Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing

Katharina Bogad
Fraunhofer AISEC
Garching near Munich, Germany
katharina.bogad@aisec.fraunhofer.de

Manuel Huber
Fraunhofer AISEC
Garching near Munich, Germany
manuel.huber@aisec.fraunhofer.de

ABSTRACT

Due to the rise of the Internet of Things, there are many new chips and platforms available for hobbyists and industry alike to build smart devices. The SDKs for these new platforms usually include closed-source binaries containing wireless protocol implementations, cryptographic implementations, or other library functions, which are shared among all user code across the platform. Leveraging such a library vulnerability has a high impact on a given platform. However, as these platforms are often shipped ready-to-use, classic debug infrastructure like JTAG is often times not available.

In this paper, we present a method, called Harzer Roller, to enhance embedded firmware security testing on resource-constrained devices. With the Harzer Roller, we hook instrumentation code into function call and return. The hooking not only applies to the user application code but to the SDK used to build firmware as well. While we keep the design of the Harzer Roller generally architecture independent, we provide an implementation for the ESP8266 Wi-Fi IoT chip based on the xtensa architecture.

We show that the Harzer Roller can be leveraged to trace execution flow through libraries without available source code and to detect stack-based buffer-overflows. Additionally, we showcase how the overflow detection can be used to dump debugging information for later analysis. This enables better usage of a variety of software security testing methods like fuzzing of wireless protocol implementations or proof-of-concept attack development.

CCS CONCEPTS

• **Security and privacy** → *Mobile and wireless security*; **Software security engineering**; *Software reverse engineering*.

KEYWORDS

linker-based static instrumentation; binary instrumentation; embedded firmware instrumentation; SDK analysis; software testing; fuzzing

ACM Reference Format:
Katharina Bogad and Manuel Huber. 2019. Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing. In *Proceedings of Reversing and Offensive-oriented Trends Symposium (ROOTS '19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ROOTS '19, November 28–29, 2019, Vienna, Austria
© 2019 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of Reversing and Offensive-oriented Trends Symposium (ROOTS '19)*, https://doi.org/10.1145/nnnnnnn.nnnnnnn.

1 INTRODUCTION

In recent years, significant advances have been made in embedded technology. Cheap computing technology paired with pervasive internet connectivity led to the rise of a new class of embedded devices and computing technologies, which have been summarized under the name Internet of Things (IoT).

Historically, these highly embedded computing devices have a bad track record regarding their security and have been a prime target for hackers of all sorts. Oftentimes these devices are built to fulfil a designated task and either employ no or insufficiently secure means to upgrade the embedded firmware. Although this is starting to change, the firmware life cycle poses a major challenge. As these devices are often spatially dispersed and deployed in large quantity, the firmware needs to be especially secure. Any attack that could reliably achieve execution of arbitrary code also has the ability to permanently destroy deployed hardware, for example through targeted Serial Peripheral Interface (SPI) flash wear-out of the boot sector. A common off the shelf SPI flash memory chip, for instance, the Winbond W25Q128V, is specified with „more than 100,000“ program and erase cycles [1]. When targeted at a relatively small number of flash cells, flash wearing can be achieved in a few minutes. Depending on the specific device in question, other DoS attacks or device misuse might be possible as well [4].

In the past, IoT devices have been leveraged for large scale attacks, for example with the Mirai botnet [10]. Efforts have been made to secure the development process of firmware, however little attention has been drawn on coprocessors of such an IoT system. A broadly used communication/Wi-Fi coprocessor with over 100 million devices in use across the globe [18] is the *ESP8266* family of the Chinese company Espressif, or its successor, the *ESP32*. These chips either employ a standard AT firmware which provides network connectivity over a serial UART connection or can alternatively be programmed with custom firmware using Espressif's SDK. While the SDK contains some open source components, various BLOBs linked into every built firmware remain. There have been efforts to reverse engineer these parts [15] to establish a fully open source stack, but some of these BLOBs are still required.

The result is that even if the main application code is correct and free of bugs, the SDK could potentially introduce vulnerabilities that are highly invisible due to the nature of binary code. While some vendors open source their SDKs – or at least make it available under a source available license – often times, the SDK is provided in binary-only form; thus preventing the use of source code instrumentation techniques. To actively search for vulnerabilities in such a scenario, black box testing of these closed-source components needs to be conducted. However due to the constraints of embedded systems, usually there is no MMU available. This has

