29N51.SYS in the Intel 2200BG wireless driver 9.0.3.9 allows remote attackers to cause memory corruption and execute arbitrary code via a series of crafted beacon frames. NOTE: some details are obtained solely from third party information.
CVE-2006-6332 (DRIVER/LIN)	Stack-based buffer overflow in net80211/ieee80211_wireless.c in MadWifi before 0.9.2.1 allows remote attackers to execute arbitrary code via unspecified vectors, related to the encode_ie and giwscan_cb functions.
CVE-2006-6125 (DRIVER/WIN)	Heap-based buffer overflow in the wireless driver (WG311ND5.SYS) 2.3.1.10 for NetGear WG311v1 wireless adapter allows remote attackers to execute arbitrary code via an 802.11 management frame with a long SSID.
CVE-2006-6059 (DRIVER/WIN)	Buffer overflow in MA521nd5.SYS driver 5.148.724.2003 for NetGear MA521 PCMCIA adapter allows remote attackers to execute arbitrary code via (1) beacon or (2) probe 802.11 frame responses with an long supported rates information element. NOTE: this issue was reported as a "memory corruption" error, but the associated exploit code suggests that it is a buffer overflow.
CVE-2006-6055 (DRIVER/WIN)	Stack-based buffer overflow in A5AGU.SYS 1.0.1.41 for the D-Link DWL-G132 wireless adapter allows remote attackers to execute arbitrary code via a 802.11 beacon request with a long Rates information element (IE).
CVE-2006-5972 (DRIVER/WIN)	Stack-based buffer overflow in WG111v2.SYS in NetGear WG111v2 wireless adapter (USB) allows remote attackers to execute arbitrary code via a long 802.11 beacon request.

Side-note: these issues were very common (2007)

CVE-2006-5882 (DRIVER/WIN)	Stack-based buffer overflow in the Broadcom BCMWL5.SYS wireless device driver 3.50.21.10, as used in Cisco Linksys WPC300N Wireless-N Notebook Adapter before 4.100.15.5 and other products, allows remote attackers to execute arbitrary code via an 802.11 response frame containing a long SSID field.
CVE-2006-5710 (DRIVER/OSX)	The Airport driver for certain Orinoco based Airport cards in Darwin kernel 8.8.0 in Apple Mac OS X 10.4.8, and possibly other versions, allows remote attackers to execute arbitrary code via an 802.11 probe response frame without any valid information element (IE) fields after the header, which triggers a heap-based buffer overflow.
CVE-2006-3992 (DRIVER/WIN)	Unspecified vulnerability in the Centrino (1) w22n50.sys, (2) w22n51.sys, (3) w29n50.sys, and (4) w29n51.sys Microsoft Windows drivers for Intel 2200BG and 2915ABG PRO/Wireless Network Connection before 10.5 with driver 9.0.4.16 allows remote attackers to execute arbitrary code via certain frames that trigger memory corruption.
CVE-2006-3509 (DRIVER/OSX)	Integer overflow in the API for the AirPort wireless driver on Apple Mac OS X 10.4.7 might allow physically proximate attackers to cause a denial of service (crash) or execute arbitrary code in third-party wireless software that uses the API via crafted frames.
CVE-2006-3508 (DRIVER/OSX)	Heap-based buffer overflow in the AirPort wireless driver on Apple Mac OS X 10.4.7 allows physically proximate attackers to cause a denial of service (crash), gain privileges, and execute arbitrary code via a crafted frame that is not properly handled during scan cache updates.
CVE-2006-3507 (DRIVER/OSX)	Multiple stack-based buffer overflows in the AirPort wireless driver on Apple Mac OS X 10.3.9 and 10.4.7 allow physically proximate attackers to execute arbitrary code by injecting crafted frames into a wireless network.
CVE-2006-1385 (PARSER)	Stack-based buffer overflow in the parseTaggedData function in WavePacket.mm in KisMAC R54 through R73p allows remote attackers to execute arbitrary code via multiple SSIDs in a Cisco vendor tag in a 802.11 management frame.
CVE-2006-0226 (DRIVER/BSD)	Integer overflow in IEEE 802.11 network subsystem (ieee80211_ioctl.c) in FreeBSD before 6.0-STABLE, while scanning for wireless networks, allows remote attackers to execute arbitrary code by broadcasting crafted (1) beacon or (2) probe response frames.

