RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Coverage-guided fuzzing of industrial controllers

Maximilian Wolodin

Master's Thesis – March 23rd, 2020

Chair of System Security

1st Supervisor: Prof. Dr Thorsten Holz

2nd Supervisor: Dr Ali Abbasi

hgi SYSSEC

Acknowledgements

This work was mainly inspired by Dr Ali Abbasi. Thank you for guiding me through the ups and downs of reverse engineering, asking the right questions to crack even the hardest nuts, and giving structure to my work.

Additionally, I appreciate the help of my colleague Norbert Schindling very much, as he saved me from bricking the controller he provided me with.

I would also like to recognise the support of my wife during the hundreds of hours that I was working on this thesis and research. Furthermore, I appreciate her indulgence about being woken up by various alarms in the middle of the night.

This work stands on the shoulders of giants. I am deeply indebted to the communities and individuals behind the multiple open-source projects and research on which my work relies.

Thank you for sharing your research and code!

Abstract

Critical infrastructure, such as power plants, is at risk due to cyber-attacks. The most critical assets of those plants are the industrial controllers that ensure the smooth-running operation of the processes. Despite their high significance, security research on industrial controllers has only just started, and thus far mainly the low hanging fruits have been harvested. A structured search for vulnerabilities in industrial controllers has not yet been conducted.

A promising way to find vulnerabilities in software running on industrial controllers is fuzzing. Fuzzing is a form of executing software in thousands or even millions of iterations while providing random or unexpected input to the functions. A more elaborate method is to use coverage-guided fuzzing, where feedback from the fuzzing target is used to mutate the input data more purposely.

The most widely used fuzzer of this type is American Fuzzy Lop (AFL). Unfortunately, AFL cannot be easily applied to embedded devices such as industrial controllers due to multiple challenges.

In this work, we introduce and evaluate embeddedAFL, an integration of AFL that enables us to find vulnerabilities in industrial controllers using its SoC hardware tracing capabilities. This software showed very good results while being applied to an ABB AC800F industrial controller as a proof of concept, even though the execution speed was limited to about 2.5 executions per second by the fuzzing target.

CENSORED

Contents

1 Introduction

The electrical and water utilities, the transport network, as well as information and communication systems, constitute the so-called *critical infrastructure*, which is essential to maintaining vital societal functions. Damage or destruction of the critical infrastructure by cyber-criminal activity may have negative consequences for the security of any nation and the well-being of its citizens [1].

The proper functioning of the critical infrastructure typically depends on the operation of complex processes that are managed by instrumenting and controlling thousands of parameters. As this task is far too fast and complex for humans to handle, *industrial controllers* supervise the physical processes.

Due to its importance, the critical infrastructure is in the crosshairs of terrorists and hostile nation-state actors. In particular, industrial controllers are in focus, as they are the most crucial components, and through them the physical process could be disturbed or destroyed. While attacks in the past have focused on multipurpose operating systems like Windows, attacks such as TRITON [2], CRASHOVERRIDE [3], and Stuxnet [4] have shown that attacks which threaten industrial controllers are already possible.

Vendors of industrial controllers and control systems have mainly relied on 'security by obscurity' [5] as this was enough in past decades. However, most operating systems and programmes running in an industrial controller are likely to have been programmed in C or C++ [6–8]. These languages are not memory safe [9] and are, therefore, prone to memory corruption bugs [10] for which exploit mitigations are not easy to design and implement [11].

This leads to the conclusion that there are ultimately numerous vulnerabilities in the applications on which our modern society is built. This leads to the following question:

Is there a way to find vulnerabilities in applications?

The answer is a simple yes. One way to find bugs in software is fuzzing. Fuzzing is a 'method for discovering faults in software by providing unexpected input and monitoring for exceptions' [12]. An even smarter approach is called coverage-guided fuzzing, which was introduced by Michal Zalewski from Google [13] with the tool American Fuzzy Lop (AFL). This approach takes feedback from the fuzzed software and mutates the input data based on that feedback. AFL has found thousands of bugs in all kind of applications. This fuzzer is widely used because of its carefully researched fuzzing strategies [13].

As good as this sounds, very difficult problems must be solved before any coverage-guided fuzzer can be applied to an existing industrial controller. The hardest challenge is running instrumentation code in an industrial controller. This

code should give AFL information about the paths that are taken during the execution of fuzzed software. Therefore, a feedback mechanism must be implanted into the industrial controller that bookkeeps the current control flow. This implant must, in any case, not alter the normal operation of the controller.

