A glimpse into the Linux Wireless Core: From kernel to firmware

Francesco Gringoli
University of Brescia
• Linux Kernel Network Code
  – Modular architecture: follows layering
• Descent to (hell?) layer 2 and below
  – Why hacking layer 2
  – OpenFirmWare for WiFi networks
• OpenFWWF: RX & TX data paths
• OpenFWWF exploitations
  – TCP Piggybacking
  – Partial Packet Recovery
Linux Kernel Network Code

A glimpse into the Linux Kernel Wireless Code
Part 1
Linux Networking Stack
Modular architecture

• Layers down to MAC (included)
  – All operations above/including layer 2 done by kernel code
  – Net code device agnostic
  – Net code prepares suitable packets

• In 802.3 stack
  – Eth code talks with device drivers
  – Device drivers
    • Map/unmap DMA desc to packets
    • Set up Hardware registers
What happens with 802.11?

- New drivers to handle WiFi HW: how to link to net code?
- A wrapper “mac80211” module is added
Linux & 802.11
Modular architecture

• Layers down to LLC (~mac) common with 802.3
  – All operations above/including layer 2 done by ETH/UP code

• Packets converted to 802.11 format for rx/tx
  – By wrapper “mac80211”
    • Manage packet conversion
    • Handle AAA operations

• Drivers: packets to devices
  – One dev type/one driver
    • Add data to “drive” the device
• Convert agnostic info into device dependent data
• Compute Control Word, Duration, sequence number
• Fill header, add LLC (0xAA 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00)
• Add information for HW setup (device agnostic) in info fields

Wait TX opportunity

Set HW registers

From kernel to firmware
• Opposite path: conversions reversed
• ☹ Several operations involved for each packet
• ☺ Multiple buffer copies (should be) avoided
  – E.g., original packet at layer 4 correctly allocated
    • Before L3 encapsulation output device already known
• ☹ Packets are queued twice
  – Qdisc: before wrapper
  – Device queues: between wrapper and driver
• Bottom line:
  – Clean design but can be resource exhausting
- Forwarding/routing packet on a double interface box
• On CPU limited platform, fw performance too low
  – Need to accelerate/offload some operations
• Ralink was first to introduce SoC WiFi devices
  – A mini-pci card hosts an ARM CPU
  – Main host attaches a standard ethernet iface
  – The ARM CPU converts ETH packet to 802.11
  – Main host focuses on data forwarding

• Question: where can be profitably used?
Linux & 802.11: setup

• A simple BSS in Linux
  – One station runs hostapd (AP)
  – Others join (STAs): wpa_supplicant keeps joining alive
    • Why? Kernel (STA) periodically checks if AP is alive
    • If management frames lost, kernel (STA) does not retransmit!
    • A supplicant is needed to re-join the BSS
  – In following experiments we fix arp associations
    $: ip neigh replace to PEERIP lladdr PEERMAC dev wlan0
  – Traffic not encrypted
  – QoS disabled
• Check the device type with
  $: lspci | grep -i net

• Load the driver for Broadcom devices
  $: modprobe b43 qos=0

• Check kernel ring buffer with
  $: dmesg | tail -30

• Check which other modules loaded
  $: lsmod | grep b43

• Bring net up and configure an IP address
  $AP: ifconfig wlan0 192.168.1.1 up
  $STA: ifconfig wlan0 192.168.1.10 up
Linux & 802.11: hostapd setup

• Configuration of the AP in “hostapd.conf”

  interface=wlan0
driver=nl80211
dump_file=/tmp/hostapd.dump
ctrl_interface=/var/run/hostapd

 ssid=NOISE-B43
  hw_mode=g
  channel=1
  beacon_int=100
  auth_algs=3
  wpa=0

• Runs with

  $: hostapd -B hostapd.conf

• Check dmesg!

  Try to send SIGUSR1

  PIPE used by hostapd_cli

  BSS properties

  No encryption/authentication
Linux & 802.11: station setup

- Configuration of STAs in

```
ctrl_interface=/var/run/wpa_supplicant
network={
  ssid="NOISE-B43"
  scan_ssid=1
  key_mgmt=NONE
}
```

- Runs with

```
$: wpa_supplicant -B -i wlan0 -c wpa_supp.conf
```

- Check dmesg!