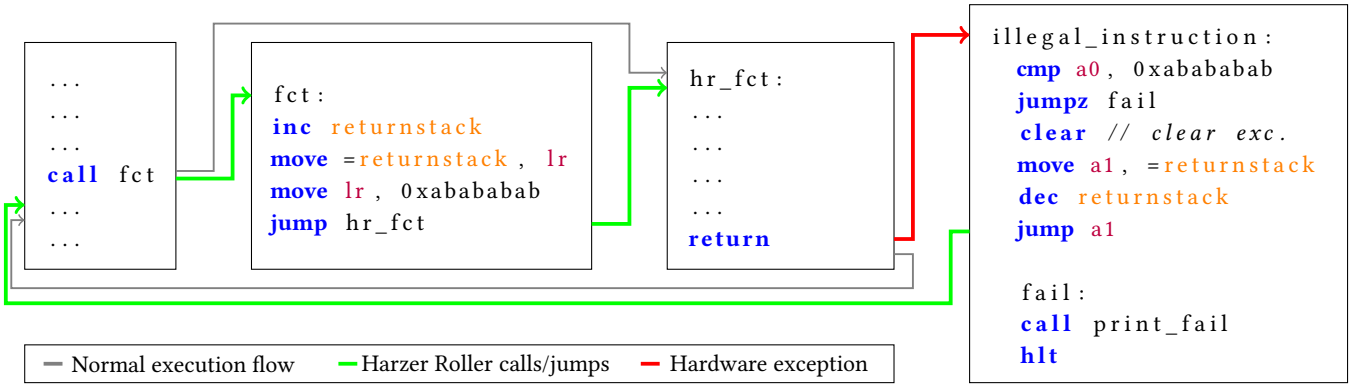

Figure 1: Call and return control flow hijacking with the Harzer Roller in pseudo-assembly.

consequences unfavourable for vulnerability testing: (1) there is no means to intercept memory accesses and (2) most of the times there is no way to trap accesses to unmapped memory (such an access either returns a repeated pattern of fixed values, completely random values, zero or any combination of those). This makes memory corruptions highly invisible, as pointed out by Muench et al. [13]. For instance, memory corruptions might not immediately trigger a fault but only become obvious at a later crash or malfunction of the device. Furthermore, source code instrumentation using stack canaries or various sanitizers is usually neglected to save the limited computational resources.

To increase visibility of memory corruptions, happening for example through stack overflows, various approaches have been proposed before: (1) full system virtualization, where an IoT chip in its entirety is replicated in a virtual machine like in [3], which is oftentimes difficult to achieve because the exact environment and all physical interactions (with SPI flash, GPIO pins, ...) are hard to simulate, (2) partial virtualization like PROSPECT [9] which solves this problem by forwarding hardware and pin access to the real device but impacts execution speed or traditional debug interfaces like the industry-standard JTAG port, which is – especially in commercially available chips – oftentimes not available. Depending on the exact virtualization setup, having access to the source code might be a requirement.

In this paper, we present a novel method to instrument object-only code that is commonly found in SDKs at the linker stage. We present an approach to instrument function calls as well as returns from these functions, using only means provided by the linker. Our approach is feasible even without source code access of the object files to be linked. We showcase our method on an example platform, the Espressif ESP8266 based on the xtensa Instruction Set Architecture (ISA), and show function call tracing and detection of overwritten saved return addresses as two possible real-world applications of the our method, as well as their application in automated vulnerability discovery via e.g. fuzzing. Inspired by stack canaries, we call our method *Harzer Roller*[1]. Furthermore, we discuss potential attack vectors specific to the ESP8266.

[1]https://en.wikipedia.org/wiki/Harz_Roller

2 LINKER-BASED INSTRUMENTATION

In this section, we describe the design of the Harzer Roller. We first give a brief overview, continue by explaining the Harzer Roller's call-path instrumentation, and, based on this technique, introduce return-path instrumentation.

We design the Harzer Roller in a way that allows us to hook the control flow of executables on calls and returns to and from subroutines, respectively. To achieve this, we modify the object files before linking the final executable so that the linker relocates calls to an instrumented function to an assembly sequence of our choice (see Subsection 2.1), instead of modifying the pro- and epilogues of the function themselves. At the end of our injected assembly sequence, we jump to the called function. Our injected assembly sequence may override the link register with a canary value that is guaranteed to result in an illegal instruction exception when jumped to. We introduce a fault handler that can recover from such an exception, thus hooking into returns from subroutines (see Subsection 2.2). We ensure that our method is able to preserve semantics between function calls and returns.

Figure 1 depicts this linker-based binary instrumentation which we explain in the following in more detail. It is noteworthy that the instrumentation of the return path is optional and the Harzer Roller can also be used to instrument the call path only.

2.1 Call-Path Instrumentation

In our approach, we treat functions as black boxes. We thus need to preserve all registers and stack contents throughout the instrumentation. These are considered input to a function. A special case to this is the link register which should never be modified by the called subroutine.[2] Modifying the contents of this link register however is the target of an attacker attempting to gain control over the execution flow.

To instrument any function that gets called by either another object or the user application, we need to wrap the function and inject code into the execution flow. For function entry (i.e. upon a call to a function), we leverage `gcc`'s `-ffunction-sections` flag, which places every function in a source file in its own section. While

[2]Whenever we refer to the link register, we mean the place the saved return address is stored. While on some architectures a CPU register is used, the return address may be placed on the stack on other architectures.

our method also works without this flag, it is more effective when enabled because we have a higher number of relocations to hijack. We inject the code into the execution flow in three steps: First, we rename the symbol in the object file that should be instrumented and prefix it with some value, for example `hr_`. Obviously, we need to make sure that this renaming does not introduce conflicts with already existing symbols in the object files to be linked. We then re-add all symbols we renamed in the first step, but instead of pointing to the concrete implementation provided in the object file, we leave them as `UNDEFINED` in order to be filled in by the linker later. If the function gets called from another object file, this shifts the execution flow from the grey direct call in the left half of Figure 1. Last, we look for all references to the re-named functions in all sections with relocation information and rewrite those references that point to the renamed symbols to point to the added imports instead. In this way, we shift the calls within the object file to the Harzer Roller control flow path. We then generate another object file that contains all wrapping code needed for the instrumented object file and finally ensures the jumps to the real, renamed function. This completes the upper green call path depicted in Figure 1. To generate a valid executable file, we link both object files: the target we modified and the file containing our injected code.