Source: “Wi-Fi Advanced Fuzzing” by Laurent Butti at BlackHat Europe 2007.

Other grammar-based fuzzers

BooFuzz protocol fuzzer (fork/successor of Sulley)

1. The structure of each message is first defined

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

Name of the message

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"),
))
```

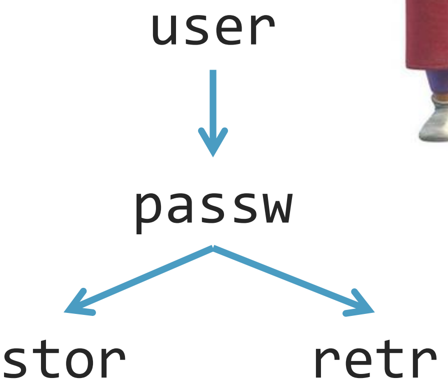
Note: these block-based grammars are inspired by the (underdocumented?) SPIKE fuzzer

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.

KU LEUVEN

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

Downsides of previous fuzzers:

- Need to specify packet formats *and* state machine
- The state machine itself isn't fuzzed (i.e., order of packets)

This 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 inspect the inferred state machine and then use it for stateful fuzzing.

Black-box state inference

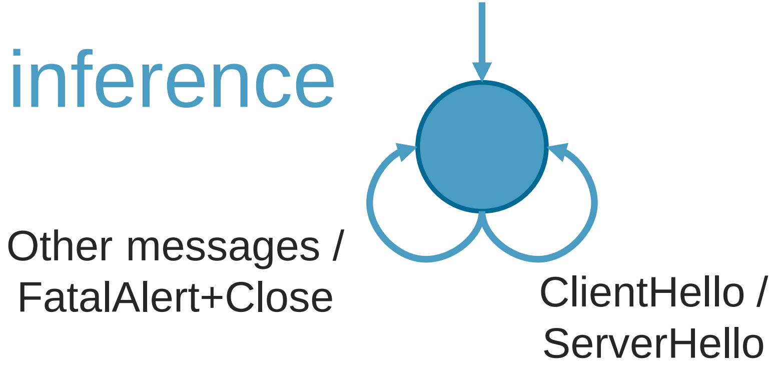