- **Simple experiment: ping the AP**

```
$: ping 192.168.1.1
```

- **Simple experiment (continued): try capture traffic**
• On both AP and STA run “tcpdump”
  
  $: tcpdump -i wlan0 -n

• Is exactly what we expect?
  – What is missing?
  – Layer 2 acknowledgment?

• Display captured data
  
  $: tcpdump -i wlan0 -n -XXX

• What kind of layer 2 header?

• What have we captured?
Linux & 802.11: capturing packets

- Run "tcpdump" on another station set in monitor mode
  
  ```
  $: ifconfig wlan0 down
  $: iwconfig wlan0 mode monitor chan 4(?)
  $: ifconfig wlan0 up
  $: tcpdump -i wlan0 -n
  ```

- What’s going on? What is that traffic?
  - Beacons (try to analyze the reported channel, what’s wrong?)
  - Probe requests/replies
  - Data frames

- Try to dump some packet’s payload
  - What kind of header?
  - Collect a trace with tcpdump and display with Wireshark
Linux & 802.11: capturing packets

- Exercise: try to capture only selected packets
- Play with matching expression in tcpdump
  
  ```
  $: [cut] ether[N] ==|!= 0xAB
  ```
- Discard beacons and probes
- Display acknowledgments
- Display only AP and STA acknowledgments
- Question: is a third host needed?
Virtual Interfaces

- Wrapper/driver “may agree” on virtual packet path
  - Each received packet duplicated by the driver
  - mac80211 creates many interfaces “binded” to same HW
  - In this example
    - Monitor interface attached
    - Blue stream follow upper stack
    - Red stream hooked to pcap

```
$: iw dev wlan0 interface add \
   fish0 type monitor
```

- Try capturing packets on the AP
  - What’s missing?
Descent to layer 2 and below
An open firmware

A glimpse into the
Linux Kernel Wireless Code
Part 2
Linux & 802.11
Modular architecture

Wrapper for all hw
Find interface;
remove eth head;
add LLC&dot11 head;
fill (sa;da;ra;seq);
fill(control;duration);
set rate (from RC);
fill (rate;fallback);

Ethernet & upper layers

mac80211

b43 carl9170 ath9k

PCI USB M-PCI

Trento 29/4/2011
Set up hw regs; Fill hw private fields; Send frame on DMA; Get stats; Reports to mac80211

Several MAC primitives missing! Who takes care of ack?
We will hack the firmware today but first...
Let’s check why we should do that 😊
Radio access protocols: issues
  - Some are unpredictable: noise & intf, competing stations

Experimenting with simulators (e.g., ns-3)
  - Captures all “known” problems
    - Testing changes to back-off strategy is possible 😊
  - Unknown (not expected)?
    - Testing how noise affects packets not possible 😞

In the field testing is mandatory
  - Problem: one station is not enough!
Programmable Boards

• Complete platforms like
  – WARP: Wireless open-Access Research Platform
  – Based on Virtex-5
  – Everything can be changed
    • PHY (access to OFDM symbols!)
    • MAC
  – Two major drawbacks
    • More than very expensive
    • Complex deployment
  – If PHY untouched: look for other solutions!
Off-the-shelf hardware

• Five/Six vendors develop cheap WiFi hw
  – Hundreds different boards
  – Almost all boards load a binary firmware
    • MAC primitives driven by a programmable CPU
  – Changing the firmware ➔ Changing the MAC!

• Target platform:
  – Linux & 802.11: modular architecture
  – Official support prefers closed-source drivers 😞
  – Open source drivers & Good documentation
    • Thanks to community! 😊
Linux & 802.11
Broadcom AirForce54g

- Architecture chosen because
  - Existing asm/dasm tools
    - A new firmware can be written!
  - Some info about hw regs
- We analyzed hw behavior
  - Internal state machine decoded
  - Got more details about hw regs
  - Found timers, tx&rx commands
  - Open source firmware for DCF possible
- We released OpenFWWF!
  - OpenFirmWare for WiFi networks
Broadcom AirForce54g
Basic HW blocks

