

# Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment

Sylvester Keil\*

Clemens Kolbitsch<sup>†</sup>

Secure Systems Lab, Technical University Vienna in cooperation  
with SEC Consult Unternehmensberatung GmbH

## Abstract

*This paper documents the process of identifying potential vulnerabilities in IEEE 802.11 device drivers through fuzzing. The relative complexity of 802.11 as compared to other layer two protocols imposes a number of non-trivial requirements on regular 802.11 protocol fuzzers. This paper describes a new approach to fuzzing 802.11 device drivers on the basis of emulation. First, the process of creating a virtual 802.11 device for the processor emulator QEMU is described. Then, the development of a stateful 802.11 fuzzer based on the virtual device is discussed. Finally, we report the results of fuzzing the Atheros Windows XP driver, as well as the official and open source MADWifi drivers.*

## 1. Introduction

“People move. Networks don’t.” With this statement, Matthew Gast opens his *Definitive Guide* to 802.11 wireless networks. It is true that in recent years wireless networks have evolved into a mainstream technology. An ever increasing mobility in computing is sustained by the solid growth of a wireless infrastructure through the prolificacy of wireless networks and the prevalence of wireless technology in computer hardware. However, new developments are always associated with new risks, and advantages such as increased mobility and distribution obviously come at a cost. Due to their physical characteristics, wireless networks are almost per definition prone to security and privacy issues. Hence, several efforts to provide confidentiality have been made. Typically, these efforts are cryptographic approaches to prohibit eavesdropping, unauthorized use, traffic injection and traffic analysis. However, a medium in which all parties communicate in the same space is inherently vulnerable to other vectors of attack that directly target a victim’s wireless device driver software.

Traditionally, device drivers have always been an

Achilles heel of operating systems: They are often proprietary and supplied by third parties. Furthermore, because they run in unprotected kernel-mode, potential vulnerabilities are particularly dangerous. Moreover, drivers of wireless devices are jeopardized even further by an underlying technology that exposes them to virtually everyone in their physical vicinity. At the same time, there appears to be a lack of awareness of this exposed nature of wireless devices and the danger inherent therein, because these drivers operate in lower levels of the OSI reference model (i.e. they are mostly transparent to widespread security and monitoring tools that are often only concerned with OSI levels three and above).

This paper describes a *fuzz-test-suite* to evaluate the implementation of the IEEE 802.11 MAC specification of wireless device drivers. *Fuzz testing* — or simply *fuzzing* — is a software testing technique developed at the University of Wisconsin-Madison that has proved particularly effective in detecting potential security vulnerabilities in protocol implementations. For example, a piece of software may be penetrated with a large number of frame contents, some of which may be unexpected.

While fuzzing basic 802.11 functions such as beacon frames<sup>1</sup> can be handled quite easily by simply injecting frames on the medium at certain intervals, many advanced wireless functions such as network authentication or active scanning<sup>2</sup> introduce problems that have not been dealt with in previous wireless fuzz test tools.

Because it is possible for multiple networks to operate on the same medium, wireless network devices rely strictly on receiving responses or acknowledgments to packets they sent out before a certain timeout. This time interval is so small, however, that monitoring the medium, analyzing in-

---

<sup>1</sup>In order to inform wireless stations of their presence, wireless access points publish this type of packet at regular intervals. The information includes, among others, the *SSID* or name of the network, the frequency used for conversation and data rates supported by the device.

<sup>2</sup>A wireless device that is scanning in active mode does not rely solely on received beacon frames but additionally publishes so-called probe requests. Access points then reply with a probe response that includes information very similar to the previously mentioned beacon frame.

---

\*sk@seclab.tuwien.ac.at

<sup>†</sup>ck@seclab.tuwien.ac.at

coming frames and injecting appropriate responses cannot be done reliably. Previous approaches, such as those mentioned in Section 1.1, thus solely focus on frames that are sent out at regular intervals, ignoring a considerable portion of possible frame types.

In this paper, we present a novel approach to testing wireless device drivers and fuzzing in general that replaces the underlying wireless device hardware with an emulation software. Thus we gain full control over the device driver. Our work is built on top of QEMU[1], a generic and open source system emulator, enhanced with a virtual network device that reacts to commands by the device driver, halts the virtual CPU and afterwards informs connected fuzz modules of the event.

By stopping the execution, the fuzzing module obtains the time necessary to generate an appropriate answer that is injected into the emulated hardware, followed by the resumption of the emulation. Obviously, this approach solves any timing difficulties inherent to traditional 802.11 fuzzing-techniques and introduces even further practical advantages that are discussed in more detail in the following sections.

## 1.1. Previous Work

A recent publication of Laurent Butti demonstrated the effectiveness of fuzzing wireless device drivers. His work concentrated on, what we later define as, *stateless* communication by flooding the wireless medium with a large amount of packets when a target system starts scanning for available networks.

In his publication, Butti identifies two major drawbacks with this technique[7]:

1. One *must* answer very fast as the client performs channel hopping, and
2. One *cannot* be sure if the frame was analyzed or not by the driver.

As we show in this paper, these two drawbacks can be overcome with our novel fuzzing approach.

