Dinosaur Resurrection: PowerPC Binary Patching for Base Station Analysis

Uwe Müller, Eicke Hauck, Timm Welz, Jiska Classen, Matthias Hollick
Secure Mobile Networking Lab, TU Darmstadt
{umueller,ehauck,twelz,jclassen,mhollick}@seemoo.de

***Abstract*—Even though PowerPC mostly disappeared from the consumer device market, its architectural properties continue being popular for highly specialized systems. This particularly includes embedded systems with real-time requirements that are deeply integrated into critical infrastructures as well as aeronautics, transportation, control systems in power plants, etc. One example is Terrestrial Trunked Radio (TETRA), a digital radio system used in the public safety domain and deployed in more than 120 countries worldwide: base stations of at least one of the main vendors are based on PowerPC. Despite the criticality of the aforementioned systems, many follow a security by obscurity approach and there are no openly available analysis tools. While analyzing a TETRA base station, we design and develop a set of analysis tools centered around a PowerPC binary patcher. We further create various dynamic tooling on top, including a fast memory dumper, function tracer, flexible patching capabilities at runtime, and a fuzzer. We describe the genesis of these tools and detail the binary patcher, which is general in nature and not limited to our base station under test.**

I. Introduction

The TETRA specification dates back to 1995 and serves a similar purpose as Global System for Mobile Communications (GSM)—wireless voice and data transmission but for emergency services [5]. While both specifications stem from the same era, TETRA was designed with safety and security applications in mind. More than 120 countries over the world deployed it [31]. It is the predominating technology used by the police, fire departments, and ambulances across Europe. As a governmental project with many dependencies, the rollout took longer than expected. Thus, despite being rather old from a technological perspective, it is very new from an organizational standpoint—there are no plans to transition to a more recent technology. As of today, the older analog communication it was meant to replace still exists as fallback. While GSM, known as 2G cellular network, got multiple major revisions and is updated to 5G now, TETRA remains on 2G-like transmission quality. 3G and onwards have more modern extensions for emergency communication systems, but TETRA sufficiently serves the purpose of voice and text message transmission. TETRA uses completely independent infrastructure, which ensures that it is available during mobile carrier outages, which is likely the reason to stick to this technology.

Being an interesting target for security research, we acquired a TETRA base station for testing purposes, manufactured by one of the top five vendors. Surprisingly, this base station is running on a customized PowerPC platform. PowerPC is a RISC architecture originating from the same era as the TETRA specification. Despite mostly disappearing from the consumer device market, PowerPC has interesting properties for Industrial Control Systems (ICSs) and network equipment. Popular real-time operating systems, which form the base of such devices, continue having PowerPC support [14], [20].

The TETRA specification is publicly available except for its export-restricted encryption modes [5], [6]. The only open-source Software-Defined Radio (SDR) implementation lacks features like sending data to a base station [36]. Thus, TETRA base stations cannot be tested over-the-air with existing tools. Moreover, over-the-air testing is very limited when it comes to crash analysis. Instead, we create a PowerPC binary patcher and testing toolsuite with the following features:

- A generic PowerPC binary patcher capable of inserting custom hooks written in C into Executable and Linking Format (ELF) files.
- A thread-aware function call tracer compatible to the *Callgrind* format.
- Extension of the underlying *Enea* Operating System Embedded (OSE) commands to dynamically execute arbitrary firmware functions on the system.
- A fuzzer that observes the base station's state and panic messages.

In this paper, we focus on the PowerPC binary patcher, function call tracer, and dynamic function caller. Parts of this work are publicly available on GitHub [33]. Since TETRA is critical infrastructure, this paper does not cover any vulnerability research or reverse-engineering of security-critical functions.

Our base station setup and why we decided against emulation-based approaches are described in Sec. II. In Sec. III, we start with reverse-engineering the firmware that we aim to patch. The actual PowerPC binary patcher is designed and implemented in Sec. IV. Implementations of various dynamic debugging capabilities including a function tracer are provided in Sec. V. The fuzzing framework is described in Sec. VI. We conclude our work in Sec. VII.

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021, Virtual
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23009
www.ndss-symposium.org

II. Background

In this following, we document how we set up the base station to enable custom firmware injection. Then, we discuss advantages and disadvantages of emulation-based approaches compared to running on physical hardware.

A. Base Station Setup

The TETRA Base Station (BS) analyzed in this paper consists of a Base Radio (BR) and a Site Controller (SC). The site controller serves firmware and configuration files to the base radio via Trivial File Transfer Protocol (TFTP). To this end, they are connected via an Ethernet cable. Since 10BASE-T and 100BASE-TX only require two pairs of pins to operate and this data rate is sufficient, the remaining two pairs are used for a proprietary CP2 clock signal operating at 20 MHz. In our setup, we split this cable to connect the base radio to the proprietary site controller CP2 clock signal but control firmware and configuration files via a Raspberry Pi. Even though this setup requires physical access, it is still valuable for firmware analysis. The overall setup is depicted in Fig. 1.

Moreover, the base radio has an authenticated serial interface that can issue selected commands. These commands include an *AirTracer*, which shows all TETRA air interface traffic. Except from that, the debugging capabilities are limited.

For safety reasons, we configured the base station to an invalid frequency, attach a dummy load instead of an antenna, and place it into an Electromagnetic Field (EMF) shielded room.

B. Binary Analysis Options

The firmware sent via TFTP comes as ELF file. The ELF file is not stripped, meaning that it does not only contain basic information about the target, but even includes function names. However, the target architecture is PowerPC and the precise CPU variant is undocumented. The ELF file can be analyzed statically and dynamically. Especially dynamic analysis methods vary a lot, such as full-system emulation, emulation of single functions or simply running on the base radio with special hooks. We considered the following methods:

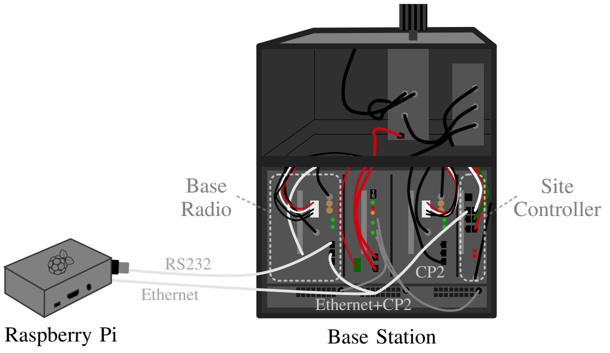


Fig. 1: TETRA base station setup with firmware injection.

1) Firmware Emulation: Quick Emulator (QEMU) can emulate PowerPC and attach a debugger to it to step through the code [26]. Loading the ELF file into QEMU and booting it fails at code related to hardware detection. This error can be bypassed by hooking the hardware detection function and returning a valid hardware model. Then, the firmware tries to configure and enable a timer that does not exist in QEMU for this machine. This timer is likely for the CPU scheduling routine and, thus, the operating system cannot run without it. It might be possible to fix these issues, however, they also show how hardware-dependent the firmware is.