Common method is to use algorithms for automata learning

- **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:

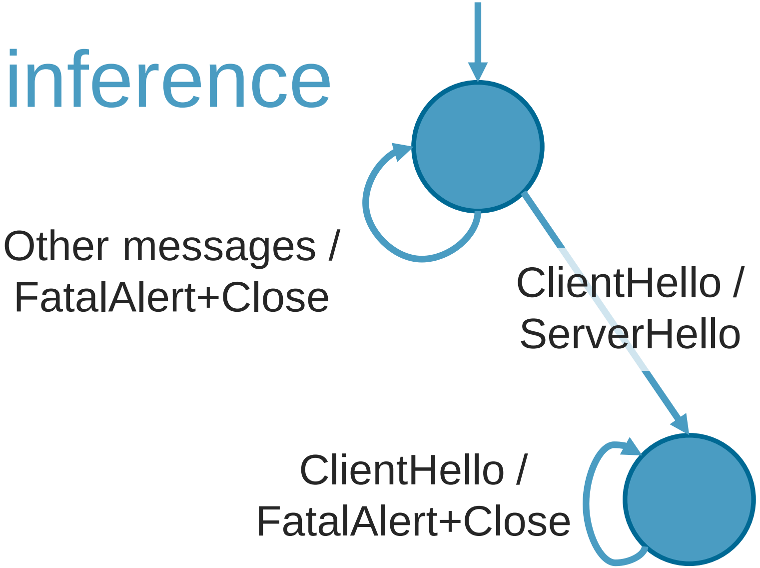
- ClientHello / Server Helo, ClientHello / FatalAlert+Close
- Update state machine to handle this case

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:

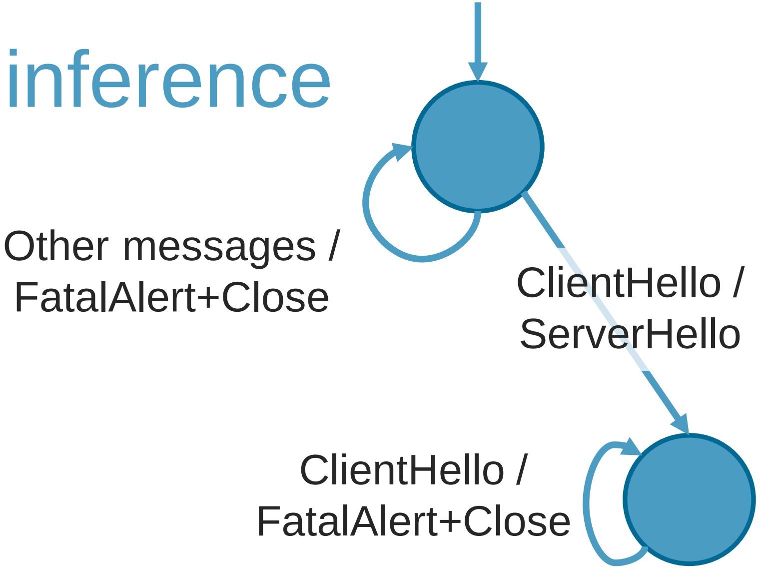
- ClientHello / Server Helo, ClientHello / FatalAlert+Close
- Update state machine to handle this case

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

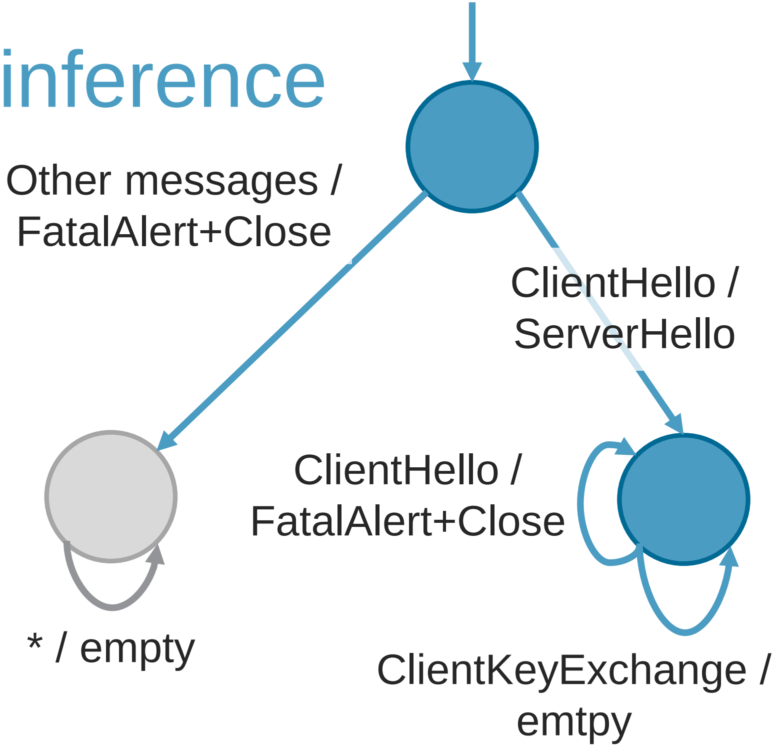


This example is based on “Protocol State Fuzzing of TLS Implementations” by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

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

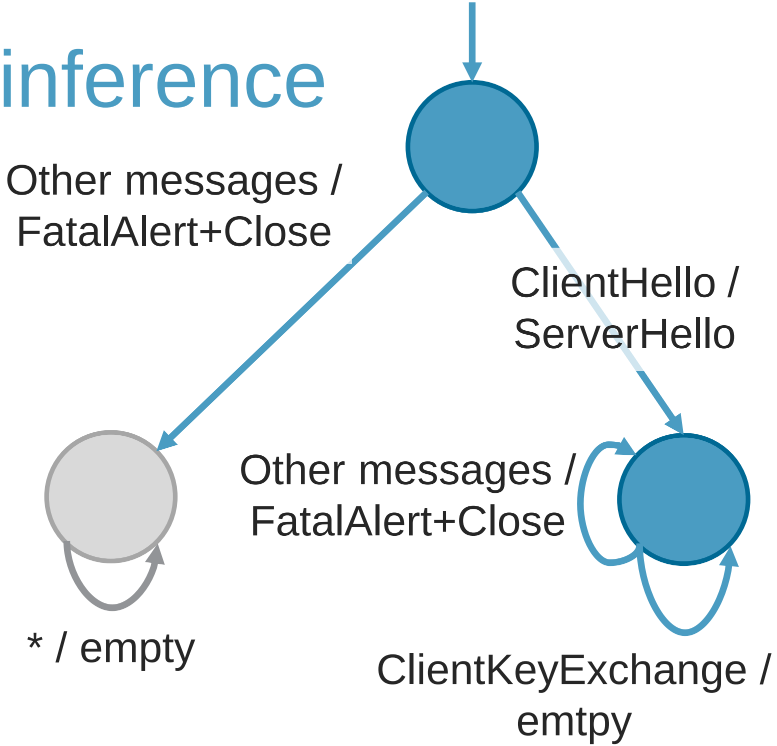


This example is based on "[Protocol State Fuzzing of TLS Implementations]" by Joeri de Ruiter and Erik Poll, USENIX Security 2015.

