HARZER ROLLER: LINKER-BASED INSTRUMENTATION FOR ENHANCED EMBEDDED SECURITY TESTING

Katharina Bogad and Manuel Huber // firstname.lastname@aisec.fraunhofer.de

Reverse-Engineering and Offensive Oriented Trends Symposium, 28.11.2019

Fraunhofer

AISEC

FAHRPLAN

- Motivation
- A short introduction to the xtensa architecture and the ESP8266 processor
- The Harzer Roller
- Results
- Demo
- Conclusion & Future Work

MOTIVATION

- New class of devices: highly embedded, connected, „smart“ devices – Internet of Things
- Software designed like hardware: not user upgradable
  - Also, often times lax coding standards
- Abused by Mirai, …
- IoT platform vendors like Espressif or Arduino made building IoT devices much more accessible

Motivation

- However: SDK may only comes in binary form
  - Hidden vulnerabilities
- Searching for Bugs in binary-only code is hard
  - Most methods rely on the presence of debug interfaces or an MMU
- End goal: Apply black-box fuzzing, get as much state of a crash as possible
- End goal 2: get CVEs.

THE XTENSA ARCHITECTURE

- 32 Bit RISC architecture
- 16 and 24 Bit instructions
  - 16 Bit = narrow encoding (`ADDI.N`, …)
- 16 Registers: a0..a15
  - `a0` = return value, a1 = stack pointer, a2…a5 arguments
  - All registers callee-save in gcc-lx106 calling convention
- Highly configurable
  - Exceptions? Optional.
  - MMU? Optional.
  - DSPs? May be configured.

Exceptions on xtensa

- Hard-coded exception handlers in the system ROM
- Special registers are used for storing exception information
  - Virtual address that resulted in an exception
  - Exception number / reason
  - …

Programming xtensa

- Arithmetic, branches, etc as usual
  - Some narrow-encoded variants available
  - Hardware multiplication support is optional
  - No integer division, FP-division depends on floating point option
- Only near jumps: `next_pc = pc + signed imm8 + 4`
- Loading constants is complicated
  - Word-aligned, encoded, only possible to load from `pc - [4, 262141]`
  - `dangerous relocation: l32r: literal target out of range` 

THE ESP8266

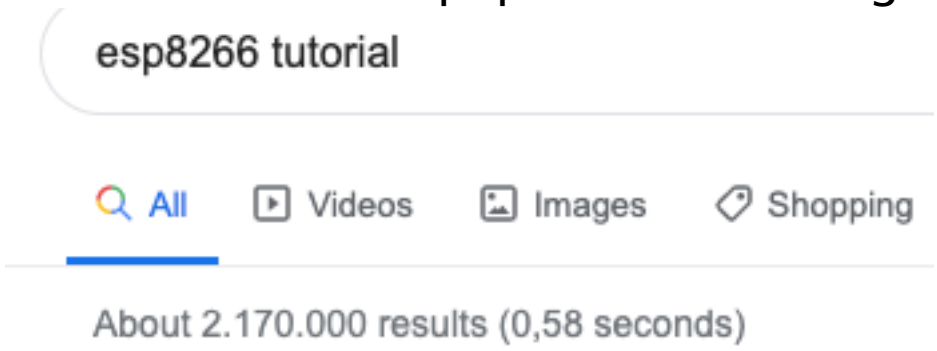
- Beginner-friendly, inexpensive IoT-Chip
- Tensilica xtensa L106 32-Bit architecture
- On-Board 802.11 b/g/n WiFi
- WiFi Mesh with espnow
- Max. current draw ~200mA under load, ~10-50mA while sleeping
- **No** JTAG 😕
- No debugging Interface 😕
- Barely any official documentation beyond a few examples 😕

ESP8266 SDKs

- Non-OS and RTOS SDKs
- Based on gcc-xtensa-lx106 compiler and calling convention
  - No register windowing
- Somewhat MIT-Licensed
  - Espressif MIT License: Permissions are granted for use on esp8266 chips only
- Uses third party Open Source technology: lwip, mbedtls, libjson
- Still: most parts are binary-only!