However, before we start racking our brains regarding the design of the instrumentation code, we first must find a way to load our manipulated firmware onto the industrial controller. As industrial controllers are not open source, this requires much more effort than one might think. The original firmware must be retrieved, its binary representation must be rebuilt, the code must be decompiled or disassembled, and the main functionality must be understood. That, however, is just the beginning of the actual work.

Embedded systems, of which industrial controllers are a subgroup, do not run on multipurpose hardware. They are most likely to be running on specially crafted hardware or even on special silicon. This means that emulation of the hardware is not straightforward. Therefore, the existing code must be analysed in a disassembler like IDA Pro, while the instrumentation code must be developed directly in binary code.

This means that no high-level language — no compiler, just the developer manual for the CPU and zeros and ones — can be used. Therefore, a significant amount of time and effort will be needed to extract information such as the type of the Instruction Set or peripherals internals.

As if matters were not complicated enough, various industrial controller vendors deploy firmware integrity protection or have custom firmware formatting. Therefore, any instrumentation effort for the firmware must also involve bypassing those protections or providing valid formatting for the new firmware.

If the instrumentation is working correctly and AFL receives code coverage information and can craft very good input data, it is now necessary to tell AFL when the fuzzing target shows anomalies, i.e. when it crashes. The last requirement is that the information exchange between AFL and the fuzzing target must be as fast as possible.

This work shows how embeddedAFL helps to tackle some of the problems mentioned above for embedded devices that provide hardware tracing functionality and how all of these challenges have been solved to provide a proof of concept by fuzzing a widely used proprietary industrial controller. An ABB AC800F controller is used as an example.

1.1 Objective

The goal of this thesis is to improve the security of industrial controllers. Therefore, a proof of concept is developed that shows that coverage-guided fuzzing can be applied to industrial controllers without access to any source code. Furthermore, a framework is developed and tested to help others apply tools like AFL to embedded systems. It is shown that AFL, in combination with the

framework and instrumentation code, can fuzz an industrial controller with coverage guidance.

1.2 Research Questions

This work must overcome several challenges. Several research questions are used to structure the investigation. The first research questions are related to the reverse engineering of the actual environment, while the later research questions are related to the full process of fuzzing an industrial controller.

RQ1 Is it possible to retrieve the firmware of an industrial controller and analyse it, to get an understanding of its main parts, its structure and its functionality?

In order to be able to even think about instrumenting the controller, it must first be possible to understand the firmware and its functions. This is the only way to find a starting point. As a foundation, the hardware must be examined to gather information about the components used and create a plan for how the instrumentation can be done.

RQ2 Is it possible to instrument the firmware of an industrial controller and run the instrumented firmware in the controller?

In the second step, the firmware needs to be instrumented to extend its functionality and control the hardware. This task must also show that the instrumented firmware can be loaded onto an industrial controller and the instrumentation does not interfere with the original code. On top of this, the speed of the instrumentation code must be reasonable.

RQ3 Is it possible to extract trace information from the controller and use this information as coverage guidance for AFL?

Coverage-guided fuzzing relies massively on feedback regarding which code paths are taken while sending fuzzing data to the target. This information must be stored in the controller while the fuzzing data is processed to create minimal disturbance. After the fuzzing data is processed, a way to extract this trace information must be found. Furthermore, it must be ensured that the trace information is valid and that the calculation of the bitmap bytes is correct.

RQ4 Is it possible to monitor the current state of an industrial controller and control the hardware at every moment?

To detect crashes and timeouts, the fuzzing target needs to be supervised at every moment, and the fuzzer needs to have control over the target in such a way as to reset it to a defined state and control special functionalities. This control is not inherent to the controller and must be added by instrumented code.

RQ5 Is it possible to run coverage-guided fuzzing against any industrial controller?

This work builds a proof of concept by applying coverage-guided fuzzing to an industrial controller. As this work shows, many industrial controllers are at risk. This final research question should investigate whether the work already done and described in this thesis could enable others to apply coverage-guided fuzzing to further industrial controllers or embedded systems to improve their security.

1.3 Contribution

This work contributes to the security research on industrial devices in two different ways. First, it is shown that security by obscurity is not a valid way to prevent an attacker from altering the firmware for an industrial controller and eventually executing malicious code on it. This work demonstrates that it is even possible to run instrumented code that has no effect on the original functionality of the device.

Second, even though impressive results have been shown for coverage-guided fuzzing in all kinds of software and operating systems, it has not yet been applied to embedded systems or, specifically, to industrial control systems.

Therefore, the framework *embeddedAFL* is built, which enables other researchers to apply coverage-guided fuzzing to any other embedded systems if certain requirements are met. In addition, the core fuzzing component has not been altered, so any future improvement of AFL or any other fuzzer can positively affect this work.