2.2 Return-Path Instrumentation

So far, we injected code into the call path (left half of Figure 1). To be able to inject code into the return path too, we ensure that the Harzer Roller catches all return paths. For example, it is not uncommon to have a function check its input at the beginning and return early when it detects invalid input. These return statements do not necessarily need to be in the same basic block. Therefore, any function can have any finite number of return points. In our method, we rely on the fact that some instructions on a microprocessor are not valid and guaranteed to cause an exception, which may be handled by an exception handler. We can use this to craft a special return address that will generate such an illegal instruction exception and continue to do so even if that return address is partially overwritten (e.g. by placing the target address into unmapped memory, which usually generates an exception when jumped to).

We use the call-path instrumentation described in the previous section to modify the link register within the injected wrapper function before control is passed to the instrumented function. As depicted in Figure 1, we overwrite the contents of the link register with a special canary value, that is chosen in a way to guarantee an invalid instruction exception to occur when jumped (or returned) to. As depicted with a red arrow in Figure 1, this shifts execution flow from the direct return into the exception instrumentation path.

Since we are able to inject code before the execution flow shifts to the called function and also on every return path, we can now transparently modify input to the function and validate the functions output and/or rewrite it. To pass control flow back to the function that called our Harzer Roller instrumented function, we need to restore the state of execution. This mandates two requirements: first, all instrumentation in the return path must not overwrite any registers containing return values, callee-saved registers or stack contents still in use. Second, because the link register gets overwritten to be able to hook this return path, we need to restore the address that the return should jump to. We do this by saving the link register to a memory region not used by either stack or heap before overwriting it. We call this structure *return stack* as it is a FIFO queue of saved return addresses. Depending on the concrete instrumentation done in this step, we could also save additional metadata to this structure that is only known upon call, but needed in the return path. An example of such data would be the name of the called function.

In the following section, we provide a proof of concept implementation of the Harzer Roller. While it is targeted at a specific platform, our method is generally applicable to other platforms as well.

3 IMPLEMENTATION ON THE ESP8266

For a showcase implementation of our concept we chose Espressifs ESP8266 chip as a platform. Its MIT-licensed [11] SDK, which is available online [12], contains numerous BLOBs to dissect. Specifically, we ran our experiments with firmware built on top of version 3.0 (`2f9e0bb`) of the ESP8266 NONOS SDK.

The ESP8266 is based on the Tensilica xtensa ISA family [19]. As the ISA is highly customizable, it can be configured with or without certain features like MMU, JTAG or various DSPs. The exact configuration of the architecture for the ESP8266 processor is unknown, however the general consensus seems to be that very few features above the base package are included; especially no JTAG or MMU features [7]. We first describe our implementation for call path instrumentation and function wrapping. We then elaborate on how to obtain the exception table on the ESP8266. We require this table for the instrumentation of the return path, which we describe subsequently.

3.1 Call Path Instrumentation and Function Wrapping

Because only limited space is available on the SPI flash chips supported by the ESP8266 SDK, we must take care to keep the overhead of function wrapping as low as possible. In the xtensa ISA, we do this by taking advantage of the narrow-encoded instructions, like `addi.n`, which only take up two instead of three bytes.

We also observe that only a small portion of the actual wrapping code depends on the function that is called. We can thus save additional space by separating the wrapper code into an independent part that we only put once in the resulting firmware file and by generating as less instructions as possible for each wrapped function. Of course, the resulting size depends on the functionality that should be achieved with the instrumentation, however we assume that most of this size cost can be located in the function independent part of the handler.

As the Harzer Roller injects itself into function calls, it must be completely transparent to the caller and the callee (with the obvious exception being the return address); in particular once the return address is saved, the stack pointer and all registers except `a0` – the link register of xtensa – must be the same as without instrumentation. This requires space in memory to perform these operations. We solve this problem by allocating a temporary stack frame to save all used callee-save registers.

Still, we need one caller-save register to hold the address of the wrapped function. We chose `a15` as this is inherently a caller-save register. Therefore, it can be modified freely by the called function, which is guaranteed not to depend on this register.

The second part of the wrapping code is relatively straight forward: we save the actual return address and the associated information to the current cell of the return stack, increase the pointer to the return stack, restore the registers `a1` to `a3` and jump to the called function. We set `a0` to the canary value, in our case `0xdeaddead`. While the exact value of the canary is not important, we need to choose it in a way such that we can guarantee that it reliably generates an exception. We settled on this particular value because it has the added benefit of being very easy to spot in a debugger or memory dump and rather unlikely to be incidentally encountered.