## 2. 802.11 Frames

Wireless network frames can be divided into a wide range of different types, whose explicit meaning and usage is quite different from each other. Their internal message format stays the same for all types, however. Like higher level protocols, 802.11 frames are divided into a header and a body section, whereas the header section, similar to IP datagrams, includes

- Type/subtype information
- Sender, receiver and network hardware address

- Sequencing and other control information.

The body section varies depending on the *type* field that further classifies individual frames into three categories:

### Control frames

Control frames are used to manage who is allowed to use the medium for packet transmission, to manage the distribution of the network and other advanced functions such as power saving. Because of its simple and (more importantly) fixed length structure, this type of frame is not particularly interesting for fuzzing.

### Management frames

Management frames are used primarily for network identification and controlling who is part of a wireless network. Depending on the *subtype* field included in the header, a wireless device can send out

- *Probe requests* to request information about a network
- *Authentication* frames and *association requests* to connect to a network
- *Reauthentication* frames and *reassociation requests* when roaming from one access point to another
- other frames for similar purposes.

The management frames' information is encoded in *information elements* or *IEs* that follow the *TLV*<sup>3</sup> format. Since this is obviously an ideal situation where fuzzing can be applied, our work concentrates on these frame types. A complete list of different information elements is found in Table 5 in Appendix B.

### Data frames

Data frames are used for actually transmitting data packets between two wireless network devices. Except for possible fragmentation information, the frame format is straight forward and does not provide a good starting point for fuzz-testing.

<sup>3</sup>Within data communication protocols, optional information may be encoded as a type-length-value or TLV element inside of the protocol.

The type and length fields are fixed in size (1-4 bytes) and the value field is of variable size. These fields are used as follows:

- Type - a numeric code which indicates the kind of field that this part of the message represents
- Length - the size of the value field (typically in bytes)
- Value - variable sized set of bytes which contains data for this part of the message[2]

## 2.1. Management Frame States

As mentioned in the previous section, our work focuses on fuzzing the information elements inside a management frames' body section. To better highlight the advantages of our approach and point out differences to previous work, our paper generally distinguishes wireless network traffic into two distinct groups: *stateless* and *stateful* frame types.

### Stateless

Stateless (management) frames are considered not to be sent as response to a message or request from a wireless device. Although this description basically boils down to one type—the *beacon frame*—when networking in managed mode<sup>4</sup>, most information elements can be fuzzed in this group (as almost all information can be sent out using beacons).

Although stateless fuzzing in *ad hoc mode* includes a much wider range of management frame types since the return frames can be ignored (as long as the conversation is based on a simple request-response paradigm) and thus meets our requirements of a stateless conversation, this group of frames is also considered *stateful* as the results of the fuzzing can be estimated more precisely when analyzing the response packets as well.

### Stateful

As a stateful frame type, we consider anything that does not meet the above requirement for stateless frame types. Particularly, this includes

- Active scanning using *probe request - response* frames
- 802.11 *authentication* frame exchange
- 802.11 *association request - response* frames
- Action frames.

A table summarizing which type of frames are fuzzed in which situation (state) is summarized in Tables 3 and 5.

## 3. Stateful Fuzzing

Most previous work on 802.11 fuzzing can be summarized as follows: One wireless device (from now on called the *fuzzer*) is placed into *monitor*<sup>5</sup> and *promiscuous*<sup>6</sup> mode

<sup>4</sup>In *managed mode*, all wireless devices communicate with each other using a central access point they first connect to. Devices in *ad hoc mode*, on the other hand, do not require such a central node and communicate directly.

<sup>5</sup>In *monitor mode* the device driver accepts all incoming frames and passes them on to the network stack without removing any 802.11 header information.

<sup>6</sup>In *promiscuous mode* the network stack accepts all incoming packets, regardless of their intended destination address

and receives all sent packets from a second wireless device (from now on called the *target*) that is to be fuzzed.

The fuzzer continuously listens for incoming packets until the target is up and running (the attacker can see this because the target will usually start sending out probe request packets). It then starts injecting a large load of fuzzed packets, assuming that at least one of them will be received by the target.

Often, a second (usually direct, hard-wired) network connection between the fuzzer and the target is used to monitor the impact of the fuzzed network packets (assuming that an invalid packet would crash the device driver or even the whole system or at least produce an entry in the system's log).

Generally, this approach brings good results when fuzzing stateless 802.11 conversations. When dealing with the much larger range of stateful frame types, the uncertainty of the device driver actually handling the packet presents fuzzers with a number of problematic issues:

1. **Timing:** As mentioned previously, 802.11 conversations rely on strict timeouts. The device driver may therefore drop a packet because the answer arrives too late.
2. **Overload:** The device driver or hardware may drop packets because the large load of incoming packets would overflow the device's receive buffer.
3. **Corruption:** A large load of packets or other disturbances of the medium might alter and thus corrupt the content of the frame. The device could thus drop the packet before analyzing its content because of an invalid checksum.
4. **State mismatch:** The target might have accepted a previous frame (possibly from some other station) and left the state in which the fuzzed packets are accepted.
5. **State timeout:** If the target device is brought into a desired state, but the fuzzed packets are not received within a certain time frame, the target device may decide to reset itself into a lower default state and drop any incoming higher state packets.