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



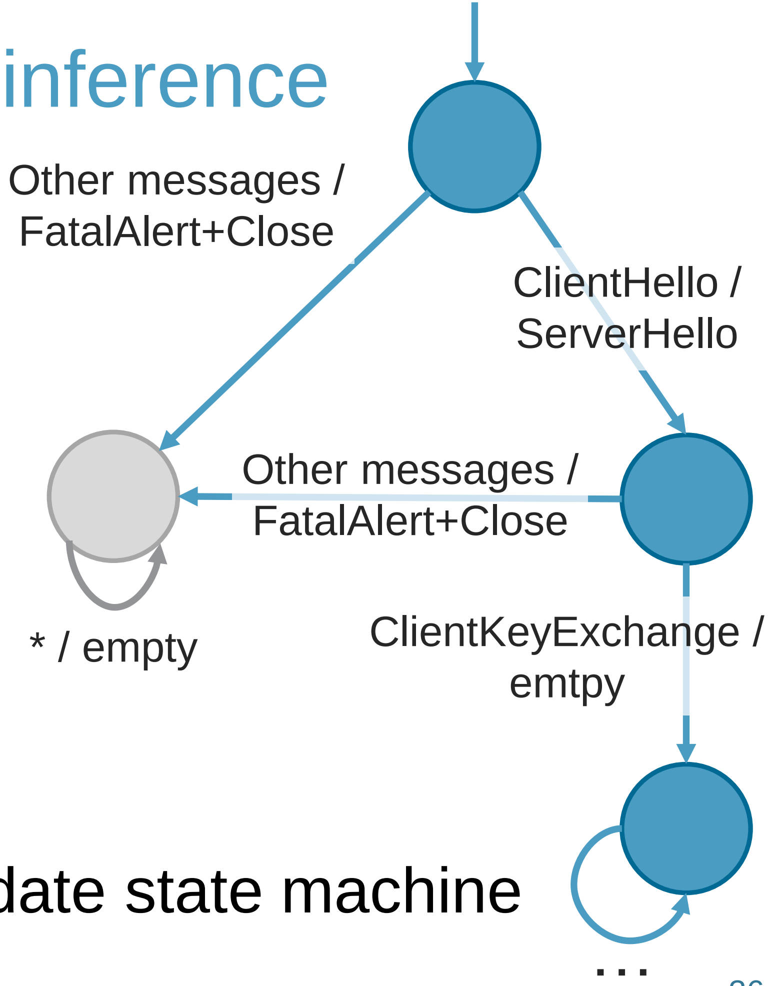
Continue with traces of length 3 & update state machine

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.