3.2 Registering the Exception Handler

To make use of the return-path instrumentation, we need to register our own exception handler for illegal instruction exceptions. Unfortunately, there is no documented way of doing this with an API function of the SDK and neither the SDK API Reference [20] nor the architecture manual [8] specify the mechanism by which exceptions are actually handled in the ESP8266 core. However, the ISA manual specifies that any implementation of the exception option needs to specify a user, kernel and double exception vector. Fortunately, these vectors are specified in the default linker script for the platform [17] as `0x40000030` for the kernel and `0x40000050` for the user vector, respectively.

From the memory map [6], we deduct that this location is in the processor's internal ROM which cannot be written to. Any code that resides in this ROM is the same across all ESP8266 devices with the same revision. Unfortunately, the license of this code is not clear – for legal reasons we therefore assumed it to be closed source and applied black-box testing.

Dumping and examining the ETS system RAM revealed an array of function pointers at `0x3fffc000`, which turned out to be the kernel exception handler tables. Each entry of that table corresponds to one exception of the kernel exception class. The index of the table refers to the cause of the exception as described in the ISA manual.

Using this information, we overwrite the first entry of the exception table (the `IllegalInstructionCause`) with our custom exception handler function, thus ensuring execution of the Harzer Roller exception path (see `illegal_instruction` in Figure 1).

To be able to make use of the return path instrumentation, we need to ensure that our custom handler is registered before the first return of a protected function happens. Depending on which functions in the built firmware are instrumented, we searched for a way to move the hooking code to a different function for different firmware builds. This problem is similar to what led to the introduction of so-called Master Codes [14] in cheat devices like Datels Action Replay v3 for the GameBoy Advance. In essence, their system allowed end users to overwrite values in the games memory, thereby altering values like health or experience gained. To overwrite these values, a routine in the cheat modules ROM was used. The game-specific master code was then used to dynamically patch the original game's ROM to inject a jump to the cheat routine

```
rsr.epc1 a3                                  1
L32R a2, canary                              2
SUB a2, a2, a3                               3
BEQZ a2, good                                4
L32R a0, returnstack                         5
ADDI.N a0, a0, -12                           6
L32I.N a2, a0, 0                             7
L32R a0, stack_chk_f                         8
callx0 a0                                    9
good:                                        10
L32R a0, returnstack                         11
ADDI.N a0, a0, -12                           12
L32R a15, stack_ptr                          13
ADDI.N a15, a15, 4                           14
S32I.N a0, a15, 0                            15
L32I.N a15, a0, 12                           16
L32I.N a0, a0, 8                             17
L32I.N a2, a1, 20                            18
L32I.N a3, a1, 24                            19
L32I.N a4, a1, 28                            20
ADDI.N a1, a1, 256                           21
wsr.epc1 a0                                  22
rfe                                          23
```

Figure 2: Implementation of our illegal instruction exception handler in xtensa assembler.

every few frames. Similar in spirit, our implementation allows the specification of a master function which will be hooked with the exception handler installation routine. For simplicity, we use the same hooking idea like in the call path instrumentation.

3.3 Return Path Instrumentation

As described in Section 2, each time a Harzer Roller-instrumented function returns, the processor generates an illegal instruction exception (see Figure 1). In this part, we describe how we handle these illegal instruction exceptions and outline a sample implementation of a stack corruption detection with Figure 2.

On the xtensa architecture, the address which triggered the exception is stored in a special register. We load this address and compare it to the Harzer Roller canary which we store in a fixed location in RAM. We only use the caller-saved registers `a2` to `a4` as these are saved on the stack by the calling function and can thus be utilized without affecting further execution. This is reflected in lines 2-4 of Figure 2.

If the canary check fails, we do not return from the exception and may hence freely utilize any register. Because an overflow must have happened, those values are considered to be invalid in any case. As with traditional stack canaries, we invoke a special function `stack_chk_fail`, which handles the abort and dumping of the execution state (lines 5 to 8 in Figure 2). As this function is part of our instrumentation implementation, we chose a human-readable format that dumps all registers (except `a0`, which cannot be recovered), and roughly 384 bytes of stack around the current stack address (`a1`).

```
1  void ICACHE_FLASH_ATTR
2  shell_tcp_recvcb(void *arg, char *
       pusrdata, unsigned short length)
3  {
4    struct espconn *pespconn =
5      (struct espconn *) arg;
6    char   xorbuf[20];
7    char   *x;
8
9    ets_memcpy(xorbuf, pusrdata, length);
10   ...
11 }
```

Figure 3: Excerpt of a vulnerable test program.

In the good and usually executed path (starting from line 10), we restore all registers to their saved values. Subsequently, we ensure that the Harzer Roller is transparent to the instrumented software. We load registers `a2` to `a4` from the stack. We store register `a0`, the new return address, on the topmost cell(s) of our return stack. We adjust the stack pointer in `a1` to `a1` + 0*x*100 in order to restore the stack frame of the function. Finally, we overwrite the special register containing the fault return address with the valid return address we saved when calling the instrumented function.