### 3.1. Novel Fuzzing Approach

To address the aforementioned problems, this paper introduces a novel approach that eliminates the wireless medium and physical distance between the two wireless devices by moving the target device into an emulated environment and allowing the fuzzer to interact with the target's virtual network device directly.

This new approach solves all issues mentioned above and additionally introduces several practical advantages to the fuzzing process:

## Timing & State Timeouts

As the target runs in a virtual machine whose CPU we control, our implementation simply stops CPU emulation when an outgoing frame is registered by the virtual device hardware and informs the fuzzer. The fuzzer may take any time necessary to construct a reply packet, inject it into the device's ingoing queue and resume the target's CPU emulation. Obviously, this solves any timing problems.

## Overload

Since the fuzzer and virtual networking device can communicate using a high level of communication, the delivery of packets can be guaranteed. This eliminates the fuzzer's need to inject more than one packet and lowers the device's input buffer to a level even below real-life situations, where retransmission of single packets can happen quite frequently.

## Corruption & State Mismatch

Because the communication between fuzzer and target does not rely on the potentially unstable wireless medium, the fuzzing process cannot be affected by any other rogue source of frame packets. Furthermore, the guaranteed delivery solves both problems of corruption and state mismatch mentioned above.

## Simple Environment

A very practical advantage of our simulation approach is its relatively low costs: when intensive fuzzing is desired, multiple computers can do the fuzzing without the need for the wireless hardware to be present. Furthermore, the need for a separate fuzzer computer or network device is eliminated. All fuzzing can be done on a single computer, a fact that simplifies the work significantly.

## Advanced Target Monitoring

When using traditional fuzzing methods, quite sophisticated target monitoring is necessary. In order to recognize errors or system crashes in the target system, advanced event logging daemons or human interaction cannot be avoided.

By moving the target into a simulated environment, completely new monitoring possibilities emerge. The virtual machine's direct access to the target's memory and CPU can be used to

- Observe unexpected halts or restarts of the CPU
- Inspect certain memory regions for error logs / dumps

- Trace the CPU's *instruction pointer* into predefined code areas such as the Microsoft Windows' Blue Screen Of Death (*BSOD*) displayed after fatal system errors) or Linux's kernel Oops functions
- Monitor the screen output for certain events (again—typical blue background of a *BSOD* or the console's output of a kernel Oops)

## System Checkpoints

As mentioned in previous sections, it can be a tedious task to bring the target system into a desired state where certain packets are accepted. When the target leaves this state for various reasons or the system crashes after finding a major bug, human interaction may be necessary in order to restore the desired state—a time intensive process that considerably complicates automated fuzzing.

By introducing *system checkpoints*—well defined situations where the current state of the CPU and all necessary memory regions are stored to disk permanently—human interaction can be eliminated by simply resuming certain checkpoints before each test run, creating a stable environment with reproducible results.

# 4. System Design & Implementation

## 4.1. System Overview

Our work is based on the QEMU[1] virtual machine and two different operating systems: One system runs Microsoft Windows XP Professional (with service pack 2 but no further updates) and on the other system we installed Kubuntu Dapper Drake (version 6.06) with the Linux kernel 2.6.20 (without a graphical environment as this slows down the emulation).

The virtual wireless network device is based on the two default NICs<sup>7</sup> *rtl8139* and *ne2000* of QEMU and simulates an Atheros AR5212 wireless chipset that is described in Section 4.3.

In the Windows system, the official *HP WLAN W400-W500 Driver for Windows 2000/XP*<sup>8</sup> was used, the Linux system was tested using the official *MadWifi Wireless Driver*<sup>9</sup>, an experimental MadWifi version with an open source implementation of the Atheros HAL<sup>10</sup>, and the *HP WLAN W400-W500 Driver for Windows 2000/XP* using *NDisWrapper*<sup>11</sup>.

<sup>7</sup>Network Interface Cards

<sup>8</sup>Available at <http://www.hp.com>

<sup>9</sup>Available at <http://madwifi.org>

<sup>10</sup>Also available at <http://madwifi.org>

<sup>11</sup>Available at <http://ndiswrapper.sourceforge.net/>

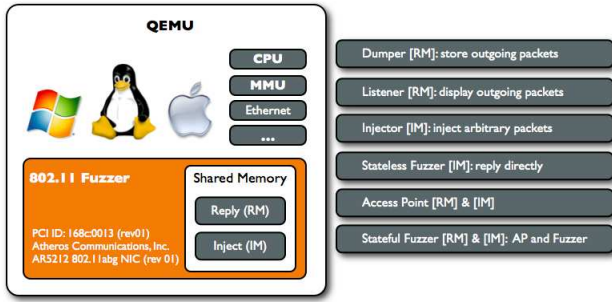


Figure 1: System Modules

## 4.2. Modular Extensions

Our implementation of the QEMU wireless device does not provide any immediate fuzzing capabilities. Instead, we decided to follow a modular design approach, allowing the virtual device to be connected to any type of communication module.

As briefly mentioned in Section 3.1, the only task of the virtual device is the detection of outgoing wireless packets, informing any connected modules and injecting possible reply packets created by these modules.