- FIFOs
- DMA
- Template RAM
- TXE
- CPU
- ucode
- Internal memory
- PHY
- RXE

From kernel to firmware
Description of the HW

- CPU/MAC processor capabilities
  - 8MHz CPU, 64 general purpose registers
- Data memory is 4KB, direct and indirect access
  - From here on it’s called Shared Memory (SHM)
- Separate template memory (arrangeable > 2KB)
  - Where packets can be composed, e.g., ACKs & beacons
- Separate code memory is 32KB (4096 lines of code)
- Access to HW registers, e.g.:
  - Channel frequency and tx power
  - Access to channel transmission within N slots, etc...
TX side

- Interface from host/kernel
  - Six independent TX FIFOs
  - DMA transfers @ 32 or 64 bits
  - HOL packet from each FIFO
    - can be copied in data memory
      - Analysis of packet data before transmission
      - Kernel appends a header at head with rate, power etc
    - can be transmitted “as is”
    - can be modified and txed, direct access to first 64 bytes
• Interface to air
  – Only 802.11 b/g supported, soon n
  – Full MTU packets can be transmitted (~2300 bytes)
    • If full packet analysis is needed, analyze block-by-block
  – All 802.11 timings supported
    • Minimum distance between Txed frames is 0us
      – Note: channel can be completely captured!!
  – Backoff implemented in software (fw)
    • Simply count slots and ask the HW to transmit
RX side

• Interface from AIR
  – HW acceleration for
    • PLCP and global packet FCS - Destination address matching
  – Packet can be copied to internal memory for analysis
    • Bytes buffered as soon as symbols is decoded
  – During reception and copying CPU is idle!
    • Can be used to offload other operations
  – Packets are pushed to host/kernel
    • If FW decides to go and through one FIFO ONLY
    • May drop! (e.g., corrupt packets, control...)

Trento 29/4/2011
Example:
TX a packet, wait for the ACK
Example:
RX a packet, transmit an ACK

Send to host
PKT is for me

Wait SIFS
What lesson we learned

• From the previous slides
  – Time to wait ack (success/no success)
  – Dropping ack (rcvd data not dropped, goes up)
  – And much more
    • When to send beacon
    • Backoff exponential procedure and rate choice
  – Decided by MAC processor (by the firmware)

• Bottom line:
  **Hardware is (almost) general purpose**
• OpenFWWF
  – It’s not a production firmware
  – It supports basic DCF
    • No RTS/CTS yet, No QoS, only one queue from Kernel
  – Full support for capturing broken frames
  – It takes 9KB for code, it uses < 200byte for data
    • We have lot of space to add several features
• Works with 4306, 4311, 4318 hw
  – Linksys Routers supported (e.g., WRT54GL)
Broadcom AirForce54g
Simple TDM

PKT

FIFOs

DMA

Template RAM

TXE

CPU

RXE

ucode

SHM

PHY

TDM needed!
Waiting turn

GO!

Slide 36
Trento 29/4/2011
From kernel to firmware
Sync the clock

PKT from TDM domain

From kernel to firmware
OpenFWWF
RX & TX data paths

A glimpse into the
Linux Kernel Wireless Code
Part 3
Firmware in brief

- Firmware is really complex to understand 😞
  - Assembly language
    - CPU registers: 64 registers [r0, r1, ..., r63]
    - SHM memory: 4KB of 16bits words addressable as [0x000] -> [0x7FF]
    - HW registers: spr000, spr001, ..., spr1FF
  - Use `#define` macro to ease understanding
    - `#define CUR_CONTENTION_WIN r8`
    - `#define SPR_RXE_FRAMELEN spr00c`
    - `#define SHM_RXHDR SHM(0xA88)`
    - `SHM(.)` is a macro as well that divides by 2
  - Assignments:
    - Immediate: `mov 0xABBA, r0;` // load 0xABBA in r0
    - Memory direct: `mov [0x0013], r0;` // load 16bit @ 0x0026 (LE!)
• Value manipulation:
  – Arithmetic:
    • Sum: \( \text{add} \ r1, r2, r3; \)  // \( r3 = r1 + r2 \)
    • Subtraction: \( \text{sub} \ r2, r1, r3; \)  // \( r3 = r2 - r1 \)
  – Logical:
    • Xor: \( \text{xor} \ r1, r2, r3; \)  // \( r3 = r1 \oplus r2 \)
  – Shift:
    • Shift left: \( \text{sl} \ r1, 0x3, r3; \)  // \( r3 = r1 \ll 3 \)