Other ESP8266 SDKs

- ESP8266 Open SDK: reverse engineered community effort
  - Unfinished
  - Mimics the Non-OS SDK
- Arduino ESP8266
  - For use with Arduino Studio
  - Internally using Non-OS SDK 2.2.x, but with custom heap implementation
  - This is the most popular SDK when googling for a tutorial



The SDK Problem

- Bugs in the SDK affect **all devices**, regardless of firmware
- Any remotely usable bug has a high impact
  - What if your building automation gets DoSed?
- Remote code execution can have disastrous consequences
  - What if your building automation DoSes something?
    - This has already happened: Mirai!
  - Targeted flash wear can physically destroy devices

The Debugging Problem

- Limited Resources = Limited debugging facilities
  - No jtag, no gdbstub
- Emulation only provides limited insight
- Kammerstetter, Platzer and Kastner 2014: PROSPECT Partial emulation
  - Very slow, unsuitable for timing-critical code
- Muench et. al.: What you corrupt is not what you crash
  - Memory Corruptions are often times highly invisible, and get more invisible the more embedded the device gets

The Debugging Problem: How to deal with it

- Idea: increase amount of debugging information by detecting more memory corruptions where they happen
- Introduce verbose aborts when detecting memory corruptions
- Do so without requiring source code of binary-only available libraries
- Instrument firmware **using the linker**

THE HARZER ROLLER

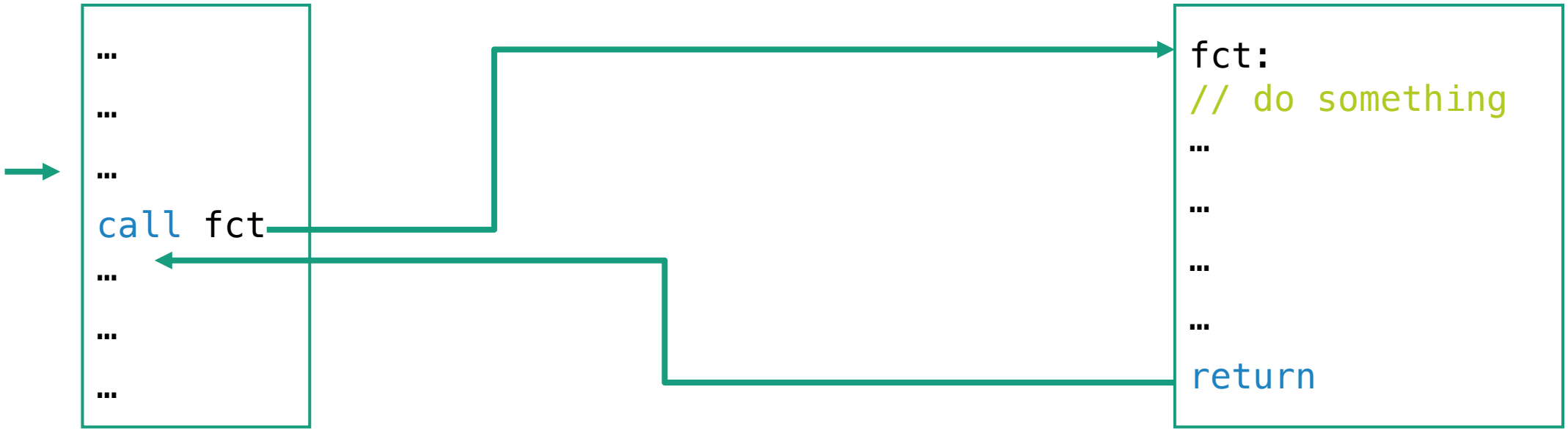
- Lat. Serinus canaria domestica
- Domestic canary bird breed from germany
- Mostly bred for its beautiful, melodious song
- Also used in german mines to detect CO gas
- And also the name-giver of our Method 😜

THE HARZER ROLLER

- Goals: Instrument Calls to and Returns from Subroutines
  - … while being completely transparent to caller and callee
  - … using as little memory and instructions as possible
  - … without recompiling the libraries to contain instrumentation
  - … without requiring the presence of a MMU
- Use the instrumentation to **trace execution flow** or **check memory for corruption**
- **Split** instrumentation in two:
  - Call-Path instrumentation
  - Return-Path instrumentation