1.4 Organisation of this Thesis

This thesis is structured in such a way as to give the reader an idea of the work process that is needed to apply the results to any other industrial controller or embedded device. Therefore, after the introduction, which has been given already, background information about industrial controllers and coverage-guided fuzzing is presented in Chapter 2. Furthermore, the challenges of fuzzing an industrial controller are outlined.

In Chapter 3, the expected outcome of the reverse engineering is sketched out, and the results for ABB's AC800F controller are presented. This information is the foundation that will be used in all of the later chapters of this thesis. The reverse engineering section is also the part that consumed the most time in this work, as many complex processes had to be analysed and enriched with self-crafted code.

In Chapter 4, the design of embeddedAFL is defined, such that this tool can fill the gap between AFL and the industrial controller. The software is explicitly designed to be modular, so it can leverage any other work regarding the coverage-guided fuzzing of embedded devices.

After the design specification, the implementation is described in detail in Chapter 5. In particular, the specific interfaces of the software are described. The instrumentation code running inside the controller is also presented, and its functionality explained.

In Chapter 6, the performance of the overall system is examined with respect to the fuzzing speed and accuracy. The speed testing is done over the complete

system, as well as over the individual interfaces, to allow deeper analysis of blocking actions.

The related work is discussed in Chapter 7, while a conclusion is given in Chapter 8. The limitations of the approach chosen are outlined in Chapter 0.

2 Background

The following chapter provides the reader with background information. First, the typical field of application of industrial controllers is described, and they are differentiated from other embedded systems. Next, the advantages of coverage-guided fuzzing are explained, and the prerequisites are outlined. In the last section, the challenges of fuzzing industrial controllers are presented.

2.1 Industrial controllers

Critical infrastructure and the systems that are part of it are now an inevitable part of the modern world. Today it is hard to imagine a country like Germany without a water supply, electricity, pharmaceutical products, or high-tech materials. Most of these things are provided or manufactured with the help of industrial control systems.

The operators of those plants and their industrial control systems rely on their secure and safe operation. The lifecycle of a typical plant is several decades. The lifecycle of an industrial controller is about 20–30 years [14]. This means that a controller built around the turn of the millennium is most likely to be still in operation, even if the human–machine interface (HMI) hardware has been replaced several times in between.

In earlier times, those systems were insular systems that were not interconnected with other systems, but this has changed. Now the industrial control systems are much more connected to other systems than ever.

With this connectivity come security implications. When doing risk assessments for industrial control systems, there are several assumptions regarding the security of those devices. First, it is assumed that an attacker is not capable of attacking the controller or one of its communication protocols because no attack vectors are known. Second, the probability that an attacker can attack the controller is very low. And at last, the risk of a direct attack at the controller level cannot be quantified.

Unfortunately, the attacks mentioned in the introduction and others have shown that these assumptions are no longer true. Instead, it is true that security researchers and attackers have just not targeted industrial controllers very much yet. For example, only 47 CVEs are listed as of February 2020 for the products manufactured by the number one distributed control systems vendor [15] in the MITRE Vulnerability database [16]. Most of them are related to standard IT protocols like FTP and HTTP.

Industrial controllers are special-purpose computers that are designed to fulfil exactly one task. The task is to process physical input signals from processes and calculate how actuators must operate to fulfil a given task. This calculation must be done in a defined timeframe to satisfy real-time requirements. The hardware of industrial controllers is mostly tailored to exactly this task and ruggedised for use in demanding industrial environments.

If required, the industrial controller or components of it can be operated in a fully redundant mode.

The software either runs on a common real-time operating system like QNX RTOS [17] or VxWorks [18] or is proprietary, i.e. specially developed by the vendor of an industrial controller.

The controllers have several common interfaces to fieldbuses like PROFIBUS or PROFINET [19] or other devices that connect to external IOs. The connection to a Supervisory Control and Data Acquisition (SCADA) system is often done via proprietary protocols over TCP.

2.2 Coverage-guided fuzzing

One of the most effective ways to discover vulnerabilities is fuzzing [20]. Fuzzing is a form of executing software in thousands or even millions of iterations while providing random or unexpected input to the functions of a piece of software [21]. Based on that random input, a fuzzer can observe whether a programme crashes in response to a specific input. Some of these crashes may be the outcome of a security vulnerability. As the mutation of the inputs is done without any feedback by the fuzzed programme, this approach is classified as *blind mutational fuzzing*. Unfortunately, this approach has a low degree of efficiency in finding bugs despite its high usage of resources [22]. The problem is shown in the following figure:

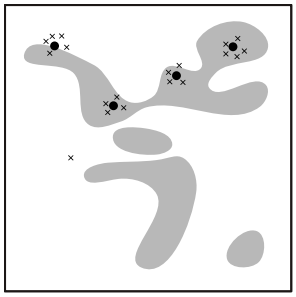


Figure 2-1 Blind fuzzer findings near seed values [23]

The fuzzer mainly explores the space state near the initial seed values. This implies that the fuzzer will only be successful if the seed values are good enough.