• Pay attention:
  – In 3 operands instruction, immediate value in range [0..0x7FF]
  – Value is sign extended to 16bits
• Code flow execution controlled by using jumps
  – Simple jumps, comparisons
    • Jump if equal: `je r2, r5, loop;` // jump if r2 == r5
    • Jump if less: `jl r2, r5, exit;` // jump if r2 < r5 (unsigned)
  – Condition register jumps: jump on selected CR (condition registers)
    • on plcp end: `jext COND_RX_PLCP, rx_plcp;`
    • on rx end: `jext COND_RX_COMPLETE, rx_complete;`
    • on good frame: `jext COND_RX_FCS_GOOD, frame_ok;`
    • unconditionally: `jext COND_TRUE, loop;`
  – A check can also clean a condition, e.g.,
    • `jext EOI(COND_RX_PLCP), rx_plcp;` // clean CR bit before jump
  – Call a code subsection, save return value in link-registers (lr):
    • `call lr0, push_frame;` // return with ret lr0, lr0;
• OpenFWWF is today ~ 1000 lines of code
  – Not possible to analyze in a single lesson
  – We will analyze only some parts

• A simple exercise:
  – Analyze quickly the receiver section
  – Propose changes to implement a jammer
    • When receives packets from a given STA, jams noise!
RX code made easy

- During reception CPU keeps on running
  - Detect end of PLCP
  - May wait for a given number of bytes received
  - May prepare a response frame (ACK)
  - Wait for end of reception
  - May schedule response frame transmission after a while now

[Image: PLCP received
WAIT first N bytes
Received: analyze header
If from jam target setup jam
Wait for packet end
Reception complete]

[Image: JAM READY!
[PLCP]
[M-1...N] [N-1...0]]

Slide 43   Trento 29/4/2011
From kernel to firmware
RX code made easy/2

Slide 44  Trento 29/4/2011  From kernel to firmware
• During reception
  – CR RX_PLCP set when PLCP is completely received
  – CR COND_RX_BADPLCP set if PLCP CRC went bad
  – SPR_RXE_FRAMELEN hold the number of already received bytes
  – First 64B of packet are copied starting at \texttt{SHM\_RXHEADER = SHM(0x908)}
    • First 6B hold the PLCP
  – CR COND_RX_COMPLETE set when packet is ready

• We can have a look at the code flow for a data packet
  – \texttt{rx\_plcp}: checks it’s a data packet
  – \texttt{rx\_data\_plus}: checks packet is longer than 0x1C = 6(PLCP)B + 22(MAC)B
  – \texttt{send\_response}: copy src mac address to ACK addr1, set state to TX_ACK
  – \texttt{rx\_complete}: schedule ACK transmission
• If first byte of a packet are copied to SHM
• If we have ways of displaying SHM
  – Could we find evidence of received packets?
• Useful tool
  – $: b43-fwdump [-s]
  – Display r0..r63 registers
  – Switch “-s” dump content of SHM
• Run this experiment: Ping the AP very fast from the STA
  $: ping -i 0.1 192.168.1.1 -b size
  – On AP dump the SHM: locate the ICMP packet
  – Fix the rate on STA: how do the first 6 bytes change?
  – Try for different ICMP size.
Disturbing a station when sending data
  - Jammer recognizes tx’ed data and sends fake ACK packet
    - Starts little before the SIFS
    - Send a slightly longer packet

Maybe (for testing) jamming all packets is too much
  - Selected packets?
• Propose changes to code flow for a selected data packet
• Exercise: only for UDP packets to port 43962
  – rx_plcp: checks it’s a data packet
  – rx_data_plus: checks packet is longer than 0x1C = 6(PLCP)B + 22(MAC)B
  – send_response: copy src mac address to ACK addr1, set state to TX_ACK
  – rx_complete: schedule ACK transmission
JAM code

• To switch to a different firmware
  – Look at /lib/firmware
  – Link the desired firmware release as “b43”
  – Remove b43 module, reload and bring back the network up
    $: rmmod b43 . . .