We now explore two possible applications of the linker instrumentation: execution tracing and detection of memory corruption.

4 EVALUATION

In this section we evaluate the Harzer Roller using the example implementation presented in Section 3. Specifically, we demonstrate that execution flow tracing and crash dump information extraction can be achieved using our method. Additionally, we investigate the implications of the Harzer Roller regarding size increase of the resulting binary and execution time. Finally, we showcase a fuzzing setup that relies on the Harzer Roller to collect crash information of a fuzzed ESP8266 device.

For our tests, we implemented a simple xor-as-a-service test program, where anything that retrieved is byte-wise XORed with `0x42` and then sent back. It contains a stack-based buffer overflow vulnerability (see Figure 3) to simulate a real-world scenario in which an attacker gains control of the device by overwriting the saved return address on the stack. We then compiled this program using our call- and return path instrumentation.

4.1 Execution Tracing

We utilized the call path instrumentation of the Harzer Roller to insert a dump function into every indirect function call of the object file that was compiled from our test code. This dump function has the full program state at the time of the call available. Because any call to (UART) printf-functions that can print state information would clobber any used registers, we took care to fully save (and restore) the contents of all registers. However, this is a necessity specific to the xtensa architecture, as its calling convention does not make use of any callee-save registers. Still, the needed stack space for these operations may limit the applicability of our dump

```
(0x3ffe8070) a0=0x40229fb5 a15=0x3ffef500   1
     name='tcpserver_connectcb' sp=3
    ffffd74
tcp connection established                  2
(0x3ffe8070) a0=0x4022a974 a15=0x0 name='   3
    shell_tcp_recvcb' sp=3ffffd84
```

Figure 4: Sample output of the call instrumentation of our test program. The first address is the current location of the topmost entry of the return stack used in return-path instrumentation.

```
*** STACK SMASH DETECTED***                          1
returning from function shell_tcp_recvcb             2
halting execution. pc=23232328, canary=               3
    deaddead
                                                      4
Register state:                                       5
a0=(unk)          a4=00000000       a8                6
    =00000000     a12=00000000
a1=3ffffd90       a5=00000000       a9                7
    =00000000     a13=23232323
a2=000002d0       a6=00000008       a10               8
    =00000000     a14=23232323
a3=00000000       a7=46464646       a11               9
    =00000000     a15=40217868
                                                      10
stack dump at 3ffffd00:                               11
0x3ffffd00: 00 00 00 00 23 23 23 23  00               12
    00 00 00 98 fc ff 3f
...                                                   13
0x3ffffd50: 23 23 23 23 23 23 23 23  a9               14
    02 00 00 00 79 21 40
0x3ffffd60: 23 23 23 23 23 23 23 23  23               15
    23 23 23 23 23 23 23
0x3ffffd70: 23 23 23 23 23 23 23 23  23               16
    23 23 23 23 23 23 23
0x3ffffd80: 23 23 23 23 23 23 23 23  23               17
    23 23 23 23 23 23 23
0x3ffffd90: 23 23 23 23 23 23 23 23  23               18
    23 23 23 23 23 23 23
...                                                   19
```

Figure 5: Example of crash dump information that can be extracted using the Harzer Roller.

function. Under tight memory constraints, e.g. when handling non-trivial recursion, our injected code could lower the limit of the maximum possible recursion count. For an example of the output of our test program, see lines 1 and 3 of Figure 4. We use this information to track values across functions and non-public API endpoints. This enables us to have a better understanding of the inner workings of the SDK besides the public reference documents.

4.2 Crash Dump Information

We also utilized the return-path instrumentation to collect crash dump information if the saved return address does not match our canary value. As pictured in Figure 5, we are able to recover and print all registers (except `a0`) and the stack frame of the faulty function. We triggered this dump by exploiting the test vulnerability through sending a large amount of `a` characters. We see that the stack pointer at the time of failure points to `0x3ffffd90`, and from the generated assembly we can deduce that our initial, overflown buffer, was located at `0x3ffffd60`. The return address was saved to `0x3ffffd8c` and was overwritten. Because we saved the function name that was called to a stack structure during a function call using our call-path instrumentation, we can identify the function that contained the fault.

4.3 Binary Size and Performance Overhead

There is a non-trivial size increase when using the Harzer Roller; making complete instrumentation of all SDK functions in real-world applications not practical. However, the size increase varies greatly between the different libraries (see Figure 6), so it is possible to choose exactly those of interest. Generally, a higher size increase is directly related to better code coverage of the instrumentation. The best coverage can be achieved when compiling with gccs `-ffunction-sections` argument as each function is placed into its own section, thus retaining a symbol name that can be instrumented. In this scenario library-internal functions that are not part of any exposed API can be fuzzed as well.

A special case considering size increases is `libgcc.a`. Although this library is relatively big (about 73 KiB), it's size increase is only 4, 87%, making it the library with the smallest size increase. This is due to the library mostly containing softmath library code which will not be instrumented at all.

Even with optimizations and hand-written assembly using narrow instruction encodings to reduce the size of the wrapping code, the introduced overhead is still large for embedded systems and especially the ESP8266. Currently, the SDK of our example platform supports only up to 16 Mbit SPI flash chips which is not large enough to instrument all SDK functions in a given application firmware at once.