Our code relies on shared memory combined with a set of semaphores allowing us to simultaneously connect multiple modules to the virtual device. These modules have the possibility to

- Observe outgoing packets
- Reply to outgoing packets
- Inject ingoing packets (independent of outgoing packets)
- Observe the virtual device’s state (like current frequency).

During the implementation, a set of modules (Figure 1) was created for testing, observing and fuzzing target systems:

### Listener / Dumper

The most basic *listener* and *dumper* modules are used for inspection of outgoing packets and form a foundation for more sophisticated modules. These modules are responsible for recording and analyzing outgoing packets.

Every packet is inspected on the basis of its type/subtype combination and its destination address, and is then stored on disk using the *pcap*<sup>12</sup> file format to allow for network

<sup>12</sup>Pcap is an application programming interface for packet capturing. The implementation of pcap for Unix-like systems is known as libpcap; the Windows port of libpcap is called WinPcap.[3]

analysis tools such as *Wireshark*<sup>13</sup> to be used to further inspect the packets’ contents.

The following source code shows the simplicity of monitoring outgoing packets. For a definition of the the structures and constants, please refer to Appendix A.

```

1  unsigned char cmd;
2  int sem = 0;
3  remote_command *mem = NULL;
4
5  if (init_connection(&sem, &mem,
6                    REMOTE_FUZZER_APP_REPLY_SEM_KEY,
7                    REMOTE_FUZZER_APP_REPLY_DATA_ID))
8  {
9      // error handling
10     ...
11 }
12
13 while (1)
14 {
15     wait_semaphore(sem, 1);
16
17     if (mem->command == REMOTE_FUZZER_PACKET_NOTIFICATION)
18     {
19         dumpFrame(&mem->frame, mem->frame_length, "output.cap");
20     }
21
22     // signal the virtual device to simply drop the packet
23     mem->command = REMOTE_FUZZER_IGNORE_PACKET;
24
25     signal_semaphore(sem, 0);
26 }

```

### Injector

This module provides an infrastructure to inject arbitrary packets into the virtual device’s ingoing message queue. It automatically checks if the device currently allows injection of packets (e.g. by asserting that the device’s current frequency corresponds to the desired one as device drivers automatically drop packets received from an unexpected frequency) and thus assures the delivery.

Like the *dumper* module, all packets are stored on disk in order to allow tracing of ingoing packets. The following code displays how packets can be injected.

```

1  unsigned char cmd;
2  int sem = 0;
3  remote_command *mem = NULL;
4
5  if (init_connection(&sem, &mem,
6                    REMOTE_FUZZER_APP_REPLY_SEM_KEY,
7                    REMOTE_FUZZER_APP_REPLY_DATA_ID))
8  {
9      // error handling
10     ...
11 }
12
13 mem->command = REMOTE_FUZZER_INJECT_PACKET;
14 mem->current_frequency = IEEE80211_CHANNEL9_FREQUENCY;
15 mem->frame_length = construct_packet(&mem->frame);
16 dumpFrame(&mem->frame, mem->frame_length, "input.cap");
17
18 do
19 {
20     // Signal new packet...
21     signal_semaphore(sem, 0);
22
23     // ... and wait for the answer...
24     wait_semaphore(sem, 1);
25
26     if (mem->command == REMOTE_FUZZER_FAILURE_WRONG_FREQUENCY)
27     {
28         // The device is currently not in the requested
29         // frequency. Wait a few moments and retry!
30         usleep(...);
31     }
32     else if (...)
33
34 } until (mem->command == REMOTE_FUZZER_SUCCESS);

```

<sup>13</sup>Wireshark (formerly known as Ethereal) is a free software protocol analyzer, or “packet sniffer” application, used for network troubleshooting, analysis, software and protocol development, and education[4]

## Stateless Fuzzer

Based on the *injector* module, the *stateless fuzzer* injects fuzzed packets into the device at regular intervals. The packet's header section follows the 802.11 specification and only TLV items in the frame body are fuzzed.

In addition to plain buffer overflows, this fuzzer supports testing for *format string vulnerabilities*<sup>14</sup>.

## Access Point

Based on *listener* and *injector*, the *access point* is the most advanced module. It is an implementation of the basic 802.11 authentication protocol that allows wireless devices to join a network.

This module acts like a real device in a 802.11 infrastructure network would and

- Repeatedly injects beacon frames
- Replies to probe requests
- Accepts and sends out authentication frames
- Responds to association requests

and thus successfully simulates an access point. Furthermore, the module includes a basic implementation of the IP/ICMP protocol that allows the target to ping the simulated device.

## Stateful Fuzzer

This module behaves very similarly to the *access point* module. However, instead of replying according to the 802.11 specification, any of the packets sent out (no matter if sent out as reply or injected at regular intervals) can be substituted with a fuzzed packet. Again, plain overflow- and string format vulnerabilities are supported.

## 4.3. AR5212 Chipset Reverse Engineering

Our work simulates an Atheros AR5212 802.11 a/b/g wireless chipset. Note that not every feature is fully supported by our virtual device, as they are not required by the fuzzing process.