• How to test JAM code? “iperf” performance tool

• On AP run in server mode (receiver)
  $: iperf -s -u -p 10000 -i 1

• On STA run in client mode (transmit)
  $: iperf -c 192.168.1.1 -u -p 10000 -i 1 -t 10
• Packets are prepared by the kernel
  – Fill all packet bytes (e.g., 802.11 header)
  – Choose hw agnostic device properties
    • Tx power to avoid energy wasting
    • Packet rate: rate control algorithm (minstrel)
  – A driver translates everything into hw specific
    • b43: rate encoded in PLCP (first 6B)
    • b43: append a fw-header at packet head
      – Firmware will setup hw according to these values
TX made easy/2

- **Kernel (follows)**
  - b43: send packet data (+hw info) through DMA

- **firmware:**
  - Continuous loop, when no receiving
    - If IDLE, check if packet in FIFO (comes from DMA)
    - If packet does not need ACK, TX, report and exit
    - If packet needs ACK, wait ACK timeout
    - If ACK timeout expired:
      - if ACK RXed, report to kernel, exit
      - If ACK not RXed, setup backoff, try again
    - If too much TX attempt, remove packet from FIFO, report to kernel, exit

Trento 29/4/2011
New packet in FIFO
TX attempt = 0

Device TX FIFO

Second attempt:
increase backoff

TX STATUS FIFO

ACK ok
Report to kernel

Status N=2

IRQ wake status handler in kernel

RX ANT

Packet corrupt
No ACK back

ACK

Timeout!
• Summary

- FW reports to kernel the number of attempts
  - Kernel feeds the rate control algo
  - A rate for the next packet is chosen
• Currently “minstrel” is the default RC algo
  – At random intervals tries all rates
  – Builds a tables with success “rate” for each “rate”
  – In the short term it selects the best rate
  – How to checks this table from userspace?
    • DEBUGFS 😊
    • Take a look at folder
      sys/kernel/debug/ieee80211
TX made easy: exercise

• Firmware: backoff entered if ack is not rx
  – Simple experiment
    • Two STAs joined to the same BSS
    • iperf on both STAs to the AP
    • They should share the channel
  – What happen if we hack one station fw?
  – Let’s try...
    • TX path really complex, skip
    • But at source top we have a few “_CW” values
OpenFWWF Exploitation: Two concrete MACs released

A glimpse into the Linux Kernel Wireless Code
Part 4
OpenFWWF Exploitation: TCP-PIGGYB-ACK

In collaboration with
Ilenia Tinnirello & Pierluigi Gallo
University of Palermo
TCP flow over WiFi

- AP: sends data segments to STA (e.g., from remote)
- STA: sends TCP ACK to AP (that forwards them)
  -- Two separate channel accesses
- Idea: TCP ACK is short
  -- Why not replacing L2 ACK with a mixed L2+L4 ACK?

\[ T_a = \text{TCP}_\text{DATA} + \text{SIFS} + \text{ACK} + \text{DIFS} + \text{TCP}_\text{ACK} + \text{ACK} + \text{DIFS} + E[\text{backoff}] \]
• Expected behavior: TCP-PIGGYB-ACK!

• Enhanced behavior, work in progress.

T_c = 2 TCP_DATA + 3 SIFS + 3 DIFS + 2 TCP_ACK + 2 ACK + 2 E[backoff]

T_b = 2 TCP_DATA + 2 SIFS + 2 DIFS + 2 TCP_ACK + ACK + E[backoff]
TCP-PIGGYB-ACK: scenario

If this packet gets lost...
It will never be retransmitted!

TCP will correct this at next ACK
TCP-PIGGYB-ACK: changes

• FW @ rx
  – Piggyback: only if a TCP DATA is received
    • Avoid Ping-Pong
  – Piggyback: only if a TCP ACK is in queue
    • If not, send L2 ACK
  – Piggyback: header is L2ACK, longer!