Call-Path Instrumentation

- Normal Execution flow:



Call-Path Instrumentation

- Because flash space is tight, SDKs are usually compiled with `--ffunction-section`
- It's the linkers job to relocate the call in the final firmware image!
- We can tamper with the relocation information so the linker links instrumentation code instead of the actual function
- We call this the Harzer Roller Call-Path Instrumentation

Call-Path Instrumentation

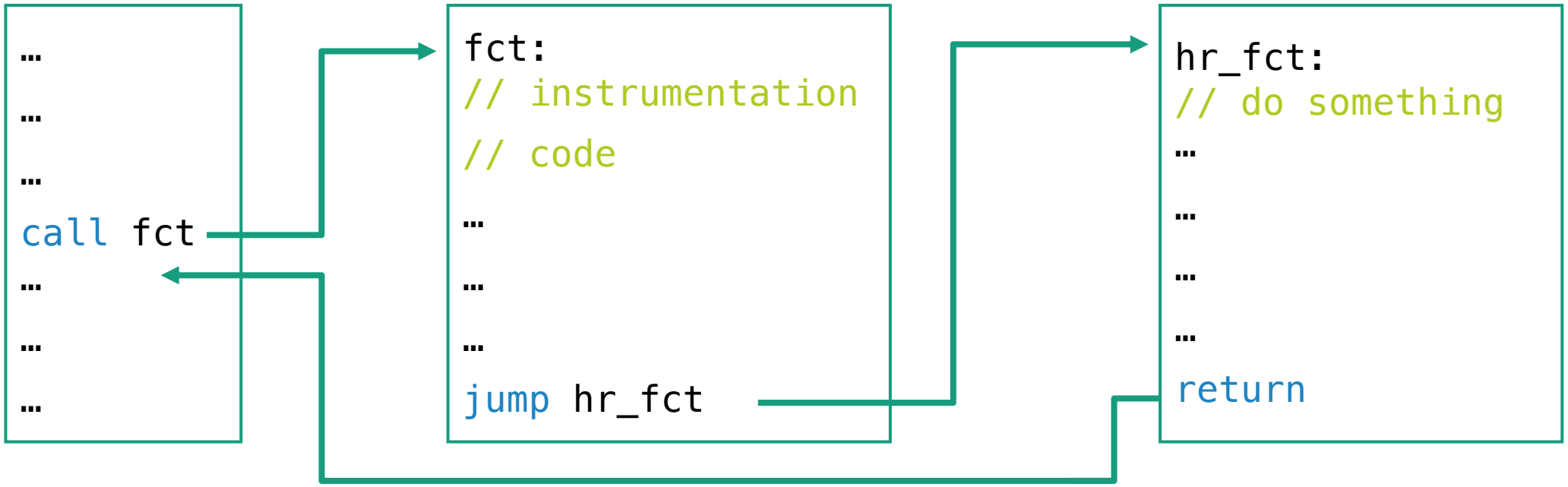
- Rename the function, but not the relocation:

```
…
…
…
call fct
…
…
…
```

```
hr_fct:
// do something
…
…
…
…
return
```

Call-Path Instrumentation

- Auto-generate instrumentation code for each function before linking:



Call-Path Instrumentation

- We can use stack space to preserve all registers we use in our instrumentation
- Function calls are possible, but may introduce endless loops
  - Instrumentation to printf() calls printf()
  - Workaround: call uninstrumented function directly
- Exemplary usage:
  - Call-Path tracing
  - Poor girl's UART strace
  - …
  - Set up Return-Path Instrumentation 😜