We decided to use the AR5212 because it seems to be one of the predominant chipsets used today. Furthermore,

<sup>14</sup>Format string attacks are a class of software vulnerability discovered around 1999, previously thought to be harmless. Format string attacks can be used to crash a program or to execute harmful code. The problem stems from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as `printf()`. A malicious user may use the `%s` and `%x` format tokens, among others, to print data from the stack or possibly other locations in memory. One may also write arbitrary data to arbitrary locations using the `%n` format token, which commands `printf()` and similar functions to write the number of bytes formatted to an address stored on the stack.[5]

as hardly any wireless chipset specification is publicly available, the experimental open source MADWifi driver seemed to be a fairly good starting point for our research<sup>15</sup>.

After creating a device prototype that was able to react to basic commands of the device driver, the virtual device was iteratively improved by comparing its behavior to that of a real hardware device as shown in figure 2.

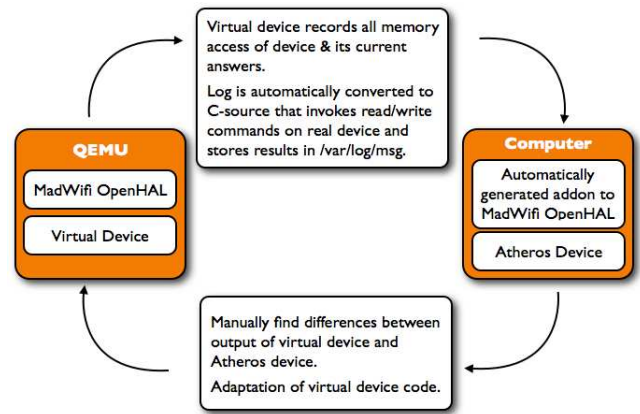


Figure 2: Iterative Reverse Engineering Process

The following code samples are taken directly from our implementation of the virtual device. They show the interaction between the device driver and the device through shared memory regions.

QEMU allows to specify callback functions that will be called when the CPU invokes a write operation on predefined physical address regions. QEMU will pass the accessed memory address, the value written to that location and a pointer to the virtual device instance to the function.

Most constants inside the source code were taken from the open source MadWifi driver, some additional values were defined during the reverse engineering process.

## Device Reset

The first listing shows the invocation of a device reset. By writing to a well-defined 4-byte memory region (line 11) the device can be told to reset all internal states or do a warm-start depending on the written value (line 12).

```
1 #define SET_MEM_L(_mem, _addr, _val) \  
2     _mem[_addr >> 2] = _val  
3  
4 // depending on the output-level, DEBUG_PRINT  
5 // will log read/writes to the device, called  
6 // functionality, etc  
7 #define DEBUG_PRINT(_msg) ...  
8  
9 static void mm_writel(  
10     WLANFUZZERState *s,
```

<sup>15</sup>Although the official Linux MadWifi driver is an open source project, it includes a binary-only HAL (hardware abstraction layer). An *open-hal* subproject of MadWifi has already successfully reverse engineered big parts of this HAL, however.

```

11         target_phys_addr_t addr,
12         uint32_t val)
13     {
14         uint32_t h;
15         switch (addr)
16         {
17             case AR5K_RESET_CTL:
18                 if (val == (AR5K_RESET_CTL_CHIP | AR5K_RESET_CTL_PCI))
19                 {
20                     DEBUG_PRINT(("reset device (MAC + PCI)\n"));
21
22                     /*
23                      * claim device is inited
24                      */
25                     SET_MEM_L(s->mem, AR5K_STA_ID1, 0);
26                     SET_MEM_L(s->mem, AR5K_RESET_CTL, 3);
27                 }
28                 else if (...)
29                     ...
30                 break;

```

## Interrupt Masking

The next section takes care of interrupt enabling, disabling and masking. These values are necessary when injecting packets into the device or acknowledging the transmission of outgoing packets.

```

1         case AR5K_IER:
2             if (val == AR5K_IER_DISABLE)
3             {
4                 DEBUG_PRINT(("disabling interrupts\n"));
5                 SET_MEM_L(s->mem, AR5K_GPIOD0, 0x0);
6
7                 s->interrupt_enabled = 0;
8             }
9             else if (val == AR5K_IER_ENABLE)
10            {
11                DEBUG_PRINT(("enabling interrupts\n"));
12                SET_MEM_L(s->mem, AR5K_GPIOD0, 0x2);
13
14                s->interrupt_enabled = 1;
15            }
16            break;
17
18            case AR5K_PIMR:
19                DEBUG_PRINT(("setting primary interrupt-mask to 0x%x (%u)\n", val,
20                             val));
21                s->interrupt_p_mask = val;
22
23                SET_MEM_L(s->mem, addr, val);
24            break;

```

## EEPROM Access

The AR5212's *EEPROM*<sup>16</sup> holds various device information that is read out by following a four-step procedure:

1. Tell the device the requested EEPROM item (address),
2. Specify the access type (read or write operation),
3. Wait for the device to copy the data to a well-defined memory-location,
4. Read this memory location to obtain the value.

Steps one and two are shown in the following code section, step three is eliminated in the virtual device, because the copy process is an atomic action and does not require any additional time (whereas on a real hardware it does).