• Kernel @ tx
  – If L2ACK long (=TCP ACK) received
    • Forge and inject a recovered TCP ACK in the stack
TCP-PIGGYB-ACK
Performance Evaluation

• Testbed & measurement
  
  – Two peers, several other BSS
  – One peer is the Access Point

while(1) {
  For 60 sec: exchange traffic with no PIGGYBACK
  Measure throughput T1 at rx
  For 60 sec: exchange traffic with PIGGYBACK
  Measure throughput T2 at rx
  Plot(T1, T2)
}
Performance Evaluation

Data rate fixed to 2Mb/s
Performance Evaluation

Data rate fixed to 11Mb/s
Performance Evaluation
Data rate free

Slide 65  Trento 29/4/2011  From kernel to firmware
TCP-PIGGYB-ACK: Comments

• Lost TCP-ACK in piggybacking
  – Not retransmitted

• Problems with rate control algorithm?

• Not all TCP segment are piggybacked with TCP-ACK
  – E.g., when the queue is empty
TCP-PIGGYB-ACK: exercise

• Switch module and firmware
  – We have a single kernel module for rx/tx
  – Still two separated FW – Not production!

• Keep in mind: for debug purposes
  – Experiments “legacy” to port 12346
  – Experiments “piggy” to port 12345
  – AP should receive TCP data, generate L2+L4 ACK
  – STA should transmit TCP data

• Play with /sys/kernel/debug/b43/phyN/specack
• Use iperf/tcp
  – AP(rx) $: iperf -s -p 12345|12346 -i 1
  – STA(tx) $: iperf -c 192.168.1.1 -p 12345 -i 1 -t 10
• At the end on both, issue
  – $: sudo cat /sys/kernel/debug/b43/phyN/specack
• To reset statistics
  – $: echo 0 | tee /sys/kernel/debug/b43/phyN/specack
OpenFWWF Exploitation: Partial Packet Recovery

In collaboration with
Errors & noise in WiFi

• Packet Error Rate of 802.11 networks is high[1]
  – Random noise can affect only a few bits
    • One or multiple blocks of corrupted bits inside a packet
  – Corrupted frames are discarded
    • Even if only 1 bit is wrong!
  – 802.11 retransmits after ACK timeout
  – Correctly received bits are completely wasted

Suppose we divide packets into 64 bytes block

– Typical packet trace of a managed station
Recent Approaches

• Forward Error Correction (FEC) based
  – ZipTx [2] sends RS redundant bits for recovery
  – Two-round coding scheme
  – Educated guess of BER and high recovery delay
    • Implemented(?) in kernel-space on Atheros devices
    • Evaluated in 11a, outdoor tests (low interference)

Recent Approaches

• Based on Automatic Repeat reQuest (ARQ)
  – PPR [3] relies on the confidence of each bit’s correctness
  – Retransmit only corrupted bits
  – Not available in commercial hardware
    • implemented and evaluated on 802.15.4 protocol stack

Our approach

• Similar to PPR
  – No access to confidence information
• Use checksum coefficient embedded in packets
• We implemented everything from scratch
  – Changes to Linux kernel
  – Changes to OpenFWWF
• We designed MARANELLO and BOLOGNA
  – AKAS Practical Partial Packet Recovery P³R!
• At rx corrupted packet is divided into blocks
  – Blocks are equally sized (apart the last one)
  – For each block apart the first compute a checksum
  – Checksums sent back to the transmitter in a N-ACK

  Corrupted packet:

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
<th>Block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  To transmitter
  – Transmitter retransmits only corrupted blocks
  – First block can’t be protected
    • It must always be retransmitted, contains the header!
Maranello: handling retransmission

Block Checksum Replacement on Header of Packet

From kernel to firmware
- Like Maranello but...
- At tx packet is expanded
  - In each block a checksum is embedded
- Rx checks all blocks:
  - If packet fails, send back a NACK
  - NACK is the bitmap of corrupt blocks
Bologna: handling retransmission

- Generation and transmission of repair packet
- Corrupt block replacement
- Packet reduction
- Block checksums computation (Fletcher16)
- Packet expansion and transmission
- Block checksums checking
- Generation and transmission of NACK
Advantages of P$^3$R

- Receiver-controlled recovery
- Utilizing the airtime reserved for ACKs
  - No additional overhead for correct packets
- Faster packet recovery
  - Recovery immediately after a transmission fails
  - Shorter recovery frames
• Time-critical operations should be implemented in firmware space
  – RX: block checksum calculation, NACK generation
  – TX: block checksum calc., block retransmissions
• Why not in driver space
  – High bus transfer delay + interrupt latency (>70 us)
• ACK, and NACK:
  – must start within 10us after receiving a frame
Implementation: Transmitter