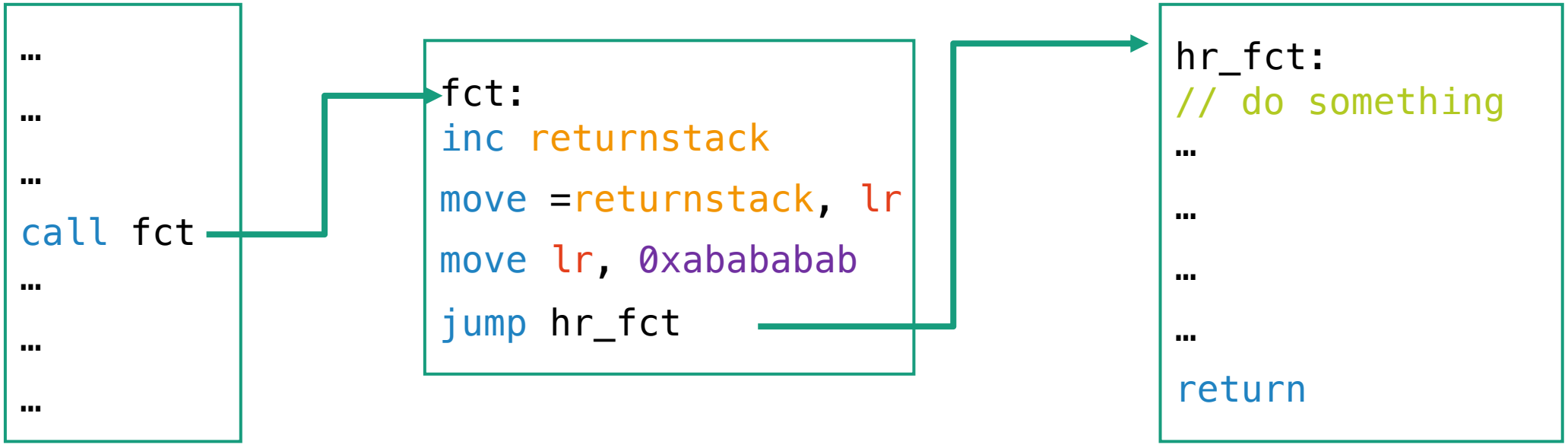
Return-Path Instrumentation

- Unfortunately, for the return path we cannot abuse the linker
- We also – in the general case – cannot just patch any return instructions as we cannot know where the function returns once it’s compiled
  - While gcc usually only emits one function exit, hand crafted assembly can do weird things
- We also need to ensure that our instrumentation runs if memory gets corrupted