A more efficient approach is *coverage-guided fuzzing*. Coverage-guided fuzzing mutates the input data based on feedback from the fuzzed programme, particularly feedback about which code paths are taken and which are not. Input

data that discover new paths that have not been seen before are treated as interesting. This approach enables the fuzzer to mutate the input data much more efficiently and find good results, even if the seed values are chosen badly [23]. As one can see in the following figure, the coverage-guided fuzzer is also capable of exploring space states that are farther from the initial seed values.

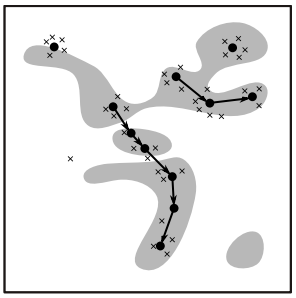


Figure 2-2 Coverage-guided fuzzers findings further from the seed [23]

As depicted above, this approach also has its limitations, as large gaps in the state space may not be overcome. Specifically, programmes with highly structured input formats typically have large gaps in the state space [23].

The most widely known coverage-guided fuzzer is American Fuzzy Lop (AFL) [13]. Due to its success, many other projects, such as LibFuzzer [24] and T-Fuzz [25], are now improving fuzzing strategies for special use cases. These tools have shown great results in finding hundreds of vulnerabilities in Microsoft Windows, Apple macOS, and GNU Linux [26]. Besides this great news, no coverage-guided fuzzer is known that can be natively applied to industrial controllers. In the next chapter, we will discuss why applying coverage-guided fuzzing to industrial controllers is a challenging task.

2.3 Challenges in Fuzzing Industrial Controllers

As described above, industrial controllers are special-purpose computers that are adapted to control large numbers of Inputs and Outputs (IOs) in real time. This implies that the hardware is specially crafted and cannot easily be emulated. Special chipsets or even special System-on-a-Chips (SoCs) are used and must be analysed and understood. Furthermore, they are designed to execute an explicit task to fulfil their limited purpose. The hardware is mainly limited to this purpose and is susceptible to any additional load. Most of them run proprietary software and communicate via undisclosed protocols.

There are some requirements for coverage-guided fuzzing. First, the software must be instrumented, and trace information must be collected. As vendors of those controllers are trying to protect their intellectual property, the source code is either available only in binary format or as a binary that is encrypted, scrambled, or otherwise not analysable without further effort. Thus, obtaining and analysing the existing firmware is still a difficult task. After retrieving the firmware, there must be a way to instrument the existing firmware. If there are

protective measures against the execution of altered firmware, these must also be overcome.

Moreover, the crash detection is not standardised, and the controller must eventually be instrumented to provide status and event information to an external fuzzer.

Due to their high degree of specialisation and the research effort invested in their development, the controllers are not cheap. This may be a reason why security researchers have not opted to work on a large scale on the security of industrial controllers, as the device can become 'bricked' in the context of trial and error. The minimal level of academic research in this area is also visible when searching for papers and articles that cover the actual hardware of physical process control.

There are no standards for how industrial controllers must be designed, or which interfaces they must provide. This means that there is no standard way to obtain trace information from the controller. For every use case, there must be a way to obtain trace information either from a hardware mechanism or in software.

In this thesis, we suggest using hardware tracing functionalities in industrial controllers to provide coverage feedback for fuzzing. Therefore, for our work, we will use a controller that is well-known and widely used in industrial applications, namely the ABB Freelance controller AC800F. This high-end industrial controller is primarily used in critical infrastructure such as oil and gas refineries, electrical substations, and water purification plants [27]. In the next section, we will discuss the initial process of reverse engineering this controller and describe how our embeddedAFL utilises its hardware tracing functionalities for coverage-guided fuzzing.

The further challenges of fuzzing embedded systems are described by Muench et al. [28].

3 Reverse engineering

Before any analysis can begin, it is inevitable to have access to the firmware or parts of it to be then able to instrument it. In the special case of an embedded system, it is additionally necessary to control the hardware and detect its status. To be able to apply coverage-guided fuzzing to an industrial controller, a way must also be found to collect coverage information and retrieve it from the controller. In this chapter, the overall process is outlined, and the results for the ABB AC800F controller are presented, so that this controller can be used as a proof of concept.