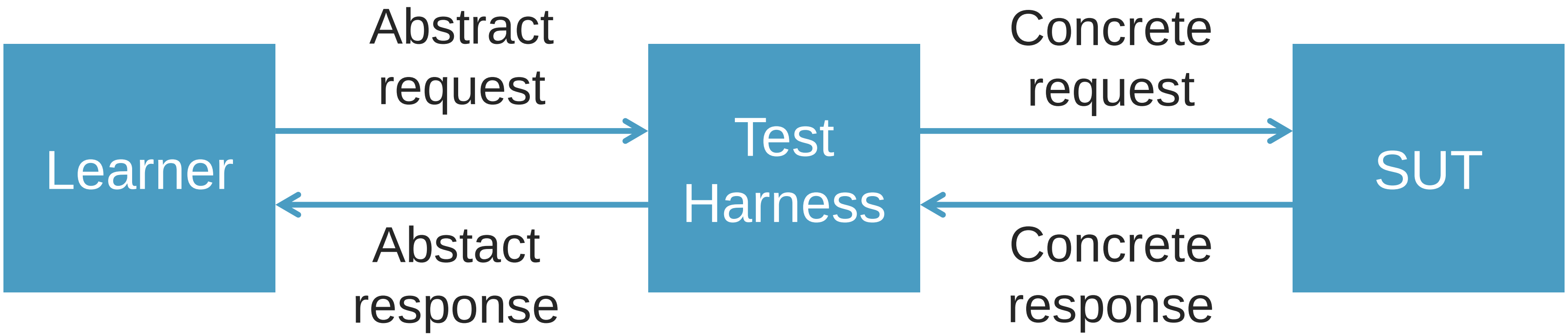
The real learning setup

- 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?

Manually inspect the state machine for flaws

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

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!

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

- Use unique behavior to detect implementation being used

Evolutionary protocol fuzzers

Previous approach are black-box fuzzers

- What if we have access to the binary and/or source code?

We can use coverage-guided fuzzing!

- = detect **interesting inputs** that execute **new code**
- Recent approach is to **modify AFL** to fuzz network protocols

Examples: SNPSFuzzer, AFLNet, SnapFuzz, and so on.

- We will discuss their **high-level strategies**

How to use AFL on network services?

Write **unit-level tests** that interact with the software using their (public or internal) APIs

- Used by Google's OSS-Fuzz. Requires a lot of manual effort.
- Usually only a single state is fussed in each unit-level test

Need to **handle side-effects**

- Some protocols, such as FTP, write data to the file system or exchange network messages.
- Need to reset these side-effects on every new input

Desired properties of the fuzzer

We want to avoid writing unit-level tests

- Avoid manual overhead of modifying the SUT

Automatically detect and explore states

- Fuzzer should be able to (heuristically) detect new states

Our solution should be fast and efficient

- The more input we can test/second, the more bugs we found

High-level: the stateful grey-box fuzzing loop

1. Select the most interesting state from a **state chain queue** to be fuzzed → the fuzzer model the state machine.
2. Select a message from the **message queue** to mutate.
3. Pick a **mutating strategy** to mutate the message.
4. Put the SUT in the desired state & send mutated message
5. Check if new a new state is reach or new code is executed. If so, **update the state chain or message queue**.
6. Go back to step 1!

Based on “SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols using Snapshots” by Junqiang Li et al. (2022).

KU LEUVEN

Question: how to detect new states?

AFLNet and SnapFuzz

- Use **response code** in the protocol to infer the SUT's state
- Example: FTP, IRC, or HTTP status codes

Stateful Greybox Fuzzing (USENIX Security 2022)

- Detected state **variables in the source code** (e.g., `enum`'s)

StateInspector (CCS 2023) and StateAFL (arXiv)

- Grey-box method to detected **state-defining memory**

Remaining problem: fuzzing is slow

SNPSFuzzer by Junqiang Li et al.