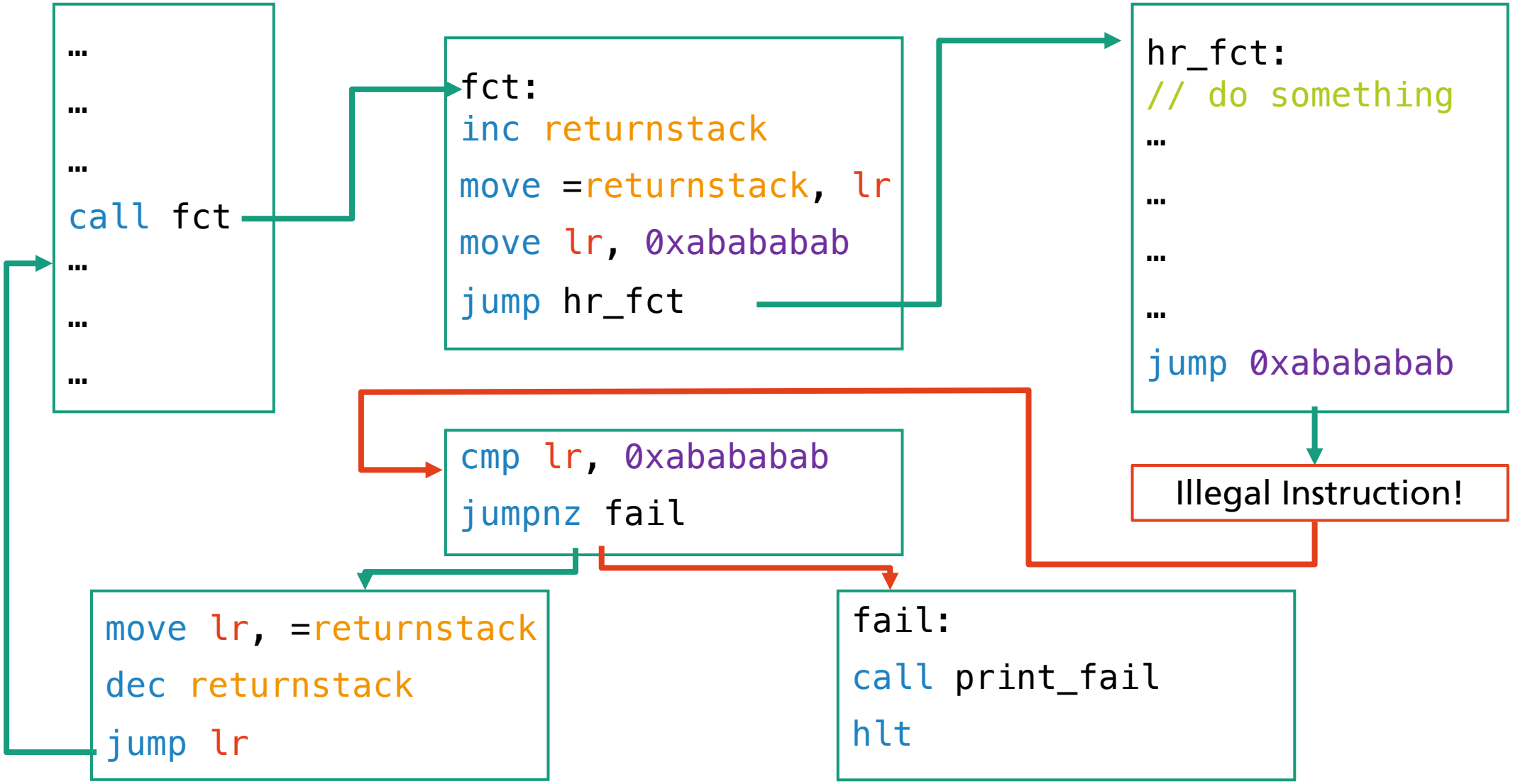
Return-Path Instrumentation

- Due to the sparsely mapped address space, corrupt return addresses usually result in Illegal Instruction Exceptions when jumped to
- Using our Call-Path Instrumentation we can change the return address seen by the called function
- We can modify it to contain a value that intentionally results in an Illegal Instruction Exception
- By registering a custom exception handler we can introduce instrumentation code
- We can recover from such an exception if we use the call-path instrumentation to build a call stack
  - This need special handling in multicore environments where tasks can switch CPU cores
  - In this case, we need to build one call stack per task

Return-Path-Instrumentation



Return-Path-Instrumentation



Return-Path Instrumentation

- Return stack is a FIFO-Queue of information about the last called subroutine
- Must save: real return address
- Optionally save additional metadata (at the cost of RAM):
  - Function name
  - Register state(s)
  - ...
- Contents largely depend on the instrumentation

IMPLEMENTATION ON THE ESP8266

- Experiments ran with SDK 3.0 (git commit 2f9e0bb)
- Little space available
  - Instrumentation code must be size-optimized
- Call- and Return-Instrumentation
- Be as transparent as possible
  - Unfortunately, we still need to overwrite one register
  - We can use the frame pointer (a15) for that as it isn’t needed in the called function
- We aim to open source this implementation

Size improvements

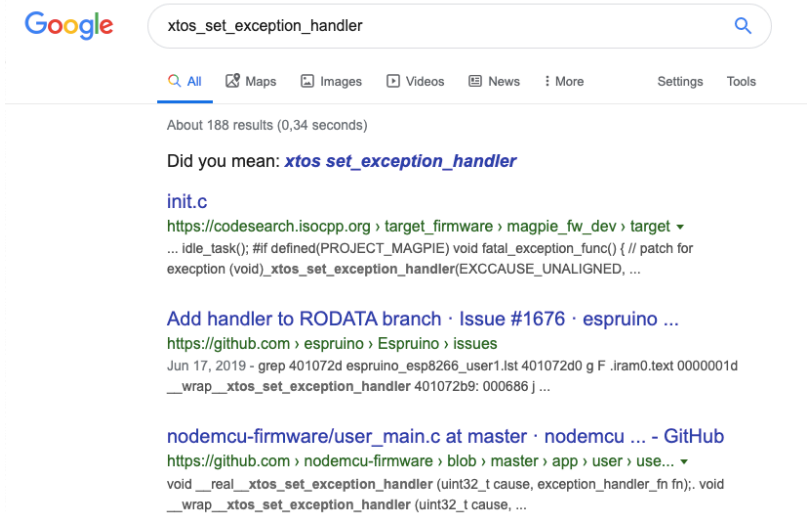
- Only little code of the call-path instrumentation actually depends on the function
- Split the code into a portion that’s emitted once per instrumented function and one part per instrumentation
  - Using narrow-encoded functions we can squash the first down to 32 + len(function_name) Bytes
  - Function name gets written to the call stack to be dumped if overflow occurs
- Hand-crafted assembly goes long ways here
  - Also, care must be taken to not mess up the registers