3.1 ABB AC 800F

As a proof of concept, ABB's distributed control system Freelance has been chosen because it is sold globally and is part of several critical infrastructure systems. It has been around for several decades but is still actively supported. The Freelance control system offers three different types of industrial controllers, which are used in thousands of different applications in industrial processes. The controllers are named AC700F, AC800F, and AC900F, the last two of which support controller redundancy. All these controllers can be combined in one project. Two different types of human–machine interfaces can be set up to work as a SCADA system. The system architecture is outlined in the figure below:

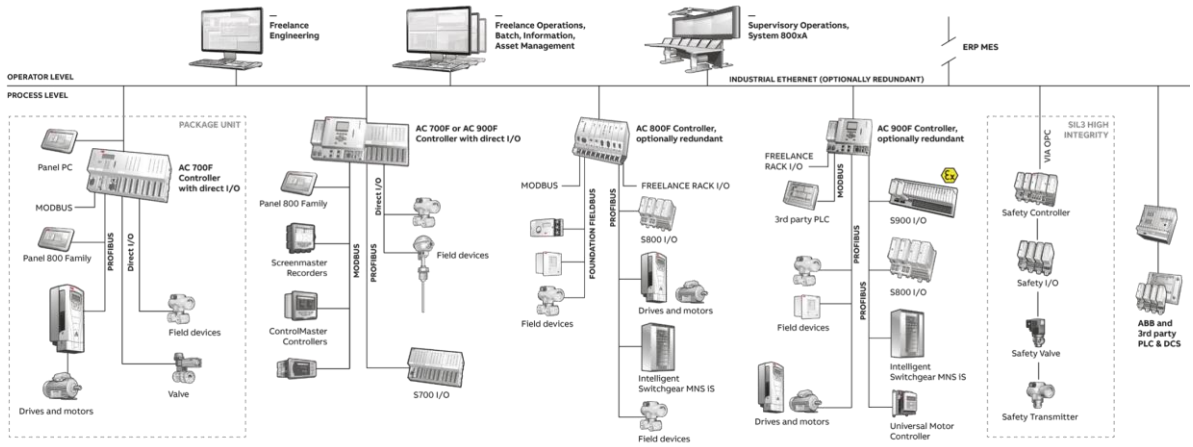


Figure 3-1 ABB Freelance system architecture [29]

The AC800F controller is chosen as a research object because a vulnerability would affect a wide variety of industries and critical infrastructure systems. The controller is already tested in ABB's Device Security Assurance Center (DSAC)

with the help of the Achilles Test Platform from Wurldtech and tools from Tenable, Nmap, and Spirent [30].

The controller consists of a CPU on the backplane and modules for power supply and interface cards. The chosen controller is equipped with the PM 802F Base Unit with 4 MB, battery-buffered RAM. A single controller can typically support around 1,000 IOs.

Figure 3-2 Base module PM802F and fully equipped controller [29,31]

The controller supports all major field bus protocols, like FOUNDATION Fieldbus, HART, and PROFIBUS [27]. Besides this, the controller in its most basic setup exposes a serial and an ethernet interface.

A web server with rudimentary status information is also available via TCP.

On the backplane, a 75 MHz RISC-based [32] Intel i960HT processor is mounted [33].

Figure 3-3 Intel i960HT processor on an AC800F backplane

With the available information, the capabilities of the processor can be examined, and the firmware can be analysed.

ABB uses the real-time operating system pSOS+ [34] for all three controller types.

3.2 Interfaces of the AC800F Controller

The AC800F controller has only two external interfaces by default, a serial interface and an ethernet interface. The serial interface is operating at 9600 Baud

with 8 data and one stop bit. The parity is none, and the flow control is working with XON/XOFF.

The ethernet interface with a speed of 10/100 Mbit provides several ports, as shown in the table below:

Port	Service	Purpose	Remarks
80/tcp	HTTP	Web Port	These ports will be used for maintenance purpose only. Ports can be disabled via the controller boot menu.
6660/tcp	Telnet	Menu Port	
7660/tcp	Telnet	Trace Port	
9991/tcp	Control	Communication between controller and CBF	Important ports for vertical communication
9992/udp	Nameserver	Controller nameserver protocol	
9995/udp	TimeSync	Time synchronisation	
9990/tcp	RedCom	Redundant controller communication	Ports for horizontal communication
9993/udp	RedCom	Redundant controller communication	
9996/udp	LatCom	Lateral communication between controllers	

Table 3-1 Communication Ports of the AC800F controller

The web and menu ports are accessible via ethernet. Unfortunately, the menu provided via ethernet shows limited information compared to the one delivered via serial port, especially in case of the boot process or in case of a fault. The trace port was observed, but no communication could be initiated, even though menu entries seem to enable some tracing. Therefore, neither telnet port is used via ethernet.