Another option would be to only partially execute selected parts of the firmware for security analysis. *Unicorn* [32], which is based on QEMU, recently got experimental PowerPC support in a fork [30]. However, PowerPC support did not exist when we started this project. Moreover, while executing a packet handler and passing a single input to it is simple to achieve with *Unicorn*, more advanced utilization comes with similar issues as booting the whole firmware in QEMU due to its hardware dependencies. Even worse, *Unicorn* lacks peripheral support included in QEMU.

In general, this can be solved with a lot of engineering work to implement interrupts, timers, and even thread switches. For example, *Avatar* solves hardware dependencies [37], [22], and *Frankenstein* goes as far as emulating a Bluetooth firmware with a virtual modem and task switches [27]. Recently, not only single parsers of a *Mediatek* baseband were fuzzed [19], but even the *Shannon* baseband firmware was emulated including hardware components [15]. However, implementing this for a TETRA base station requires a lot of work, even with the progress in the field of PowerPC emulation. Thus, despite this would bring better introspection capabilities, we stick to the concept of physically running modified firmware.

2) Physically Running Modified Firmware: A different option is to inject modified code into the base radio. This eliminates challenges like hardware dependencies. However, this approach comes with other difficulties, such as creating a valid firmware image, being limited to the base radio's speed, non-determinism due to uncontrollable side-effects, as well as limited interaction with the target.

Nonetheless, we choose the second option. It allows us to modify packets before they are sent over-the-air, trace execution on the physical hardware for complex operations, and more. While there are some basic PowerPC binary patchers [18], [29], ours is able to patch the firmware using C and reconstruct a valid, bootable firmware image for the base station.

III. Static Firmware Analysis

As a first step, we perform a static analysis of the firmware binary running on the base radio.

A. Firmware Version and Operating System

Despite acquiring the TETRA base station in 2017, its firmware build dates indicate that it was last updated in 2007. Since it was a test device, this is reasonable. For the mere purpose of designing a PowerPC binary patcher this does not make any difference. It is unlikely that the vendor fully re-implemented the firmware since then. The TETRA specification only got minor updates over the recent years.

The bootloader, which is stored separately in a ROM on the base radio, is called *"Enea POLO Bootloader"*. This stands for Portable OSE Boot Loader [1, p. 8]. The version string *"061201"* indicates a build date in 2006. Without any modifications, the bootloader accepts unsigned firmware images over Ethernet via TFTP. Given the base station setup, where the base radio directly connects to the site controller, this still does not allow any attacks from an outside network. If the bootloader fails to load a firmware binary via TFTP, it launches a built-in shell. Otherwise, it continues executing the `_START` symbol in the firmware.

The firmware contains the string *"(C) Copyright 2003 Enea Embedded Technology"*, and the boot log hints to OSE as operating system in version 4.5.2. OSE 5 was announced in 2004 [2]. The string *"built by 'awr012' at Jan 19 2007 17:00:53"* suggests a more recent build date, meaning that the underlying OSE might have received a few more recent patches. OSE supports build options that compile both the operating system and applications into one binary [1]. Thus, the build date for the TETRA functionality is likely the same.

As of today, OSE is still maintained and sold by *Enea AB*, a Swedish company [3]. According to the operating system description, it is well-suited for TETRA, since it is optimized for high-performance communication systems with hard real-time characteristics. Its deployment areas include telecom networking systems and wireless devices.

Basic functionality extracted from the OSE manuals was documented publicly in 2005 [1]. Moreover, an old OSE kernel reference manual from 1998 was leaked [4]. The original manuals are not publicly available.

B. Firmware Format

The firmware is packed as a `gzip`-compressed ELF file. This is a common format that can be analyzed with tools like `readelf`.

The header shows that the base radio runs on a 32 bit PowerPC processor, which is a well-documented processor family [9]. Moreover, the string *"m8xxIOMemMap"* inside the firmware hints to the more specific *MPC8xx* family by *Motorola* [10], which has been acquired by *Freescale Semiconductor*. Dissecting the base radio reveals an *MPC8260ADS* System on Chip (SoC), which contains a modified version of the *MPC603e* big-endian PowerPC CPU [8]. We estimate that it has 48 MB RAM. When calling the command `board` in the bootloader, the following additional information is printed:

```
Motorola MPC8260ADS board
Bus clock 66MHz, core clock 264MHz, CPM clock 165MHz, BRG
     clock 83MHz
```

Ghidra supports this architecture [23], allowing static reverse-engineering.

The ELF file is non-stripped—it exports many symbols. In total, the firmware exports 7007 function and further 10 512 non-function names. Hence, even though the firmware is rather complex, it can be analyzed without source code and a fair reverse-engineering effort.

C. Interrupt Vectors

Based on the generic PowerPC documentation [9], we start by analyzing the interrupt handlers. These are located at offset `0x100` in the binary with the reset vector that calls `zzexception`. Most interrupt handlers are either not set, throw an exception via `SysAccessExcept`, or call an interrupt service routine using `GenIsr`.

D. Libraries and Function Prefixes

We can infer which public libraries are used along with proprietary libraries that start with common prefixes from the non-stripped ELF file. A list of the most interesting functions that we are able to assign a purpose to are listed in Tab. I.

The network stack contained in libraries like `ipcom` was developed by the Swedish company *Interpeak AB*, which was acquired by *Wind River Systems* in March 2006 [21]. The base radio's Command-Line Interface (CLI) has a `version` command that prints all information about the network stack:

```
(#) IPCOM   $Name: ipcom-ose-r5_12_3
(#) IPLITE  $Name:
(#) IPTCP   $Name: iptcp-any-r1_6_2
(#) IPTFTPS $Name: ipappl-any-r1_12_2
```

Understanding the network stack is vital to exchange data with the base radio over Ethernet instead of the existing serial interface. In the following, we will create new network services and fuzz existing ones.

The CLI that is accessible via the serial interface is contained in the `shell` and `cmd` functions. We will use these later on to add custom commands, such as calling arbitrary firmware functions with chosen parameters out of context during runtime.

TABLE I: Function groups in the ELF and their purpose.

#	**Prefix**	**Purpose**
40	—	`zlib`, symbol names match library [11].
140	—	`libc`, symbol names match library [12].
70	`efs_`	High-level file system functionality.
41	`clfs_`	Low-level file system functionality.
429	`ipcom_`	IP communication.
147	`iplite_`	IP communication.
75	`iptcp_`	Transmission Control Protocol (TCP).
17	`iptftp_`	Trivial File Transfer Protocol (TFTP).
11	`tftp_`	Trivial File Transfer Protocol (TFTP).
48	`snmp_`	Probably Net-SNMP library.
38	`scomm_`	Site communication with UDP socket abstraction.
11	`pthread_`	OSE POSIX-compliant thread wrapper.
26	`ose_`	Generic OSE functions.
116	`afm_`	OSE Atomic File Manager (AFM).
18	`fam_`	OSE Flash Access Manager (FAM).
18	`shell_`	OSE Command Line Shell.
79	`cmd_`	Shell commands like `ls` or `cat`.
25	`rtc_`	OSE Real Time Clock (RTC).
85	`pmd_`	OSE Post Mortem Dump (PMD).
133	`bs_`	Probably basic system process and timer management.
171	`core_`	Core functionality.
35	`sysconf_`	Configuration access.
177	`zz`	Functions that force the syscall interface.
21	`xx`	Kernel-side implementation of functions like `xxmutex_lock`.