Step four is not shown in the code, because it is a casual read operation from the virtual device's memory — in

<sup>16</sup>An EEPROM (also called an E2PROM) or Electrically Erasable Programmable Read-Only Memory, is a non-volatile storage chip used in computers and other devices to store small amounts of volatile (configuration) data.[6]

QEMU, this can be handled very similarly to the write callback function.

```

1         #define WRITE_EEPROM(_mem, _val) \
2             SET_MEM_L(_mem, AR5K_EEPROM_DATA_5210, _val); \
3             SET_MEM_L(_mem, AR5K_EEPROM_DATA_5211, _val);
4
5
6         case AR5K_EEPROM_BASE:
7             DEBUG_PRINT(("there will be an access to the EEPROM at %p\n", (
8                 unsigned long*)val));
9
10            /*
11             * set the data that will be returned
12             * after calling AR5K_EEPROM_CMD=READ
13             */
14            ...
15
16            switch (val)
17            {
18                case AR5K_EEPROM_MAGIC:
19                    WRITE_EEPROM(s->mem, AR5K_EEPROM_MAGIC_VALUE);
20                    break;
21
22                case AR5K_EEPROM_VERSION:
23                    WRITE_EEPROM(s->mem, AR5K_EEPROM_VERSION_3_4);
24                    break;
25
26                case 0x1f:
27                    /*
28                     * 1st part of MAC-addr
29                     */
30                    DEBUG_PRINT(("EEPROM request first part of MAC\n"));
31                    WRITE_EEPROM(s->mem, (s->phys[0] << 8) | s->phys[1]);
32                    break;
33
34                case 0x1e:
35                    /*
36                     * 2nd part of MAC-addr
37                     */
38                    DEBUG_PRINT(("EEPROM request second part of MAC\n"));
39                    WRITE_EEPROM(s->mem, (s->phys[2] << 8) | s->phys[3]);
40                    break;
41
42                case 0x1d:
43                    /*
44                     * 3rd part of MAC-addr
45                     */
46                    DEBUG_PRINT(("EEPROM request third part of MAC\n"));
47                    WRITE_EEPROM(s->mem, (s->phys[4] << 8) | s->phys[5]);
48                    break;
49
50                ...
51            }
52            break;
53
54            case AR5K_EEPROM_CMD:
55                /*
56                 * the type of access to the EEPROM is specified
57                 */
58
59                if (val & AR5K_EEPROM_CMD_READ)
60                {
61                    DEBUG_PRINT(("the EEPROM access will be READ\n"));
62
63                    /*
64                     * tell the device the read was successful
65                     */
66                    SET_MEM_L(s->mem, AR5K_EEPROM_STAT_5210, AR5K_EEPROM_STAT_RDDONE);
67                    SET_MEM_L(s->mem, AR5K_EEPROM_STAT_5211, AR5K_EEPROM_STAT_RDDONE);
68                    /*
69                     * and return the data that was set
70                     * during the write to AR5K_EEPROM_BASE
71                     */
72                }
73                ...
74            break;

```

## Packet Injection

The following code takes care of storing the location of the driver's outgoing message queue and injecting packets into it. For sending and receiving packets, the device driver sets up a single-linked list in the CPU's memory range accessible to the hardware.

After receiving a packet (function *handleRxBuffer*, line 12), the virtual device will fill possibly multiple items (depending on the size of the packet) of this single-linked list and signal an interrupt to inform the device driver of the event.

```

1     case AR5K_RXDP:
2         DEBUG_PRINT(("Setting receive queue to address 0x%x (%u)\n", val,
3             val));
4
5         SET_MEM_L(s->mem, addr, val);
6         s->receive_queue_address = (uint32_t *)val;
7         break;
8     ...
9 }
10 }
11
12 static void handleRxBuffer(
13     WLANFUZZERState *s,
14     struct mac80211_frame *frame,
15     uint32_t frame_length)
16 {
17     struct ath_desc desc;
18     struct ath5k_ar5212_rx_status *rx_status;
19     rx_status = (struct ath5k_ar5212_rx_status*)&desc.ds_hw[0];
20
21     if (s->receive_queue_address == NULL)
22     {
23         return;
24     }
25
26     /*
27      * get the descriptor of the single-linked
28      * list from physical memory
29      */
30     cpu_physical_memory_read(
31         (target_phys_addr_t)s->receive_queue_address,
32         (uint8_t*)&desc, sizeof(desc));
33
34     /*
35      * Put some good base-data into the descriptor. Length & co
36      * will be modified below...
37      */
38     desc.ds_ctl0 = 0x0;
39     desc.ds_ctl1 = 0x9c0;
40     desc.ds_hw[0] = 0x126d806a;
41     desc.ds_hw[1] = 0x49860003;
42     desc.ds_hw[2] = 0x0;
43     desc.ds_hw[3] = 0x0;
44
45     /*
46      * Filter out old length and put in correct value...
47      */
48     rx_status->rx_status_0 &= "AR5K_AR5212_DESC_RX_STATUS0_DATA_LEN;
49     rx_status->rx_status_0 |= frame_length;
50
51     /*
52      * For now, assume that the packet fits into one
53      * single item in the single-linked list!
54      */
55     rx_status->rx_status_0 &= "AR5K_AR5211_DESC_RX_STATUS0_MORE;
56
57     /*
58      * Write descriptor and packet back to DMA memory...
59      */
60     cpu_physical_memory_write(
61         (target_phys_addr_t)s->receive_queue_address,
62         (uint8_t*)&desc, sizeof(desc));
63     cpu_physical_memory_write(
64         (target_phys_addr_t)desc.ds_data,
65         (uint8_t*)frame,
66         sizeof(struct mac80211_frame));
67
68     /*
69      * Set address to next position
70      * in single-linked list
71      *
72      * The receive list's last element
73      * points to itself to avoid overruns.
74      * This way, at some point no more
75      * packets will be received, but (I
76      * ASSUME) that it is the drivers
77      * responsibility to reset the address
78      * list!
79      */
80     s->receive_queue_address = (uint32_t *)desc.ds_link;
81
82     /*
83      * Notify the driver about the new packet
84      */
85     struct pending_interrupt intr;
86     intr.status = AR5K_INT_RX;
87     wlanfuzzer_append_irq(s, intr);
88     wlanfuzzer_enable_irq(s);
89 }
90
91 }
92