As part of this work, one interface should be tested for vulnerabilities with coverage-guided fuzzing. The vertical communication is most at risk, as no standard countermeasures, like disabling the ports or limiting the communication by firewalls, can be applied. As the fuzzer must get to know if the fuzzing packet is delivered correctly to the industrial controller, a protocol with a handshake is preferred. For this reason, port 9991/tcp is selected as a fuzzing target.

Communication between the Control Builder F (CBF) is carried out on this port via the Digimatik Message Specification (DMS) protocol, which is an extended Machine Message Specification (MMS) variant [29].

3.3 Retrieving the firmware

To analyse a firmware, either the source code is available or a binary representation of the firmware must be transformed by a disassembler or decompiler. Examples of these programmes are GHIDRA [35] and the Interactive Disassembler IDA Pro [36]. While a decompiler represents the binary in a readable form, the disassembler simply maps processor instruction codes into instruction mnemonics [36]. Unfortunately, no decompiler is known for the i960 opcode [36,37], so the retrieved firmware will be converted to the processor's mnemonics with IDA Pro.

Before translating the firmware binary, the targeted firmware and bootloader must be obtained in raw binary format. Different techniques exist to obtain the raw binary representation. In the best case, someone else has already done the task and made the firmware available on the internet, or the vendor already provides the raw format.

If this is not the case, the firmware must either be retrieved by wiretapping the loading process, reverse engineering the file format, or by dumping it from the device with the help of vulnerabilities or undocumented functionalities. As a last resort, the firmware-containing chips can be unsoldered from the hardware and read out externally. Ultimately, multiple techniques must be combined.

In the case of the AC800F controller, the firmware is shipped with each installation of the engineering tool of the controller. Two files are located in the installation path that are loaded into the controller: first, the bootloader in the file *msr6rom.hex* and second, the operating system in the file *MSR6OS.BIN*.

Both files have a special format that is revealed by reverse engineering the firmware loading programme.

3.3.1 Reverse engineering the Firmware loader

The firmware loader *RBD.exe* is located at *C:/Program Files (x86)/ABB/Freelance/exe* if the standard installation path is used.

The User Interface of the AC800F engineering tool starts the loader with the following arguments:

```
Argc=4
Argv [0] = path to exe
Argv [1] = "MSR6OS.BIN" = File to be loaded
Argv [2] = "192.168.1.1" = IP address of the controller
Argv [3] = "OS" = type of firmware part
```

Listing 3-1 Arguments of the firmware loader RBD.exe

Reverse engineering of this loader revealed the header structure of both firmware files. The header can easily be removed by skipping the first 288 bytes.

Byte	Meaning
0-4	***·
5-258	Boot file info
259-259	Board Type
260-264	Version
264- '/0'	MinEPROMversion
'/0'-288	End of Header

Table 3-2 Firmware file header

3.3.2 msr6rom.hex

After the file header, both files contain the firmware that is loaded into the controller. The content of the bootloader file *msr6rom.hex* can be obviously interpreted as Intel's Hexadecimal Object File Format Specification (Intel HEX) [38].

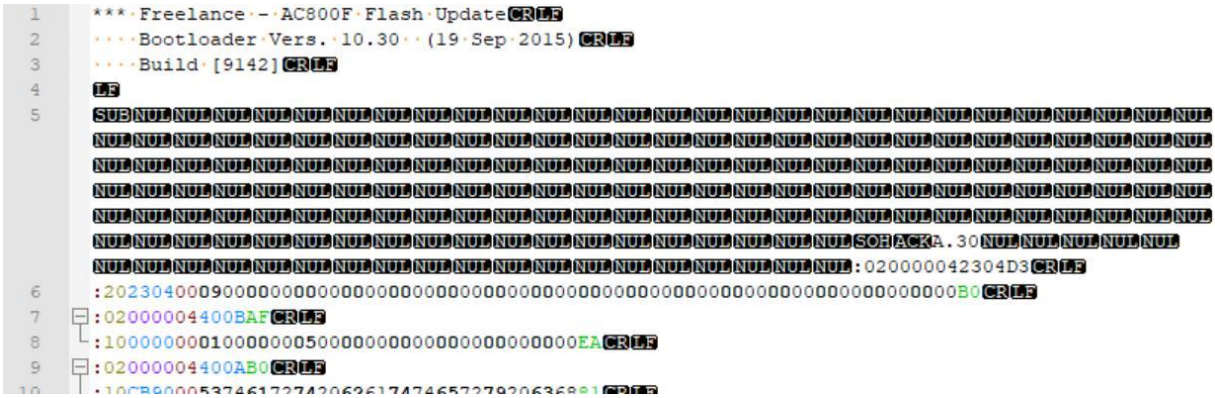


Figure 3-4 msr6rom.hex

After removal of the header of this file, the hex file can be converted with the tool *srecord 1.64* [39].

```
srecord-1.64-win32 msr6rom_stripped.hex -intel -output bootloader.bin -Binary
```