IV. PowerPC Binary Patcher

Enea OSE comes with a debugging toolsuite called *Illuminator* [2]. However, *Illuminator* is closed-source and only available for *Enea*'s customers. From the few things that are publicly documented, it seems to integrate into the OSE development platform and likely requires source code of the target [1]. Since we neither have access to the base radio source code nor to *Enea*-internal tools, we implement a binary patcher that can extend the firmware ELF file. This binary patcher is the foundation to building debug tools like dumping memory during runtime or tracing function execution.

A. Hooking PowerPC Functions

The binary patcher aims to allow functions to be extended in the beginning (`PRECALL`), end (`POSTCALL`), or to be overwritten (`REPLACE`). Adding instructions prior to or after a function is very useful for analysis and does not require to understand the code of the target function. Many binary analysis frameworks support these types of hooks, such as FЯIDA [24].

NexMon, a binary patching framework for ARM, inserts a new function at a free memory location and then overwrites selected branch instructions [28]. This is *NexMon*'s default hooking approach and significantly reduces the instruction overhead spent in the hook itself. However, this means that the patcher needs to know all references to a function, i.e., replace each `bl memcpy` instruction with `bl memcpy_patch`. Additionally, functions may also be referenced by function tables instead of branch instructions, which also need to be replaced. Since disassemblers tend to make a lot of mistakes during static analysis [25], there is a high chance to miss function references.

A more reliable solution is to modify the target function. A typical function prologue and epilogue in PowerPC assembler are shown in Listing 1. Interestingly, all functions in our target firmware start with the `stwu` and `mflr r0` instructions in any order. Both instructions are position-independent and can be moved to any other position in the address space without side effects. Thus, we can replace the first instruction of the original function and transparently jump to a `PRECALL` or `REPLACE` hook. A `PRECALL` hook needs to take care of executing the replaced first instruction of the original function before returning to the second instruction of the original function.

```
stwu r1, -0x10(r1) ; r1 is the stack pointer, make room
     ; Replace this with branch to hook
mflr r0            ; Move contents of link register to r0
stw  r0, 0x10(r1)  ; Push r0 onto stack
; Function code
lwz  r0, 0x10(r1)  ; Load r0 from stack
mtlr r0            ; Move contents of r0 to link register
addi r1, r1, 0x10  ; Restore old stack pointer
blr                ; Branch link return
```

Listing 1: Common function entry and exit.

B. Trampoline Code

In theory, replacing the first instruction would already be sufficient for patching functions assuming that the original code and the patch are using the same Application Binary Interface (ABI). However, using this simple approach we could not carry any additional metadata in hooks, such as a function's runtime or the current thread. Additionally, this triggered a `gcc` cross-compiler bug in our first implementation attempt. Thus, we introduce a special trampoline code, which is individual for each hooked function. It saves the context of the call to the stack, adds metadata like the original function name and address, which in turn allows to return to the original code path.

Another reason for the trampoline code are the PowerPC calling conventions. In contrast to architectures like x86, which push the return address to the stack when calling a function, PowerPC does not have a native stack. Instead, the stack is a software construct, and PowerPC requires special registers to store information like the return address. Access to the link register is only possible with the `mflr` (move from link register) and `mtlr` (move to link register) instructions, as they are used in Listing 1. Yet, these instructions are not mandatory—a function that does not call subsequent functions does not save the link register.

Moreover, the first eight function parameters and return values are passed as registers. Thus, if a function signature that defines all these parameters is unknown, the purpose of each register cannot always be determined.

The trampoline code saves the context including all up to eight possible function arguments, link register, function name and call stub address into a Call Information Frame (CIF). We store the CIF pointer to the register `r14`, which is non-volatile by convention, meaning that follow-up functions must preserve its contents [17]. The CIF allows calling the original function in exactly the same context as it was called originally, which is required by `PRECALL` and `POSTCALL` hooks.

C. Trampoline Performance

A patched function contains additional trampoline code consisting of three components to keep it flexible.

1) *Individual Code per Hook*
Hook-specific code prepares the trampoline CIF for the hook, such as the call stub address or the function name as string, which requires 17 instructions.

2) *Saving the Context*
Moreover, there is generic code that can be reused for all hooks to save volatile registers in the trampoline call information frame. This requires 15 instructions.

3) *Restoring the Context*
Finally, to restore volatile registers from the trampoline CIF, there is additional generic code, which requires 12 instructions.

In total, each trampoline code introduces an overhead of 44 instructions, in addition to the implementation of the actual functionality that is added using the hook. For example, a void function, if not optimized, could be represented with a single `blr` instruction, while more complex functions require an arbitrary amount of instructions.

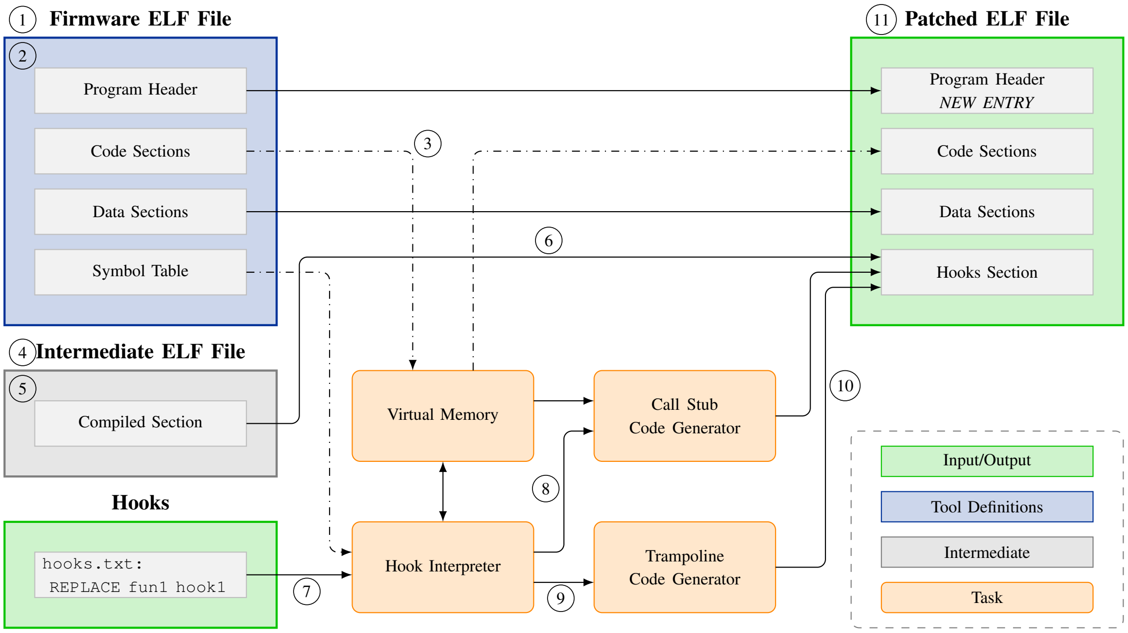


Fig. 2: Applying custom hooks to the original firmware ELF.