As a result, we can only instrument parts of the firmware at one time when fuzzing, for instance. We automate the instrumentation process, automating the unpacking of the contents of a given archive (if necessary), renaming the symbols in the object files, and generating wrappers as described, compiling them and re-packing everything to an archive that can be used for linking. All this can be configured either in the projects Makefile or via environment variables for great flexibility.

The execution time overhead is not tied to the size increase in any form, as the size increase stems mainly from the addition of function-dependent hooks that get added to the archive. Instead, the overhead scales with the amount of code that is used in the instrumentation. In our case, the biggest time sink was the printing to UART using a rather low baud rate, which was fine for our test application. However, when instrumenting time-critical code in future work we need to be careful as to not break timings on e.g. the physical 802.11 layer.

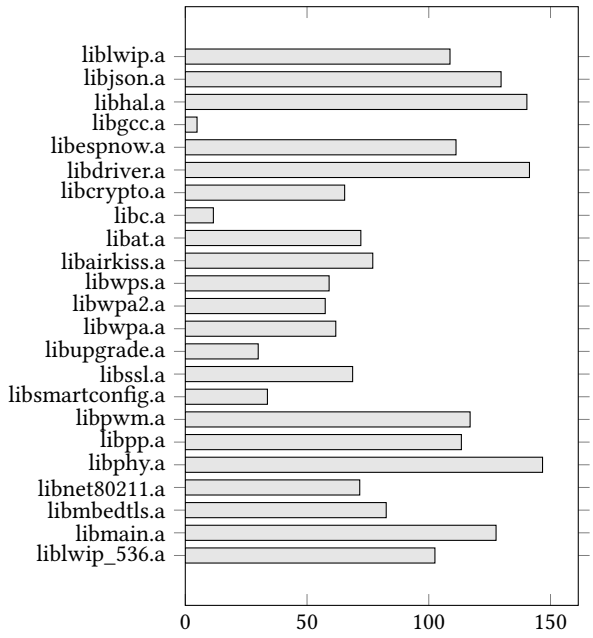

Figure 6: Increases in size in percentage of libraries instrumented with the Harzer Roller.

4.4 Fuzzing Setup

In general, there are three modes of operation supported by the ESP8266: Access Point, Station and Mesh. For our tests, we wanted to mimic a typical household IoT-Scenario. We deployed a common off the shelf router running OpenWRT to which the target ESP8266 in station mode connects. We connected the host device, a desktop computer, running the fuzzer via a standard ethernet connection to the router. The whole setup is depicted in Figure 7.

Our fuzzer is based on BooFuzz [5], a popular python layer 2/3 protocol-based modelling fuzzer. BooFuzz is capable of fuzzing various protocols of the ESP8266's network stack. For our testing purposes and to show the effectiveness of the Harzer Roller, we built a sample vulnerable binary outlined in Figure 3. This binary contains a vulnerability that must be found while fuzzing with the Harzer Roller.

We used our return-path instrumenting to print crash output and memory dumps from the ESP8266 to the UART serial connection. For efficient fault capturing, we need to capture the dumped information. Additionally, we must ensure that we can reboot the ESP8266 from every state, even if it is completely hung up, without human interaction (e.g., pulling the power cable).

To address both problems, we used a target device controller board, in our setup a sparkfun ESP32thing, that captures the UART output of the ESP8266 and forwards it to its own serial connection with the host computer. Additionally, this board may reset the ESP8266 on command by pulling its `RST` pin to `GND`. The controller board also provides power to the ESP8266.

To aid debugging of the whole setup, we multiplex the serial connection of the ESP32 to n network connections to provide an

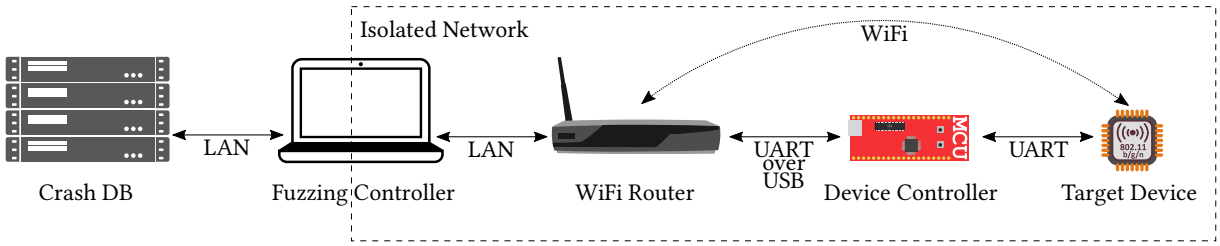

Figure 7: A schematic drawing of our fuzzing setup.

easy to use back-channel to the fuzzer while being able to monitor the serial output independently.

4.5 Practical Attacks on the ESP8266

For the following, we categorize our efforts in two categories: First, we sketch a method to achieve chosen payload execution, second, we discuss how we approach a permanent denial of service attack.