Registering the exception handler

- No public API available
- However…

```
katharina@annaberg ~/ESP8266_NONOS_SDK-3.0/ld % cat eagle.rom.addr.v6.ld | grep exce
PROVIDE ( _xtos_set_exception_handler = 0x40000454 );
PROVIDE ( _xtos_unhandled_exception = 0x400dc44 );
```

- Winner-Winner, chicken dinner:



Registering the exception handler

- Unfortunately, this never worked as we thought it would.
- The hardware-exception handler is in the closed-source ROM BLOB
- So we applied black box testing
- Looking at the RAM at runtime we found the exception handler table to be located at `0x3fffc000`
- Directly overwriting the first entry gave us low-level access to the exception handling ☺

Registering the exception handler

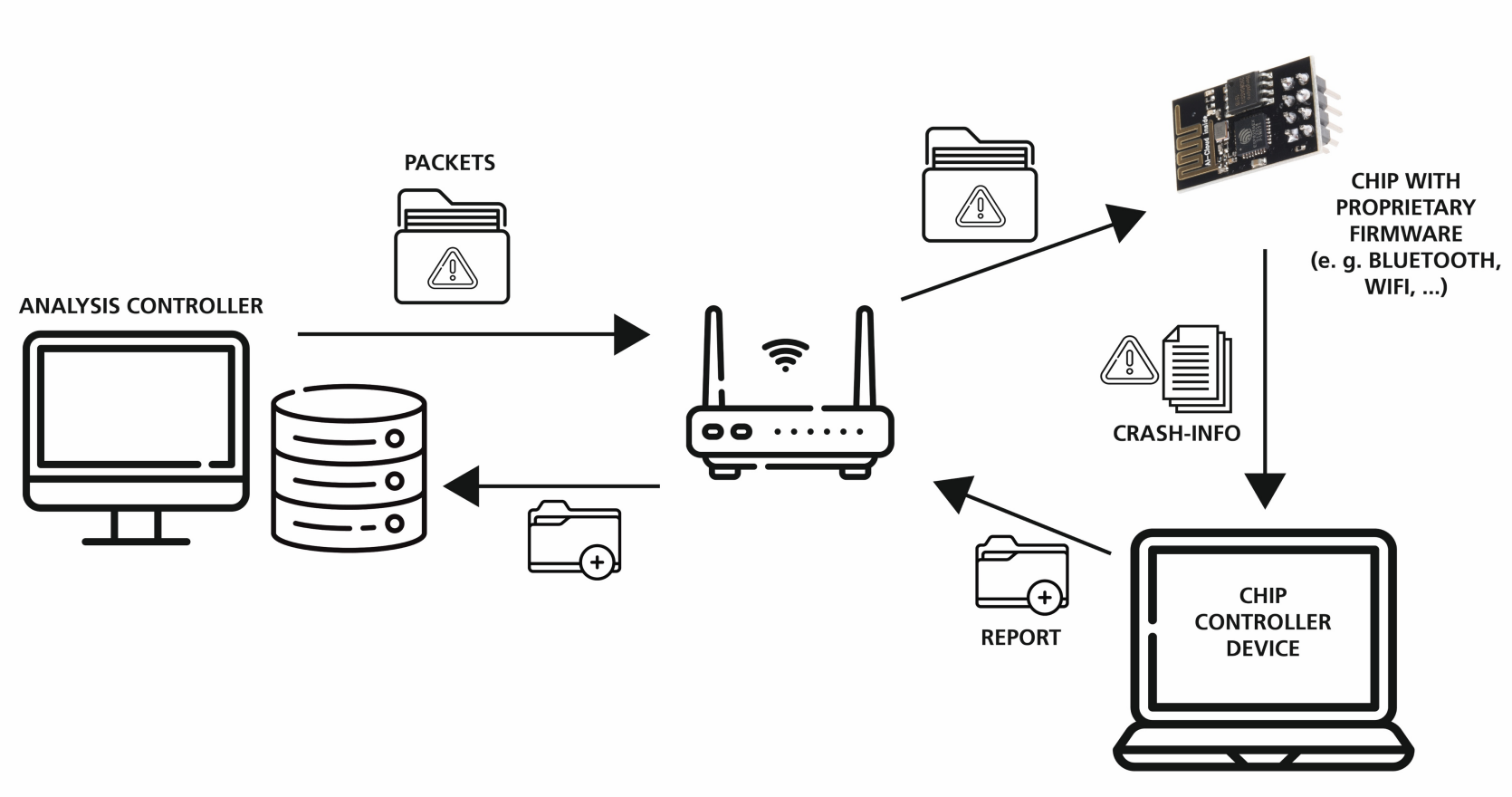
- Obviously, we must register our custom handler before the first protected function returns
- Idea: specify a master function and use call-path instrumentation to hook it and register our handler
- The master function could be anything, as long as it is called at least once
- What function to hook is hugely dependant on the instrumented functions
  - Application-Level code could be fine with user_pre_init
  - Some heavily relied functions (e.g. memory management) could need a very early hook function