- Dumps process context when the SUT is in a specific state
- Can now quickly **restore this state** without sending packets

SnapFuzz by Andronidis and Cadar:

- Use an **in-memory filesystem** to easily clean-up side-effects
- Further improve **efficiency of the forkserver** to more quickly reach the point at which mutated packets can be input

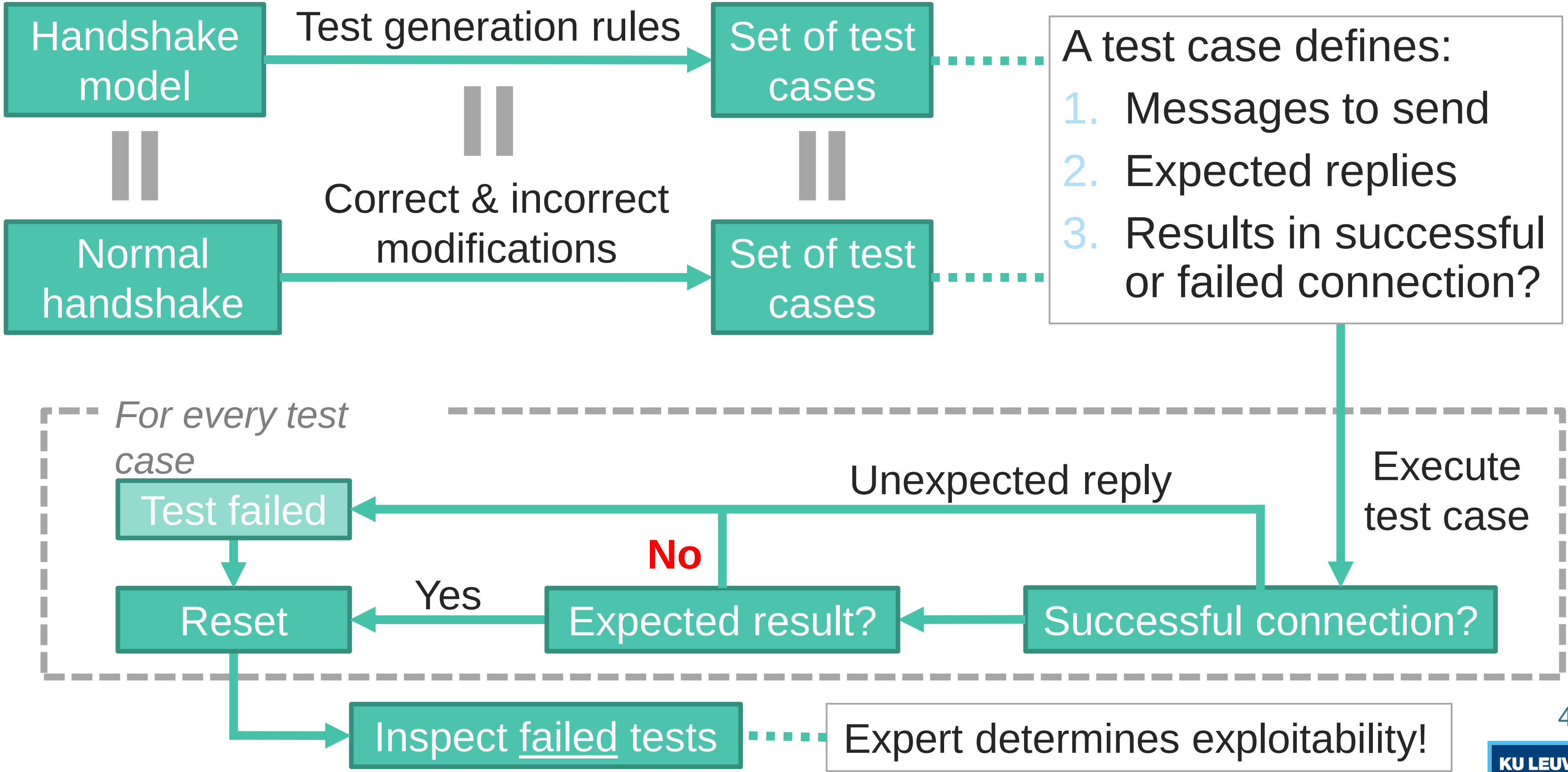
How to find logical vulnerabilities?

Model-based testing!

- Test if program behaves according to some abstract model
- Proved successful against TLS and Wi-Fi

→ We will focus on Wi-Fi work (AsiaCCS’17)

Model-based testing: our approach



Test generation rules

Test generation rules manipulating messages as a whole:

1. Drop a message
2. Inject/repeat a message

Test generation rules that modify fields in messages:

› Can use various mutating strategies depending on the protocol that is being tested!

Evaluation on 12 access points

- Open source: OpenBSD, Linux’s Hostapd
- Leaked source: Broadcom, MediaTek (home routers)
- Closed source: Windows, Apple, Telenet
- Professional equipment: Aerohive, Aironet

Can **discover logical flaws**:

- Two downgrade attacks
- Multiple denail-of-Service flaws
- Several fingerprinting methods

Thank you!
Questions?