D. Applying Patches to the Firmware ELF

The base radio has two unused memory regions: One 252 kB region in the beginning directly after the interrupt vectors, and another one of 15.6 MB after the ELF `.bss` section. While the second region would be optimal in terms of space, it cannot be reached using a single instruction jump from the two program code sections. A PowerPC branch instruction can be absolute or relative. The branch instruction itself is encoded into 6 bit, leaving 24 bit for the absolute or relative address [16]. The remaining 2 bit of the 32 bit instruction encode if the branch is relative or absolute and if it includes the link register or not. However, the empty 15.6 MB section starts at the address `0x2098d68`, which requires a 26 bit representation. This cannot be represented with an absolute branch, and a relative branch from the highest code address at `0x27f7db` would still require 25 bit. Thus, the only solution to address this space would be using function pointers by loading the address into a register, which adds further overhead to the hooks. Instead, we leave some distance to the interrupt vector definitions, which is required for stability, and end up with the hook region `0x2500–0x3ffff`, which is 246.75 kB. While this has been sufficient for our experiments, the binary patcher could be extended to use the second memory region when accepting the additional overhead of a function pointer table.

We load our code into a new section at offset `0x2500` into the ELF file. This extends the program header table and moves the remaining parts of the ELF file. Thus, we use `libelf` [13] to add new program headers and keep track of file offsets in section headers. We still need to fix the program header size manually with our toolchain afterward.

The overall patching process is depicted in Fig. 2. More precisely, the steps to patch the ELF are as follows:

1) *Load Original ELF*
This is achieved using `libelf`.

2) *Read ELF Headers and Symbol Table*
These values are stored into internal variables.

3) *Map Memory*
The ELF data is mapped into a virtual memory according to the section header, such that it can be accessed by the virtual address.

4) *Load Compiled Intermediate ELF*
Using `libelf`, the intermediate ELF is loaded.

5) *Read Intermediate ELF Headers and Symbol Table*
We use the same approach as for the original ELF.

6) *Copy Compiled Hooks*
The `.compiled` section of the intermediate ELF is copied into a local buffer, which contains all code and data of the cross-compilation. The same local buffer is used to add the trampoline code, so this buffer must be 32 bit-aligned by adding zeros if necessary.

7) *Load Hook Configuration*
Load and read the hook configuration file `hooks.txt` including function name wildcards.

8) *Generate Call Stub*
If the original function starts with a `stwu` or `mflr` instruction, as supported by our binary patcher, this instruction is used to generate a call stub. Otherwise, the hooking process aborts with an error message. The generated code is added to the code buffer, which also holds the code of the intermediate ELF.

9) *Generate Trampoline Code*
Then, the hook-specific trampoline code is generated according to the configuration file, and added to the same code buffer.

10) *Hooking*
Using the mapped memory, the code at the virtual address of the original function entry is overwritten with a branch instruction to the virtual address of the newly generated trampoline code.

11) *Add Changes to Original ELF*
The new and modified data including headers is written into the new ELF.

The generated ELF can now be loaded into the site controller or the Raspberry Pi's TFTP server, such that the base radio bootloader can launch the customized firmware.

V. Dynamic Firmware Analysis Tools

Now that we are able to modify the original firmware, we can continue to add debug and analysis capabilities. Most importantly, we built tools for fast data exchange other than the serial RS232 interface, a function tracer, and a dynamic patcher.

A. Memory Dumps and Data Exchange

The site controller has a serial RS232 and an Ethernet interface. While the CLI is based on the serial interface, the Ethernet interface is faster and better suited to exchange larger amounts of data.

We add an Ethernet-based UDP service using the `scomm` library. This library uses the `br_service_table` to register services that can send and receive UDP packets on a service-defined port using the structure shown in Listing 2. Once we extend this service table, our custom UDP service is active.

Each `scomm_service` defines three callbacks. The `socket_cb` is only a wrapper, which can change the parameter order before calling the `scomm_recv_callback`. Every time the `scomm_recv_callback` receives a packet, it calls the `rx_cb` and passes an `scomm_packet` containing the data. The `tx_cb` is called by `scomm_packet_transmit` after packet transmission.

Using a custom UDP service, we add various memory observation commands to the OSE CLI. These are triggered via the slow serial interface, but the memory contents are exchanged using the Ethernet connection. The CLI can dump specific memory regions, the whole memory, watch a memory region, and search the memory for strings and hexadecimal data.

```
struct scomm_service {
  uint32_t port;
  uint32_t unknown1;
  uint32_t unknown2;
  socket_rx_cb rx_cb;
  socket_rx_cb tx_cb;
  socket_sock_cb socket_cb;
  uint32_t status;
};
```

Listing 2: `scomm_service` struct defining a UDP service.

B. Function Call Tracer

Static code analysis can be challenging: Which functions are actively executed by the firmware? What is contained in the function parameters? Even simple information like function references is not always correctly determined by disassemblers.

Enea OSE is a multithreaded operating system. For individual observation of each process, the process identifier and every function enter and leave can be logged resulting in a call tree. If two functions are entered subsequently, this adds a new level to the call tree. While the call tree calculation can take place outside of the patched firmware binary to optimize performance, the logging itself needs to be added to the binary. In addition to function enter and leave interactions, we log the CPU cycle counter to estimate the time spent in each function.

The code that implements this functionality is shown in Listing 3. The process identifier is only available after OSE is booted and undefined otherwise. The `calltrace` functionality can be added as a hook to every function. The hook definition allows wildcards, meaning that a hook to `dlai_*` would log every function call in the air interface downlink. The trace is transmitted using a Telnet service over Ethernet.

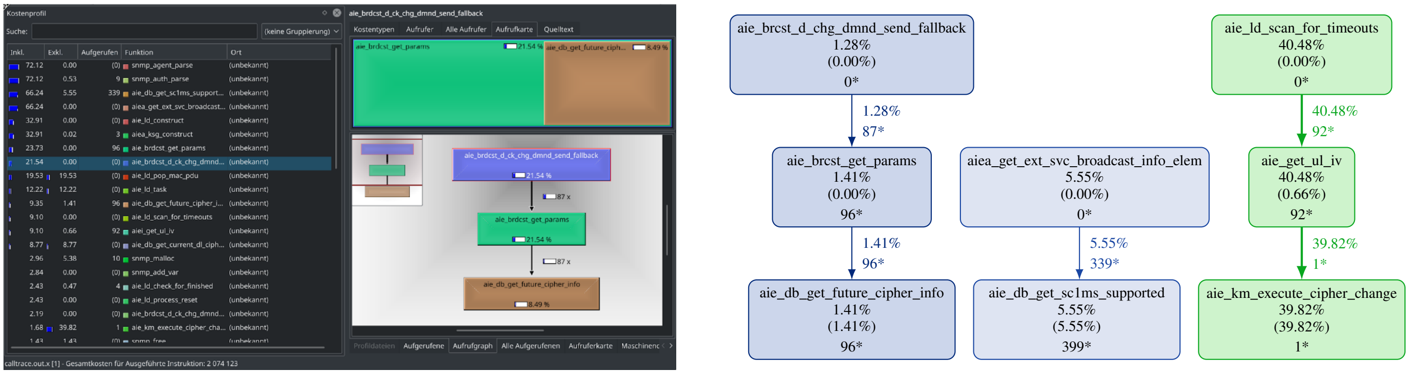
We test the function tracer on a large set of wildcards covering the most important aspects of the firmware, as shown in Tab. II. In some function groups, the trace hook leads to a firmware crash. This happens despite the generated code is valid. The station's watchdog is set to larger intervals than the additional hook instructions take to execute. Thus, we assume that this is due to some real-time requirements in functions

TABLE II: Function groups and associated hook crashes.