Listing 3-2 srecord command to convert msr6rom.hex

3.3.3 MSR6OS.BIN

The file *MSR6OS.BIN* was not directly recognisable as Intel HEX format, but it is assumed that this file also contains binary code in Intel HEX format.

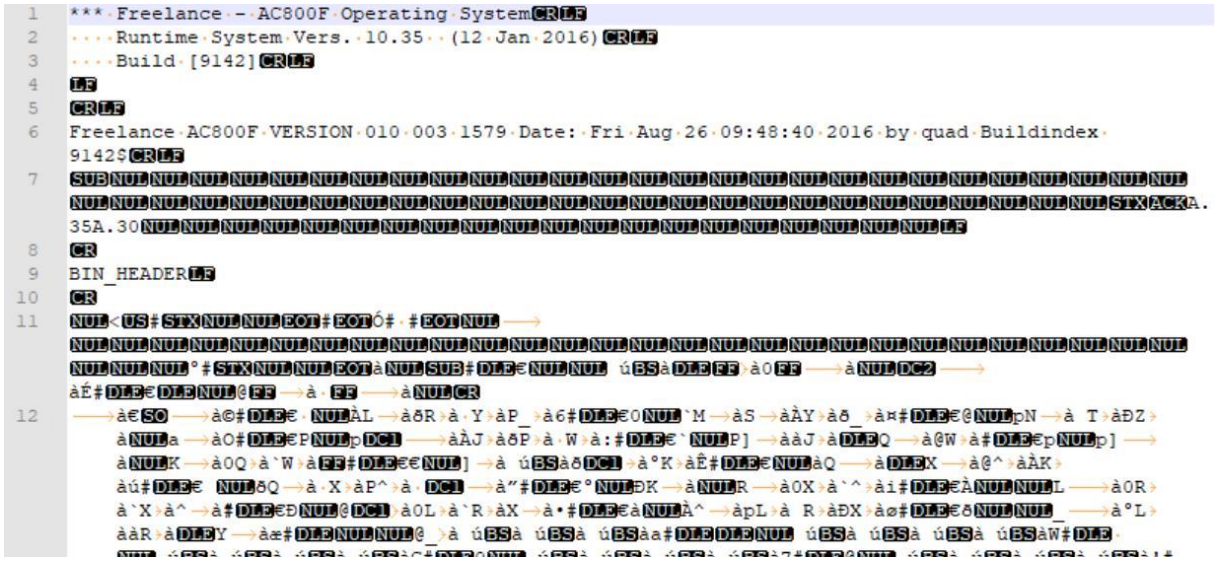


Figure 3-5 MSR6OS.BIN

To further investigate the format of the file, it has been loaded with a hex editor. To detect the start character within this file, a statistical analysis is conducted. The result is displayed in the following table:

"Character"	Relative frequency	Absolute frequency
0x0	16,49%	389625
0x10	5,67%	134015
0x23	4,82%	113938
0xE0	2,66%	62768

Table 3-3 Character frequency in MSR6OS.BIN

It turns out that the character 0x23 (#) is chosen as a start character. Furthermore, the content part of the file contains several blocks of binary code. These blocks are separated by *BIN˙HEADER* blocks that define the length of the following block. After reworking this structure, the file can also be converted to raw binary with *srecord*.

```
srecord-1.64-win32 MSR6OS_stripped.hex -intel -output os.bin -Binary
```

Listing 3-3 srecord command to convert MSR6OS.BIN

As *srecord* creates a file starting from 0x00000000, it is advisable to crop the files before loading them into IDA. In the case of the AC800F controller, the bootloader starts at 0x40070000, while the operating system starts at 0xE0000000.

3.3.4 Reviewing the firmware consistency

Fortunately, the AC800F controller has a telnet interface that can be used to dump some memory regions and a web interface that presents information that is relevant for reverse engineering, e.g. the bootloader start address.

This information is used to verify that the raw binaries are loaded correctly in IDA Pro to then be able to reverse engineer the firmware. In the verification process, it turns out that many references were not set correctly. It seems that major parts of the operating system are copied to another address space internally after the firmware is loaded into the controller.