- Kernel=>Maranello operations:
  - precompute checksums for each output packet
  - send packet and checksums to the firmware

- Firmware=>Maranello operations:
  - receive NACK: compares checksums to those precomputed
  - rebuild “special retransmission” putting pieces together

Output packet (backlogged)

<table>
<thead>
<tr>
<th>C1</th>
<th>Block 1</th>
<th>TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>Block 2</td>
<td>C2!=C2</td>
</tr>
<tr>
<td>C3</td>
<td>Block 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Block 4</td>
<td></td>
</tr>
</tbody>
</table>

NACK Header | C1 | C2 | C3

Sends this “special retransmission”

Block 1 | Block 3

Slide 81  Trento 29/4/2011  From kernel to firmware
Implementation: receiver

- **Firmware=>Maranello operations:**
  - compute checksums on packet reception
  - if frame is corrupted
    - send NACK instead of ACK, same timings
    - send corrupted packet up to kernel

- **Kernel=>Maranello operations:**
  - stores corrupted packet
  - when receives a special retransmission
    - rebuild the original packet
Other details

- Maranello & Bologna
  - We used 64-byte blocks
  
  - Checksum:
    - CRC16 is desiderata
    - OpenFWWF has not access to CRC engine
    - We used Fletcher-16/32, computing checksums on the fly
  
  - Recovered packets protected by an additional CRC32 checksum
Throughput tests

• Repeat this experiment
  – 60s UDP traffic, sta to AP (iperf), legacy => ϑ₁
  – 60s UDP traffic, sta to AP (iperf), Maranello => ϑ₂
  – Plot (ϑ₁, ϑ₂)

• Each run follows sta initialization

• Three environments
  – ATT lab
  – Maryland campus
  – Bo’s home

• Linux sta
  – Fixed channels (1, 6, 11)
  – Minstrel as RC
Throughput tests

- Reliable test?

![Graph showing throughput comparison between kernel and firmware]
Throughput tests

• Bo’s home

![Graph showing throughput comparison between Bo’s home and 802.11]

- Throughput tests
- Bo’s home
- From kernel to firmware
Throughput tests

- ATT lab

![Graph showing throughput comparison between Maranello and 802.11.](image)

- Throughput of Maranello
- Throughput of 802.11
Throughput tests

- Maryland campus
Throughput tests

- Link layer latency is reduced (shorter retr)
Maranello

**PRO**
- Partial Packet Recovery
- Backward comp. 802.11
- Link latency--
- No extra-bits in reg. packets

**ISSUES**
- NACK very long

---

BBR

**PRO**
- Partial Packet Recovery
- Backward comp. 802.11
- Link latency--
- NACK minimized

**ISSUES**
- Packet expansion
• Same comparison (preliminary results)
What to Do Next?

• Complete Bologna evaluation
• Evaluating checksum strength
  – E.g., is ok Fletcher16? Or Fletcher32 is better?
• Different block sizes
• Back-to-Back packet aggregation
• Interaction between rate control and error recovery protocols
  – Better bit rate for retransmissions
• Packet aggregation with Partial Packet Recovery:
  – For failed packets if retransmission is short
  – Instead of retransmitting only the corrupt part
  – Transmit corrupt part + new packet (if any 😊!)
• Without N-PHY we can use OpenFWWF Hack
Experiment: block error distribution

- Use “superblockanalyzer” to tx/rx traffic
- Use “codeanalyzer2” to compute distribution
- A virtual iface in monitor mode is needed on TX/RX

  $\$: sudo iw dev wlan0 interface add fish0 type monitor
  $\$: sudo ifconfig fish0 up

- On receiver
  $\$: sudo ./supercodeanalyzer -i fish0 -s -p 10000

- On transmitter
  $\$: sudo ./supercodeanalyzer -c larrybird.trento -p 10000 \ -r ./packet.pcap -B Bologna/58//fletcher16/64 \ -x 00:22:15:87:87:b3 -y 00:13:d4:bb:2c:bf -i fish0
Experiment: block error distribution/2

- Check RX screen
  - Never ending? Why?
  - Focus on “wrong blocks”? Always 0?
  - Should we have in kernel space wrong packets?