#	Prefix	Crash
95	`aie_`	—
45	`aiea_`	—
50	`aiei_`	—
12	`mac_pdu_`	—
289	`tx_`	Crash after a few seconds.
49	`rx_`	Crash immediately after boot.
74	`sm_`	—
174	`dlai_`	—
28	`ulai_`	—
42	`cca_`	`OSE_EPROCESS_ENDED`
40	`ccai_`	—
10	`lapd_`	—

```
uint32_t calltrace() {
  uint32_t pid = 0;
  if (ose_ready) {
    pid = current_process();
  }
  calltrace_log_enter(pid);
  // Get cycle count from CPU registers for time measurement
  uint64_t begin =  cpu_cycle_count();
  // Call original function w/o knowing anything about it
  uint32_t ret    = orig_call();
  // Get cycle count again for duration
  uint64_t end    = cpu_cycle_count();
  calltrace_log_leave(end - begin);
  return ret;
}
```

Listing 3: Gather calltrace information.



Inkl.	Exkl.	Aufgerufen	Funktion	Ort
72.12	0.00	(0)	snmp_agent_parse	(unbekannt)
72.12	0.53	9	snmp_auth_parse	(unbekannt)
66.24	5.55	339	aie_db_get_sc1ms_support...	(unbekannt)
66.24	0.00	(0)	aiea_get_ext_svc_broadcast...	(unbekannt)
32.91	0.00	(0)	aie_ld_construct	(unbekannt)
32.91	0.02	3	aiea_ksg_construct	(unbekannt)
23.73	0.00	96	aie_brdcst_get_params	(unbekannt)
21.54	0.00	(0)	aie_brdcst_d_ck_chg_dmnd...	(unbekannt)
19.53	19.53	(0)	aie_ld_pop_mac_pdu	(unbekannt)
12.22	12.22	(0)	aie_ld_task	(unbekannt)
9.35	1.41	96	aie_db_get_future_cipher_i...	(unbekannt)
9.10	0.00	(0)	aie_ld_scan_for_timeouts	(unbekannt)
9.10	0.66	92	aiei_get_ul_iv	(unbekannt)
8.77	8.77	(0)	aie_db_get_current_dl_ciph...	(unbekannt)
2.96	5.38	10	snmp_malloc	(unbekannt)
2.84	0.00	(0)	snmp_add_var	(unbekannt)
2.43	0.47	4	aie_ld_check_for_finished	(unbekannt)
2.43	0.00	(0)	aie_ld_process_reset	(unbekannt)
2.19	0.00	(0)	aie_brdcst_d_ck_chg_dmnd...	(unbekannt)
1.68	39.82	1	aie_km_execute_cipher_cha...	(unbekannt)
1.43	1.43	(0)	snmp_free	(unbekannt)

calltrace.out.x [1] - Gesamtkosten für Ausgeführte Instruktion: 2 074 123

(a) *KCachegrind* interpretation.

(b) `gprof2dot` interpretation.

Fig. 3: Function trace converted to *Callgrind* for further interpretation.

involved into sending and receiving packets, as the prefixes `tx_` and `rx_` indicate. This assumption is affirmed by the fact that OSE interrupt processes, which are close to hardware, have special restrictions: on PowerPC, they are not allowed to execute floating point or vector operations and cannot access the heap [1, p. 14]. While the basic hook mechanism does not contain such operations, the `calltrace` function does. As a solution, interrupt processes can be excluded from function tracing.

The `calltrace` output is still not that intuitive for interpretation. Using a custom `tbrc2callgrind.py` script, we convert the log output into the *Callgrind* format [34]. Originally, *Callgrind* output is meant to show how much time is spent in which function to allow programmers profiling and improving performance. While we do not aim at improving the base radio's performance, the output still provides us with insights about which functions are called by a thread and how they depend on each other. The *Callgrind* format is supported by various tools that provide a human-readable representation. These include the *KCachegrind* [35] tool shown in Fig. 3a as well as the `gprof2dot` [7] script that converts the the graph into the a format for plotting as shown in Fig. 3b.

C. Patching During Runtime

In the setup described in Sec. II-A, the base radio loads a patched firmware ELF file from the site controller or the Raspberry Pi. This requires a reboot of the base radio, which takes a while. This is not ideal for debugging purposes. Thus, we extend the patching toolchain to support partial firmware modification during runtime.

The overall process relies on the previously described ELF patcher. When comparing two generated ELFs files, they only differ in the following information: the hooks section, which contains the actual patches, and single 4 B-instructions with jumps to these hooks. A patched ELF file always contains the same new entry in the program header, thus, the code and hook section do not move.

Assuming that the ELF of the firmware version running on the base radio is known, it is very simple to diff it against a version that modifies these patches. For debugging purposes, two versions can be disassembled with `objdump` and diffed, making the patches human-readable. For example, the difference added by the `PREHOOK printf calltrace` looks as follows:

```
405360c405360
< 1cd5c4: 94 21 ff 18  stwu r1,-232(r1) ; orig. instruction
---
> 1cd5c4: 48 02 dc 2e  ba   2dc2c       ; jump to trampoline
586913a586914,586934
>  2dc1c: 94 21 ff 18  stwu r1,-232(r1) ; new hook code
>  2dc20: 48 1c d5 ca  ba   1cd5c8 <printf+0x4>
> ; further instructions that are added...
```

We parse the diffing results block-wise, meaning that we separate it into continuous blocks of assembler code. A function for memory modification is already contained in the patched standard firmware and the code section is not protected against writes during runtime. Thus, we can apply these changes to the base station while it is running. The blocks are applied in ascending order, meaning that the patches are written before the hook jumps are applied. This prevents the original firmware from jumping into invalid patches.

Since this process requires cross-compilation, it is not executed on the base radio directly. Instead, the base radio is attached to a Raspberry Pi or a laptop that generates and pushes the firmware patches.