EVALUATION

- Test program, XOR as a Service containing stack based buffer overflow
- Two goals:
  - Call tracing using only the call path instrumentation
  - Stack integrity checking using the return path instrumentation
- Vulnerable part:

```
void ICACHE_FLASH_ATTR shell_tcp_recvcb (void* arg , char* pusrdata, unsigned short
length)
{
   struct espconn* pespconn = (struct espconn*) arg;
   char xorbuf [20];
   char* x;
   ets_memcpy (xorbuf , pusrdata, length);
   …
}
```

DEMO



Size- and performance penalties

- Non-trivial size increase
  - Min. 32 byte per instrumented function
  - Additional space for instrumentation code
  - We have seen increases of up to 150% per archive (object file structures included)
- However, the performance overhead is somewhat constant
  - If no UART printing is involved, constant time can be achieved
  - Because of the limited architecture without branch prediction, time overhead can be calculated for a specific instrumentation

Limitations

- The Harzer Roller is no security mitigation
  - Overwriting the return address is not protected, a “good“ address would never hit our handler
- The instrumentation has to be tailored on a case-by-case basis; there is no general detect-it-all
- Currently only single core, single thread is supported
- Flash size restrictions prevent us from instrumenting the whole firmware at once
  - esp8266 only supports up to 1024 KiB of IROM, regardless of SPI flash size

FUTURE WORK

- Tooling to track found crashes, make them easily searchable and indexable
  - Track crashes against a variety of SDK versions
  - Database of object files in a given SDK to better understand the low-level connections
- Fuzzing environment to thoroughly test all public API endpoints
  - Memory checker that detects overflows in malloc()ed memory
  - Fuzz based on implemented network protocols: DNS, 802.11, espnow, …
- Port our work to the ESP32 and other IoT platforms

CONCLUSION

- Ability to inject more debug output capabilities
- Only require object code, therefore able to instrument the binary-only SDK of the ESP8266
- Used in fuzzing setup to capture crashes more local to the actual corruption
- Open source our implementation in the near future

THANK YOU. QUESTIONS?

- Further Reading:
  - Muench et. al: What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices, https://doi.org/10.14722/ndss2018.23176
  - Corteggiani, Camurati and Francillon: Inception: System-Wide Security Testing of Real-World Embedded Systems Software, ISBN: 978-1-939133-04-5
  - Kammerstetter, Platzer and Kastner: Prospect: Peripheral Proxying Supported Embedded Code Testing, https://doi.org/10.1145/2590296.2590301
  - Song et. al: PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary, https://dx.doi.org/10.14722/ndss.2019.23176