```

## 5. Fuzzing & Evaluation

To verify the usability of our system and its ability to run automated fuzzing test suites, we used the *stateful fuzzer* module described in Section 4.2 on all system/driver com-

binations mentioned in Section 4.1.

The setup of the individual test runs and the type of fuzzing employed are summarized in Table 1. For a description of state information, refer to Table 4 in Appendix B.

Although the fuzzing suites were only meant to test our system’s usability and the employed fuzzing modules leave room for improvement, the test runs found implementation errors (summarized in Table 2) in a very short amount of time.

## 6. Summary and Conclusions

In this paper, we presented a novel approach to 802.11 fuzzing. The goal of our system is to overcome timing problems a fuzzer usually faces when testing stateful frame types.

By moving the target system into a simulated environment (a QEMU virtual machine in our case) and replacing the problematic wireless communication with a high level means of interprocess communication, not only were these timing problems solved, but a number of practical advantages were achieved.

A set of communication modules demonstrate the usability and effectiveness of our approach and could serve as a well-founded starting point for further fuzzing suites.

## Acknowledgments

We would like to thank SEC Consult Unternehmensberatung GmbH and especially Bernhard Mueller for their initiative and support for this project.

Furthermore, we greatly appreciated the valuable feedback and guiding hints of Engin Kirda and Christopher Kruegel of the Technical University Vienna.

## References

- [1] Bellard, F. “QEMU, a Fast and Portable Dynamic Translator” In *Usenix Annual Technical Conference, Freenix Track*, 2005.
- [2] Wikipedia, The Free Encyclopedia, “Type-length-value” <http://en.wikipedia.org/wiki/Type-length-value>
- [3] Wikipedia, The Free Encyclopedia, “pcap” <http://en.wikipedia.org/wiki/Libpcap>
- [4] Wikipedia, The Free Encyclopedia, “Wireshark” <http://en.wikipedia.org/wiki/Wireshark>
- [5] Wikipedia, The Free Encyclopedia, “Format string attack” [http://en.wikipedia.org/wiki/Format\\_string\\_vulnerabilities](http://en.wikipedia.org/wiki/Format_string_vulnerabilities)
- [6] Wikipedia, The Free Encyclopedia, “EEPROM” <http://en.wikipedia.org/wiki/EEPROM>

<b>Fuzzed Frame Type</b>	<b>Previous Benign Frame Types</b>	<b>Target State</b>	<b>Trigger</b>
Beacon Frame	n/a	any	Time interval
Action Frame	n/a	any	Time interval
Probe response	Beacon frame	N-A	Probe request
Authentication	Beacon frame, Probe response	N-A	Authentication
Association response	Beacon frame, Probe response, Authentication	A	Association request

Table 1: Fuzzing Suites

<b>System</b>	<b>Driver</b>	<b>Result</b>
Windows	HP WLAN W400-W500	Passed all tests
Linux using NDisWrapper	HP WLAN W400-W500	Passed all tests
Linux	MADWifi Binary HAL	DoS / complete system crash
Linux	MADWifi Open HAL	Kernel mode code execution possible

Table 2: Fuzzing Results

- [7] Butti, L. “Wi-Fi Advanced Fuzzing” France Telecom / Orange Division R&D, Presentation at *BlackHat Europe*, 2007.