D. Dynamically Calling Functions with Arguments

The ELF-based patching approach requires firmware recompilation and pushing the resulting diff into the base radio's memory. Often, debugging does not require complex patches but simple functionality like calling a single function with parameters out of context. For example, a function without arguments and return value located at `addr` could be executed as follows. Note that the `void` usually is not written explicitly:

```
void (*addr)(void)
```

However, passing arguments can get complicated due to the PowerPC calling conventions. The CIF-based approach introduced in Sec. IV solves this and even saves additional metadata. When calling a single function that returns, we can let the cross-compiler take care of passing arguments. For simplicity, we only differentiate between functions without parameters and functions with the maximum number of arguments, since we can leave arguments empty, and define the following generic function call wrapper that is always added to the firmware:

```
void* execute_address(int argc, void* addr, void** args) {
  if (argc == 0) {
      return ((void* (*)(void))addr)();
  } else {
      return ((void* (*)(void*, ...))addr)(args[0],
          args[1], args[2], args[3], args[4],
          args[5], args[6], args[7]);
  }
}
```

This wrapper can be called directly from the base radio CLI without an additional compilation step. For example, we could now execute the `printf` function as follows:

```
printf("Hello %s %d", "World", 42);
```

The CLI can pass these three arguments to `printf`, which is located at address `0x1cd5c4`. Since we do not implement the same syntax as C, we still need to tell it the number of arguments and the argument types:

```
CSS: exaddr 0x1cd5c4 -p 3 %s "Hello %s %d" %s "World" %d 42
Hello World 42
```

The return type is interpreted to print it to the CLI. In principle, a function returning a `void` pointer could point to anything. We implement three options to handle the results depending on the expected result. If we expect the result to be a scalar, we print it directly to the console. Alternatively, we handle the result as address and print a given number of bytes from its destination. As a third special case we assume that the result is a null-terminated string and print it.

VI. Fuzzing with Hyphuzz

A method to evaluate firmware security is fuzzing. Randomized inputs are generated and sent to interesting parsers to locate bugs. In the case of a TETRA base radio, we can send such inputs to protocol handlers. We pick the UDP-based `scomm` stack for an initial evaluation of our fuzzer.

A. Hyphuzz Overview

Hyphuzz, the TETRA base radio fuzzer, runs on two devices. While the harness code runs on the base radio itself, the controller runs on a Raspberry Pi. This separation is necessary to be able to observe if the base radio crashed, and, in the worst case, even power cycle it using a USB-controlled power switch. However, for performance reasons, payloads are generated and injected on the base radio. A high-level overview showing this separation is depicted in Fig. 4. It consists of five components described in the following.

1) Worker: The worker is the *Hyphuzz* core component running on the base radio. It initializes a Pseudo-Random Number Generator (PRNG) using the `seed` parameter and based on this generates mutated packets injected into the target handler. We implement several test cases that randomize different parts of the packet, which can be specified using the `testCase` parameter. The `skip` parameter can be used to produce packets with the same `seed` again but discard N packets, which is useful for crash reproduction. Once started, *Hyphuzz* runs autonomously until it is stopped or the base radio reboots. It regularly reports its state to the controller using a UDP connection.

2) Command Server: The worker receives commands from the command server, which also runs on the base radio. The command server listens on UDP and waits for fuzzing parameters or the instruction to stop the fuzzer. Moreover, the command server can be used to determine if the base radio is still alive or crashed.

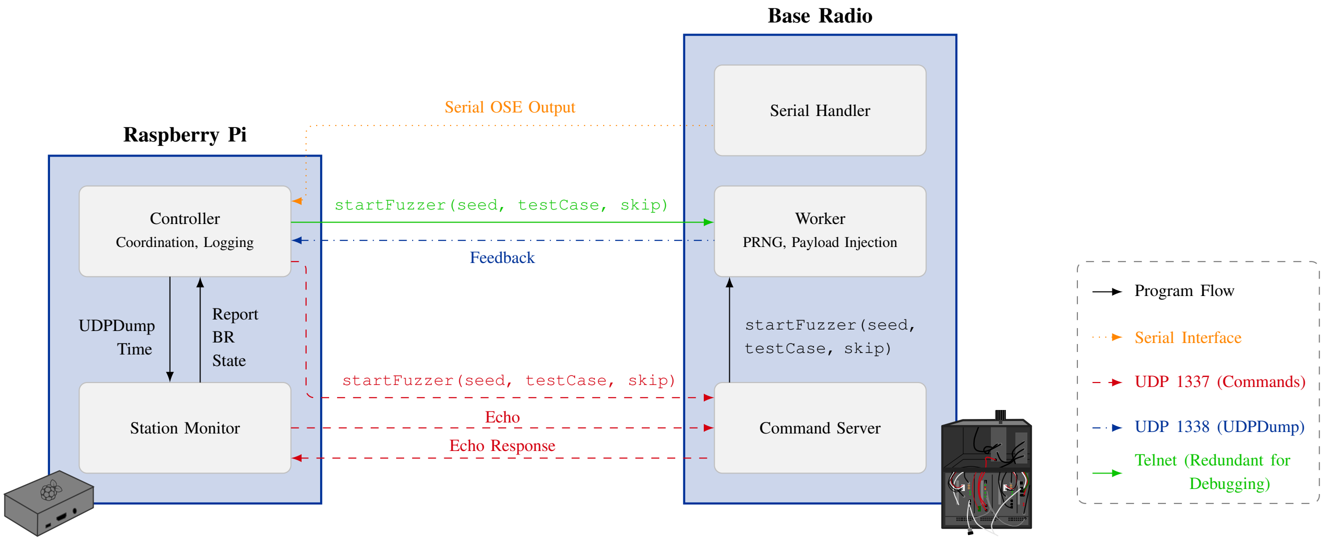


Fig. 4: Overview of the *Hyphuzz* components.

3) Controller: The controller runs on the Raspberry Pi and coordinates the overall fuzzing process. It sends fuzz cases to the worker and logs the worker's fuzzing feedback into a file. Once the base radio hangs up or reboots, the controller logs the latest statistics and serial debug output before starting the next fuzzing round.

4) Station Monitor: The station monitor oversees the base radio's current status. If the base radio stopped sending feedback for a specific amount of time, it has likely crashed or hang up. Typically, the base radio restarts itself automatically and becomes responsive again. However, sometimes, it just freezes. In this case a hard reset is automatically enforced via the power switch. Interestingly, we also have some cold boot side effects when the base radio restarts itself. Thus, we additionally enforce a hard reset even after automatic reboots and after each fuzzing round to enforce a clean state.

5) Serial Handler: The serial handler is not contained in the base radio's firmware ELF file. Instead, it seems to be a separate component integrated into the ROM similar to the bootloader. As such, we cannot modify it. However, we use it for feedback. Some crashes produce OSE error messages printed to the serial interface prior to a base radio reboot when the remaining system is already stuck.

B. Target Handler Selection and Harnessing

In the following, we select a suitable protocol parser for fuzzing and harness it to accept randomized inputs.

1) Security Considerations: The `scomm` stack can only be reached using a wired Ethernet connection.

First of all, an attacker would need physical access to the base station to attach cables—and with this capability, it is already possible to inject arbitrary firmware. Thus, research on this interface cannot uncover any critical findings on this sensitive target. The `scomm` stack is exclusively used for trusted backbone services. While this is the opposite of what a security researcher would aim at under normal conditions, keep in mind that the target is critical infrastructure and we do not want to uncover anything dangerous in the scope of this paper. Note that this is no technical limitation of *Hyphuzz*, it could be adjusted to also fuzz the TETRA air interface.

Second, the fuzzer's findings can be confirmed using an Ethernet connection. An `scomm` Proof of Concept (PoC) is as simple as sending Ethernet frames, which ensures reproducibility. In contrast, over-the-air PoCs could have severe impact if accidentally launched without EMF shielding.