Identifying exploitable vulnerabilities in ESP8266 firmware might quickly lead to severe consequences. Code execution can be gained either by a traditional buffer-overflow based attack, whether on the stack, the heap or in a structure in a R/W mapped data section. Additionally, the ESP8266 provides an easy-to-use firmware mechanism. However, there is absolutely no protection against corrupting the ESP8266's firmware; even the download has to be done via unencrypted HTTP as HTTPS is not supported. Each firmware contains a 16 bit integer describing a version number. The new, downloaded firmware that is to be flashed onto the flash memory is then checked if its version number is greater than the one that is currently booted. If so, the download continues and the flash gets overwritten with whatever is presented to the update mechanism. Obviously, setting the version number to $2^{16} - 1$ disables the update mechanism until the device is retrieved and manually flashed via a UART download. We can also achieve the latter by directly overwriting the version field present in the SPI flash, bypassing the update mechanism.

Another method to permanently put an ESP8266 out of order is by physically destroy the flash chip by repeatedly read and write single flash cells in the image header region, thus hindering boot. Templeman and Kapadia introduced GANGRENE [21] that shows the feasibility of such an attack.

5 DISCUSSION

In the following we discuss important aspects and limitations of our method in general as well as specifics of our implementation.

The recoverability of an attack of the ESP8266s firmware greatly depends on the specific deployment of the device. In home IoT settings it is hardly imaginable that such a device would be user-serviceable given the current lax standards regarding (security) updates in consumer devices in the first place. In a more professional environment it is possible to re-flash a fixed version of the firmware manually to a limited amount of devices provided the OEM designed the specific application in a way a UART download flash is possible. To go to extreme lengths, devices could also be recovered by just swapping out the SPI flash chip with a known good one; albeit that this would involve some soldering. However, in a mesh network with a large number of deployed devices, recovering them most likely is not an option if, for example, the exact location of the devices is unknown due to aerial deployment.

While we managed to achieve our goal providing more and clearer output at the point of failure, our method obviously has a few limitations. First and foremost, while we provide a valuable tool for fuzzing, the Harzer Roller is not designed to be a security feature. It relies on the fact that the return of a function generates an exception that we can catch. An attacker would simply be able to overwrite the saved return pointer on the stack and gain control of the execution flow, completely bypassing the Harzer Roller.

It is unlikely although possible that automated software testing overwrites the saved return address in a way no exception gets triggered, which would corrupt the return stack on subsequent calls. This also holds true for accidentally writing the canary value to the saved return address when in fact an overwrite did happen. Such an access would not be detectable by our implementation of the return address checker.

Obviously there is also no generic function for dumping memory contents in the case of failure. Even with limited amounts of RAM the address space is only sparsely mapped, so the dumping function needs to be aware of the target chips memory map. In some cases, it may be even specific to the SDK used for development of the chip firmware, as information about e.g., the heap usage could be directly displayed in the dump. Still, this enables researchers to more easily obtain information about a chips internal state even when no debug interfaces like JTAG are available.

Recently, IoT devices with multiple cores appeared on the market. While our example implementation is tied to the single-core ESP8266 board, our method is applicable to systems employing parallel execution as well. For the call-path instrumentation nothing needs to be changed. The return-path instrumentation hook however needs to maintain a distinct return stack for each execution strain (usually each core).

As it is the case with the ESP8266, oftentimes storage size restrictions prevent instrumenting of the whole instrumented firmware due to the added overhead. While SPI flash memory is cheaply available online, not every board supports memory-mapping SPI flash chips of arbitrary size. A workaround to this may be to employ some kind of bank-switching scheme where the firmware switches between SPI chips at runtime. The feasibility of this, especially considering interrupt handling routines, remains to be researched in future work.

In some scenarios, e.g., smart door locks or industrial applications, it might be needed to connect actuators to the chip that are

not easily moved. Our setup allows to easily separate the fuzzer host and the fuzzed board, making fuzzing easier while the device is deployed.

6 RELATED WORK

In this section we cover related work in embedded (static) binary instrumentation and software testing.

Muench, et al. [13] pointed out that memory corruptions in embedded devices oftentimes result in different behavior than in desktop systems. The ESP8266, according to their paper, is a Type III device, with a single monolithic firmware model and no OS. Muench et al. observed no visible crashes (or reboots) while probing their Type III device. The Harzer Roller aims to improve this situation through e.g. our stack overflow detection as described in Section 3.

Thomas' LIEF project [22] is like the Harzer Roller a framework to instrument binary files. It handles substantially more formats than our work, but is mainly designed for systems that have an OS (Type I or Type II according to [13]). While it can rewrite parts of an ELF file, we found this capability rather unstable on uncommon architectures like xtensa. Because LIEF only handles files in a commonly known executable format, it cannot process flat firmware images for development boards out of the box. As is the case with the Harzer Roller, LIEF therefore is only applicable before linking as libraries in the ESP8266s SDK are provided in ELF format. In particular, we used the symbol parsing part of LIEF for the Harzer Roller.

Corteggiani, Camurati and Francillon [2] introduce Inception, a framework for symbolic execution of embedded systems software. While able to operate on binaries without available source code, it still requires a JTAG port present on the target device. Avoiding this was an explicit design goal of the Harzer Roller.