## Appendix A

```

1 #define REMOTE_FUZZER_APP_REPLY_SEM_KEY          16628
2 #define REMOTE_FUZZER_APP_INJECT_SEM_KEY        16629
3
4 #define REMOTE_FUZZER_APP_REPLY_DATA_ID         16630
5 #define REMOTE_FUZZER_APP_INJECT_DATA_ID        16631
6
7 #define REMOTE_FUZZER_PACKET_NOTIFICATION        1
8 #define REMOTE_FUZZER_IGNORE_PACKET            3
9 #define REMOTE_FUZZER_INJECT_PACKET            4
10 #define REMOTE_FUZZER_FAILURE_WRONG_FREQUENCY   7
11
12
13 typedef struct remote_command
14 {
15     uint32_t command;
16     uint32_t current_frequency;
17     uint32_t frame_length;
18     struct mac80211_frame frame;
19 } remote_command;
20
21
22 int init_connection(int *semaphore, remote_command **mem, int semkey, int shmkey)
23 {
24     if (semaphore != NULL)
25     {
26         int client_semaphore;
27         client_semaphore = semget(semkey, 2, 0666 | IPC_CREAT);
28         if (client_semaphore == -1)
29         {
30             fprintf(stderr, "Creating semaphore failed!");
31             return 1;
32         }
33     }
34     *semaphore = client_semaphore;
35 }
36
37 if (mem != NULL)
38 {
39     int shmkey = shmget(semkey, sizeof(remote_command), IPC_CREAT | SHM_R | SHM_W);
40     if (shmkey < 0)
41     {
42         fprintf(stderr, "ERROR opening shared memory\n");
43         return 1;
44     }
45     *mem = (remote_command *)shmat(shmkey, NULL, 0);
46 }
47
48 return 0;
49 }
50

```

## Appendix B

<b>Id</b>	<b>Description</b>
R01	Target is scanning actively
R02	Target has transmitted a corresponding probe request
R03	Target has requested open authentication
R04	Target is currently sending beacon frames
R05	Target has requested for association
R06	ToDS and FromDS fields set to false

Table 3: Test-case requirements

<b>Id</b>	<b>Description</b>
N-A	Not authenticated, not associated, default state
A	Authenticated but not associated
A-A	Authenticated and associated

Table 4: Test-case state description

<b>Id</b>	<b>Frame type</b>	<b>Frame subtype</b>	<b>Target field</b>	<b>State</b>	<b>Mode</b>	<b>Req.</b>
011101	Management frame	Beacon	SSID	N-A	Managed	R01
011102	Management frame	Beacon	TIM	N-A	Managed	R01
011103	Management frame	Beacon	Country Info	N-A	Managed	R01
011104	Management frame	Beacon	Extended Rates	N-A	Managed	R01
011201	Management frame	Beacon	SSID	N-A	Ad-hoc	R01
011202	Management frame	Beacon	TIM	N-A	Ad-hoc	R01
011203	Management frame	Beacon	Country Info	N-A	Ad-hoc	R01
011204	Management frame	Beacon	Extended rates	N-A	Ad-hoc	R01
021201	Management frame	Probe Request	SSID	N-A	Ad-hoc	R04
021202	Management frame	Probe Request	Supported Rates	N-A	Ad-hoc	R04
021203	Management frame	Probe Request	Ext. Supported Rates	N-A	Ad-hoc	R04
031101	Management frame	Probe Response	SSID	N-A	Managed	R02
031102	Management frame	Probe Response	Supported Rates	N-A	Managed	R02
031103	Management frame	Probe Response	Country Info	N-A	Managed	R02
031104	Management frame	Probe Response	FH Pattern	N-A	Managed	R02
031105	Management frame	Probe Response	IBSS DFS	N-A	Managed	R02
031106	Management frame	Probe Response	Ext. Supported Rates	N-A	Managed	R02
031107	Management frame	Probe Response	RSN	N-A	Managed	R02
031201	Management frame	Probe Response	SSID	N-A	Ad-hoc	R02
031202	Management frame	Probe Response	Supported Rates	N-A	Ad-hoc	R02
031203	Management frame	Probe Response	Country Info	N-A	Ad-hoc	R02
031204	Management frame	Probe Response	FH Pattern	N-A	Ad-hoc	R02
031205	Management frame	Probe Response	IBSS DFS	N-A	Ad-hoc	R02
031206	Management frame	Probe Response	Ext. Supported Rates	N-A	Ad-hoc	R02
031207	Management frame	Probe Response	RSN	N-A	Ad-hoc	R02
041101	Management frame	Auth. Request	Challenge text	N-A	Managed	R03
041102	Management frame	Auth. Response	Challenge text	N-A	Managed	
041201	Management frame	Auth. Request	Challenge text	N-A	Ad-hoc	R03
041202	Management frame	Auth. Response	Challenge text	N-A	Ad-hoc	
052201	Management frame	Assoc. Request	SSID	A	Ad-hoc	
052202	Management frame	Assoc. Request	Supported Rates	A	Ad-hoc	
062201	Management frame	Reassoc. Request	SSID	A	Ad-hoc	
062202	Management frame	Reassoc. Request	Supported Rates	A	Ad-hoc	
072101	Management frame	Assoc. Response	Supported Rates	A	Managed	R05
072201	Management frame	Assoc. Response	Supported Rates	A	Ad-hoc	R05
081101	Data frame	Data	Frame body	N-A	Managed	R06
081201	Data frame	Data	Frame body	N-A	Ad-hoc	R06
082101	Data frame	Data	Frame body	A	Managed	R06
082201	Data frame	Data	Frame body	A	Ad-hoc	R06
083101	Data frame	Data	Frame body	A-A	Managed	
083201	Data frame	Data	Frame body	A-A	Ad-hoc	

Table 5: Test-case classes