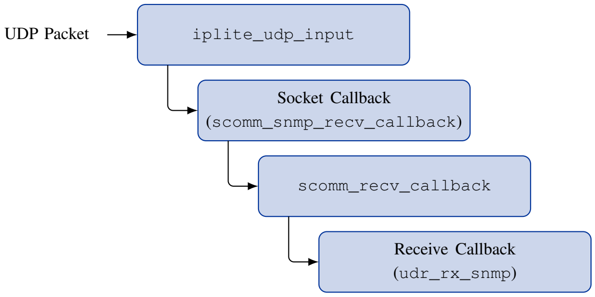


Fig. 5: UDP packet reception process.

2) Harnessing: In order to harness the `scomm` UDP protocol stack, we target the function `scomm_recv_callback` that handles incoming packets. All UDP traffic on the Ethernet interface gets processed by this function. Every time a packet is received, `scomm_recv_callback` looks up the corresponding service by using the port number defined in the packet. Each service has a callback function, which continues parsing the packet for this service. The full call graph for handling a UDP packet is shown in Fig. 5. Note that the functions calling `scomm_recv_callback` are simply wrappers that take care of calling it with the correct parameters. While this seems unnecessary from a reverse-engineering perspective, this might have looked slightly different before being optimized by a compiler. For example, the wrapper for the Simple Network Management Protocol (SNMP) callback looks as follows:

```
void scomm_snmp_recv_callback(uint32_t *rxSocketHandlePtr,
     struct NetPacketInfo *netPacketInfo, int
     packetStatus) {
        scomm_recv_callback(netPacketInfo, packetStatus,
               161, *rxSocketHandlePtr);
return;
}
```

The target function `scomm_recv_callback` takes four parameters. `netPacketInfo` is the most relevant for fuzzing. It contains a data structure with the full UDP packet and additional metadata. Our fuzzer randomly replaces bytes in this structure to generate new inputs. The `packetStatus` flag is always `1` for incoming packets. Successful packet processing is indicated by changing it to `2`. An unexpectedly closed socket leads to a status of `4`. While there are also other values for the status, these are not relevant for the purpose of our fuzzer. `161` is the port number for SNMP, passed as fixed value. `rxSocketHandlePtr` is slightly redundant and contains a pointer to an integer used as handle to reference the receive socket.

C. Crashes and Error Handlers

In the following, we describe the OSE kernel crash report format and provide insights into crash statistics during fuzzing.

1) OSE Kernel Error Messages: Depending on the crash caused by the fuzzer, the serial interface sometimes reports an OSE error message. These messages are generated by the function `krn_err_hndlr`. They were already useful during development of the fuzzer itself. Listing 4 shows an example for a stack overflow triggered due to our own programming error during fuzzer development. The general crash information contain the thread's name and the registers hint to the location of the issue.

2) Crash Statistics: Out of 453 crashes produced and logged during our fuzzing campaign, 206 (45 %) contain a kernel error message. These error messages can be used for initial triage and root cause analysis. Since an error message means that the kernel successfully detected a failure state and the base radio reboots afterward, such errors are potentially safe except from the base radio's availability. Note that silent crashes and hangs are more interesting but even harder to debug.

On this fuzzing campaign, the execution time until the fuzzer caused a crash was between 17 s and 31 min:14 s. 12 % of the test cases crashed within the first 27 s. Interestingly,

```
[ERROR HANDLER INVOKED]  fatal:YES  error:
                         OSE_ESUPERV_STACK_OVERFLOW(0x0102)
                         caller:<UNKNOWN>(0x00)
        [ERROR DETAILS]  user:NO  code:0x080000102
                         subcode:0x0aebd60
[PROCESS CONTROL BLOCK]  name:fuzzer_thread type:OS_BG_PROC
                         (64) status:<UNKNOWN>(0) priority:0
                [STACK]  top:0x0aec55f limit:0x0aebd60
         [CALLING CODE]  n/a:0
            [REGISTERS]  R0=3718B74E R1=00AEBD38 R2=002877CC
                         ...
               [ACTION]  Writing post mortem debuger info
               [ACTION]  Resetting BR
```

Listing 4: Stack overflow triggered during fuzzer development.

TABLE III: Crash types and their frequency.

#	Error Type	Caller
158	OSE_EILLEGAL_PROCESS_ID	OSE_SEND_W_S
33	OSE_ENOT_SIG_OWNER	OSE_SEND
8	OSE_ENOT_SIG_OWNER	OSE_SIGSIZE
4	OSE_EPROCESS_ENDED	<UNKNOWN>
3	OSE_EILLEGAL_SYSTEMCALL	OSE_WAIT_SEM

22 % of the crashes happen in the upper time range, between 30 min and 31 min:14 s. While we do not know the precise root cause of the crashes in the upper time range, they might be associated to a special base radio or OSE property that triggers after half an hour or memory buffer being exceeded after that time span. The remaining 66 % of the crashes are equally distributed over time.

3) Crash Types: In the following, we briefly describe the most common error messages. Tab. III summarizes these crashes and their frequency. The comparably outdated kernel reference manual still contains useful information about the error types [4], which allows further insights.

`OSE_EILLEGAL_PROCESS_ID`: If the kernel observes an illegal block or process identifier, it raises this error. The `subcode` is the corresponding invalid identifier. The caller, which causes this error, always happened to be the function `send_w_s`. Its purpose is to send a signal from one process to another. However, this function is also used within the fuzzer's UDP dumper. Thus, we modify the fuzzer to send the debug information via the serial interface. Since this crash persists, we can exclude that it is a false positive due to the fuzzer's UDP-based design.

`OSE_ENOT_SIG_OWNER`: This error is triggered when a process invokes a system call requesting a signal buffer that it does not own. The `subcode` is the address of the corresponding signal buffer. During our fuzzing campaign, we observe two different values for the signal buffer. One is owned by the process `DLA1`, and the other is owned by `iplited`.

`OSE_EPROCESS_ENDED`: When a prioritized or background process ended, this error occurs. The `subcode` contains the process identifier. In our case, the terminated process is `ose_shell`, which serves the CLI.

`OSE_EILLEGAL_SYSTEMCALL`: This error is provoked when an interrupt process executes a system call that is only allowed for non-interrupt processes. We observe this in the two interrupt processes named `VEC03` and `VEC62`. The illegal system call in both cases is `wait_sem`. As the name suggests, this system call waits until the specified semaphore is acquired—however, interrupt processes should terminate as quick as possible. We were not able to reproduce this specific type of issue by replaying packets, even though it occurred a few times during the fuzzing campaign. Thus, it seems to be very timing and state dependent.

D. Performance

The *Hyphuzz* controller spends most of the time waiting for the worker. Its main optimization is to timely detect hangs and reboots and respond with a hard reset.

The worker's overhead is quantified by additional measurement code reported to the controller, which calculates the corresponding average and writes it to the log file. We use CPU cycles for runtime comparison. We measure four components:

1) the generation of the call target's input parameters,
2) the process of sending feedback to the controller via UDP,
3) the actual time spent in the target function, and
4) the cleanup performed after this call.