- I will manage kernel and firmware switch!

- Run again the tools...

- Finally display statistics
  
  ```
  ./codeanalyzer2 -e f16 -r packet_exp0.pcap -p /
  ./packet.pcap -x 00:22:15:87:87:b3 /
  -y 00:13:d4:bb:2c:bf
  ```
Some recent news
News from .11 hardware world

ATHEROS/1

• Atheros AR9170USB
  – USB dongle, supports a/b/g/n-draft
• Atheros released opensource fw and driver
  – Otus driver: features missing, code style--
• C. Lamparter introduced carl9170
  – Pro: Everything implemented, station, ap, monitor
  – Pro: Firmware sources can be compiled from C code
  – Issue: random firmware crashes
  • Kernel handles crashes and restart wireless subsystem
• Got in touch with C. Lamparter
  – FW/Processor is not the MAC processor
    • Resembles SoftMAC
  – FW/Processor polls the hardware (e.g., MAC), no IRQ
    • Filters packets from air by type and forwards to host on DMA
    • No way(unknown?) to build responses and send them back
    • ACKs handled by MAC processor: “Response Controller”
    • ACKs can be only disabled
    • Not a real time platform!
  – But...
    • ...CCA can be disabled 😊
    • Is this enough?
Pros

- Broadcom boards ARE realtime
- Opensource firmware available: OpenFWWF
- L2 protocol exchanges: can be deeply customized
  - E.g., Partial Packet Recovery (Maranello/Bologna MAC)

Drawbacks

- We know how to do this on b/g boards:
  - What about 11n?
- We don’t know how to handle CCA
  - Minimum space between packets is 10us (follows from .11e)
- We can’t change modulation
  - E.g., no way to modify MPDU format (i.e., PLCP is fixed)
News from .11 hardware world
BROADCOM/2

- 10/10/2010 Good news!
  - Broadcom released OS drivers
  - Builds on mac80211 linux module
  - For their latest N-PHY boards (43224/225)
    - Same architecture, firmware that drives the MAC processor!

- Drawbacks
  - No open-source firmware yet, will ever?
  - Only managed mode implemented (no AP)
  - 43224/225 boards still hard to find: we have two since last week

  - We will add RE instruments to Broadcom driver

  RE work will start soon
Original developers of Broadcom drivers for Linux
- They were(are) working on N-PHY support
- More devices included, not only latest-state-of-the-art

After Broadcom announcement
- Request to open the firmware source
- Broadcom said NO!

Got in touch with main developer R. Miłecki
- We now have an opensource driver

What about firmware...
- We are working on our own firmware: Ope(N)FWWF
- RE Broadcom Firmware: interestingly they simply added features
- So we will do building up OpenFWWF!
Projects starting soon
Issues with 802.11 DCF
Packet aggregation (helper)/1

- (Real) Packet aggregation started with .11N
  - Packets TO THE SAME dst packed & sent in single A-MPDU

802.11 DCF

802.11n

- Protocol exchange much shorter!
- Deterministic time!
Issues with 802.11 DCF
Packet aggregation (helper)/2

- (1) Unfairness in DCF channel access
  - Pack packets to all destinations in a single A-MPDU
  - AP will not lose channel access
  - AP can “steals” more than 1/N access
  - Downlink packets paced as uplinks

Backoff: may lose channel access!
(1) Unfairness in DCF channel access

Problems:
- one A-MPDU means one PLCP: rate?
- How can we send acknowledgements?
Back-to-Back packet transmission

Why?

- No .11n & b/g cards + OpenFWWF limited
  - Can build internally packet < 1000 bytes
- Fallback to clause 9.10.3 of 802.11e (2005)
  - Packets spaced by minimum possible
  - 802.11e says 10us: can we shorten this?
  - Yes! A minimum of 2us was demonstrated recently
Mesh networks
Simple Forwarder/1

- Packet in transit & single radio interface
  - Best case, no collisions & no noise: two accesses
Mesh networks
Simple Forwarder/2

- Packet in transit & single radio interface
  - Best case, no collisions & no noise: one access + $\frac{1}{2}$~
  - On rx: forwarder broadcasts the rx pkt
  - Left AP receives the broadcast and sets ACK!

Saved time!
END!