Song et al. [16] published PeriScope, a probing and fuzzing framework for the hardware-OS boundary. While interested in a related target, network stacks, their approach relies on the presence of a MMU to intercept memory access. Because on many embedded systems no memory management is available, we designed the Harzer Roller in a way that works without one. However, this limits the effectiveness of our method compared to PeriScope.

7 CONCLUSION

Motivated by the rapidly growing distribution of heavily interconnected embedded devices, we proposed the Harzer Roller, a method for embedded firmware testing. The Harzer Roller is especially useful for security testing of IoT deployments using closed-source firmware components, which can potentially introduce fatal vulnerabilities. With the Harzer Roller, we hook the control flow of firmware on calls and returns to and from subroutines. This allows us fine-grained insight to code execution flow and to detect stack overflows. While keeping the design of the Harzer Roller independent of embedded architectures, we implemented a prototype for the xtensa architecture. Our instrumentation method for libraries from any archive is ELF specific, but generally architecturally independent. We evaluated the usefulness at the example of the ESP8266 Wi-Fi chip, showcasing the tracing of execution flows and the detection of stack-based buffer overflows. The Harzer Roller can easily be ported to different chips and architectures, as long as exception handling is available. Depending on the architecture specifics, the overhead may be significantly less or more than what was observed on the ESP8266 platform. Furthermore, our setup for the ESP8266 can easily be adapted for various other embedded scenarios and be used for wireless protocol fuzzing such as bluetooth or Wi-Fi protocol implementations. We aim to open source our implementation shortly after the publication of the paper.

REFERENCES

[1] Winbond Electronics Corporation. 2015. *Winbond W25Q128FV Datasheet.* https://www.winbond.com/resource-files/w25q128fv%20rev.l%2008242015.pdf

[2] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[3] dgtlrift. 2016. Patch: QEMU simulation of ESP8266 prior to flashing. https://github.com/SuperHouse/esp-open-rtos/issues/230

[4] Matheus Eduardo. 2019. Proof of Concept of ESP32/8266 Wi-Fi vulnerabilities (CVE-2019-12586, CVE-2019-12587, CVE-2019-12588). https://github.com/Matheus-Garbelini/esp32_esp8266_attacks

[5] Joshua Pereyda et.al. [n. d.]. BooFuzz Source Code repository. https://github.com/jtpereyda/boofuzz

[6] Max Filippov et.al. 2015. esp8266 Memory Map. https://github.com/esp8266/esp8266-wiki/wiki/Memory-Map

[7] Max Filippov. 2015. esp8266 processor feature config. https://github.com/jcmvbkbc/crosstool-NG/blob/xtensa-1.22.x/overlays/xtensa_lx106.tar#L16085

[8] Tensilica Inc. 2019. Xtensa® Instruction Set Architecture (ISA) Reference Manual. https://0x04.net/~mwk/doc/xtensa.pdf

[9] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '14).* ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/2590296.2590301

[10] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *Computer* 50, 7 (2017), 80–84. https://doi.org/10.1109/MC.2017.201

[11] Espressif Systems CO. LTD. 2017. Espressif MIT License. https://github.com/espressif/ESP8266_NONOS_SDK/blob/90c641efe84066b47c4616ed367697a9f49f3ac5/License

[12] Espressif Systems CO. LTD. 2019. ESP8266 NONOS SDK Source Code. https://github.com/espressif/ESP8266_NONOS_SDK

[13] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA.* San Diego, UNITED STATES. http://www.eurecom.fr/publication/5417

[14] Kong K Rool, Macrox, Tolos, DGenerateKane, HyperHacker, Viper187, and Kenobi. 2004. The Secrets of Professional Gameshark(tm) Hacking. https://macrox.gshi.org/The%20Hacking%20Text.htm#gba_non_standard_master

[15] Paul Sokolovsky. 2019. ESP Open SDK. https://github.com/pfalcon/esp-open-sdk

[16] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2019).* San Diego, CA.

[17] Espressif systems CO. LTD. 2017. esp8266 ROM addresses: Exception Vectors. https://github.com/espressif/ESP8266_NONOS_SDK/blob/release/v3.0.0/ld/eagle.rom.addr.v6.ld#L45

[18] Espressif systems CO. LTD. 2017. Espressif Achieves 100 Million Target in IoT Chip Shipments. https://www.espressif.com/en/media_overview/news/espressif-achieves-100-million-target-iot-chip-shipments

[19] Espressif systems CO. LTD. 2019. ESP8266 Datasheet. https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[20] Espressif systems CO. LTD. 2019. ESP8266 Non-OS SDK API Reference. https://www.espressif.com/sites/default/files/documentation/2c-esp8266_non_os_sdk_api_reference_en.pdf

[21] Robert Templeman and Apu Kapadia. 2012. GANGRENE: Exploring the Mortality of Flash Memory. In *Presented as part of the 7th USENIX Workshop on Hot Topics in Security.* USENIX, Bellevue, WA. https://www.usenix.org/conference/hotsec12/workshop-program/presentation/Templeman

[22] Romain Thomas. 2017. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/.