We execute the worker five times with different fuzz cases that call the target function 20 000 times per case. The average times of this experiment are listed in Tab. IV. The fuzz case generation requires the largest overhead with 9.5 % of the time. Overall, the worker introduces 11.7 % overhead.

TABLE IV: Average overhead introduced by the individual parts of the worker compared to the actual target call.

Activity	CPU Cycles	Overhead
Target Call	117 207	—
Input Generation	11 084	9.5 %
Feedback	1318	1.1 %
Cleanup	1278	1.1 %
Total Overhead	13 680	11.7 %

VII. Conclusion

In this paper, we developed a powerful PowerPC binary patching and dynamic instrumentation framework. Using a TETRA base station, we demonstrated that our implementation works in practice. The patched binaries run stable on this complex system even during fuzzing, and the framework assisted us in internal follow-up security analysis.

Features like the ELF patcher and function tracer are very generic and can be useful for similar projects, such as other outdated but still in use critical infrastructure or retro computing projects. Moreover, the insights gained into OSE and its possible error states can be helpful for other security researchers that encounter this closed-source operating system.

Acknowledgment

We thank Jannik Jürgens and Sven Neubauer for the initial setup of the base station environment as well as Patrick Dworski for the analysis of existing functionality and protocols like the *AirTracer*. Moreover, we thank Dominik Maier and A. Birkl for proofreading this paper.

This work has been funded by the LOEWE initiative (Hessen State Ministry for Higher Education, Research and the Arts, Germany) within the emergenCITY centre.

Disclaimer

No dinosaurs were harmed and no new dinosaurs were bred during our experiments. 🦕

Availability

The PowerPC binary patcher and generic tools are publicly available on GitHub [33]. Due to its security and public safety critical nature, we will neither release the TETRA firmware extracted from the base station nor the fuzzer.

References

[1] Sebastian Aland, Markus Happe, Nico Loose, and Florian Schoppmann. Seminarausarbeitung OSE. http://homepages.uni-paderborn.de/fschopp/hauptstudium/docs/rtos.pdf, June 2005.

[2] Embedded Staff. Enea Embedded Technology announces new version of OSE RTOS. https://www.embedded.com/enea-embedded-technology-announces-new-version-of-ose-rtos/, April 2004.

[3] Enea OSE Sysems AB. Real-time Operating System Enea OSE. https://www.enea.com/products-services/operating-systems/enea-ose, January 2021.

[4] Enea OSE Systems AB. OSE Kernel Reference Manual, 420e/OSE93-1 R1.0.4. https://docplayer.net/55608737-Ose-kernel-reference-manual-kernel-enea-ose-systems-ab.html, 1998.

[5] ETSI. ETSI EN 300 392-1 V1.6.1 (2020-04): Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 1: General network design. https://www.etsi.org/deliver/etsi_en/300300_300399/30039201/01.06.01_60/en_30039201v010601p.pdf.

[6] ETSI. ETSI EN 300 392-7 V3.5.1 (2019-07): Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 7: Security. https://www.etsi.org/deliver/etsi_en/300300_300399/30039207/03.05.01_60/en_30039207v030501p.pdf.

[7] José Fonseca. `gprof2dot`. https://github.com/jrfonseca/gprof2dot, January 2021.

[8] Freescale Semiconductor Inc. MPC8260 PowerQUICC™II Family Reference Manual, Rev. 2. https://www.nxp.com/docs/en/reference-manual/MPC8260UM.pdf, December 2005.

[9] Freescale Semiconductor Inc. Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture, Rev. 3. https://www.nxp.com/files-static/product/doc/MPCFPE32B.pdf, September 2005.

[10] Freescale Semiconductor Inc. PowerQUICC™I and PowerQUICC II Families. https://www.nxp.com/docs/en/fact-sheet/PWRQUCROADMPFS.pdf, January 2021.

[11] Jean-loup Gailly and Mark Adler. `zlib` - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://www.zlib.net/, January 2021.

[12] GNU C Library Contributors. The GNU C Library (`glibc`). https://www.gnu.org/software/libc/, January 2021.

[13] gnutools. `elfutils`. https://sourceware.org/elfutils/, January 2021.

[14] Green Hills Software. INTEGRITY RTOS - The most reliable and secure operating system. https://www.ghs.com/products/rtos/integrity.html, March 2021.

[15] Grant Hernandez and Marius Muench. Emulating Samsung's Baseband for Security Testing. *BlackHat USA 2020*, August 2020.

[16] IBM Knowledge Center. PowerPC Branch Instruction. https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/assembler/idalangref_bbranchinst.html, January 2021.

[17] IBM Knowledge Center. PowerPC Register Usage and Conventions. https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/assembler/idalangref_reg_use_conv.html, January 2021.

[18] JasperRLZ, Jdaster64, PistonMiner, and Zephiles. Tools for Paper Mario: The Thousand Year Door. https://github.com/PistonMiner/ttyd-tools, 2021.

[19] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, July 2020.

[20] Microsoft. Overview of Azure RTOS ThreadX. https://docs.microsoft.com/en-us/azure/rtos/threadx/overview-threadx, March 2021.

[21] Jessica Miller. Wind River to Acquire Interpeak. https://www.windriver.com/news/press/pr.html?ID=3002, March 2006.

[22] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A Multi-target Orchestration Platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[23] National Security Agency. Ghidra. https://ghidra-sre.org/, 2021.

[24] Ole André V. Ravnås. FЯIDA JavaScript API: Instrumentation. https://frida.re/docs/javascript-api/#instrumentation, January 2021.

[25] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly but Were Afraid to Ask. In *2021 2021 IEEE Symposium on Security and Privacy (S&P)*, pages 194–212, Los Alamitos, CA, USA, May 2021. IEEE Computer Society.

[26] QEMU. Quick Emulator. https://www.qemu.org/, 2021.

[27] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[28] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. Nexmon: The C-based Firmware Patching Framework. https://nexmon.org, 2017.

[29] Shellphish. Patcherex. Shellphish's automated patching engine, originally created for the Cyber Grand Challenge. https://github.com/angr/patcherex/, 2021.

[30] simigo79. Unicorn Engine PowerPC Fork. https://github.com/simigo79/unicorn-ppc, 2020.

[31] TETRA Industry Group. TETRA Around the World. https://web.archive.org/web/20120313035919/http://www.tetrahealth.info/worldCountries.htm, 2011.

[32] Unicorn. The Ultimate CPU Emulator. http://www.unicorn-engine.org/, 2021.

[33] Timm Welz Uwe Müller, Eicke Hauck. PowerPC Binary Patcher. https://github.com/jrfonseca/gprof2dot, February 2021.

[34] Valgrind Developers. Valgrind Technical Documentation: Callgrind Format Specification. http://valgrind.org/docs/manual/cl-format.html, January 2021.

[35] Josef Weidendorfer. KCachegrind. https://kcachegrind.github.io, January 2021.

[36] Harald Welte. The Osmocom TETRA project. https://osmocom.org/projects/tetra/wiki/OsmocomTETRA, 2016.

[37] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. 2014.