A first try to dump the memory area via telnet fails because of memory protection controls. These controls are therefore bypassed by instrumenting the bootloader to switch off the memory protection directly after the first few instructions. This is done by storing 0x0 in the GMU Control register while running in supervisor mode.

After the memory protection is disabled, the memory region where the firmware is copied to can be dumped.

3.4 Finding Starting Points for Instrumentation

The instrumentation code in the industrial controller must provide two services. First, there must be a bookkeeping mechanism that receives all trace information and second, there must be a possibility of providing the bitmap to the external fuzzer.

3.4.1 Collect Trace Information

AFL uses trace tuples to store information about a current branch operation. Each tuple defines a transition from one instruction to another instruction by a branch. Only if the branch is taken is the counter for the tuple incremented. Normally, the current and previous branch operation is identified by a 'compile time random' that is instrumented to the original source code. As the original source code is not available in our case, another solution must be provided.

Instead of defining a compile time random, the instruction pointer of the current branching instruction is used like the compile time random.

In the first step, the instruction pointer of the current and previous instruction must be collected at every moment a branch is taken. As the manual instrumentation of every possible branch is not feasible, the help of another mechanism is required.

The i960 processor has a trace functionality that can generate a *fault* every time a branch is taken. Implementation details regarding this functionality are given later.

The fault record contains the previous instruction pointer, while the current instruction pointer can be acquired directly from the stack. So, this fault handling mechanism can be used to enable tracing in the controller.

3.4.2 Finding the Fault Handler

However, before the instrumentation of the fault handler can be coded, the original fault handler must be found. As the fault handler is known to print out all registers, it is expected that these registers must be loaded first. So, the raw binary is searched for the opcode of the load instruction (ld), an unknown register where the original register is loaded to, and the global registers. It is assumed that the registers are loaded in the correct order.

One tool to search binaries for these kinds of structures is YARA [40]. Consequently, a YARA rule is developed, which is depicted below:

```
rule pattern
{
   meta:
      description = "Find a binary pattern to finally find the fault handler"
   strings:
       $hex_string = { 00 30 88 90 ?? ?? ?? ?? 00 30 90 90 ?? ?? ?? ?? 00 30 98 90 ??
?? ?? ?? 00 30 A0 90 ?? ?? ?? ?? 00 30 A8 90 ?? ?? ?? ?? 00 30 B0 90 ?? ?? ?? ?? }

    condition:
       $hex_string
}
```

Listing 3-4 YARA rule for finding the fault handler

Luckily, YARA found three occurrences of this pattern. One finding is related to the original fault handler.

3.4.3 Finding the Web Server

As stated in Chapter 0, the AC800F controller contains a web server that provides status information about the controller. This web server could eventually also be used to exchange other information that could be beneficial for the coverage-guided fuzzing.

To find the web server, the response to a web request is analysed. Every successful HTTP GET request is answered with 'HTTP/1.0 200 OK'. This string can also be found in the string database of IDA Pro.

```
E0184F30 48 54 54 50+aHttp10200OkSer:.ascii "HTTP/1.0 200 OK\r\n"
E0184F30 2F 31 2E 30+                                  # DATA XREF: Write_Http_Response_E008CDF0+C↑o
```

Figure 3-6 HTTP status 200 string in IDA Pro

As the reference is correctly resolved, the web server code is found:

```
E008CDF0                Write_Http_Response_E008CDF0:
E008CDF0 50 60 08 8C    lda     0x50(sp), sp                # Load address
E008CDF4 10 16 20 5C    mov     g0, r4                      # Move word
E008CDF8 11 16 18 5C    mov     g1, r3                      # Move word
E008CDFC 00 30 80 8C+   lda     aHttp10200OkSer, g0         # "HTTP/1.0 200 OK\r\nServer: Freelance\r"...
E008CE04 25 00 88 8C    lda     unk_25, g1                  # Load address
E008CE08 68 FF FF 09    call    Send_HTTP_WebData_E008CD70  # Call
E008CE08
E008CE0C 40 E0 87 8C    lda     0x40(fp), g0                # Load address
E008CE10 00 30 88 8C+   lda     aContentLengthD, g1         # "Content-Length: %d\r\nContent-Type: app"...
E008CE18 03 16 90 5C    mov     r3, g2                      # Move word
E008CE1C 48 B8 FD 09    call    sub_E0068664                # Call
E008CE1C
E008CE20 40 E0 87 8C    lda     0x40(fp), g0                # Load address
E008CE24 30 B9 FD 09    call    unk_E0068754                # Call
E008CE24
E008CE28 10 16 88 5C    mov     g0, g1                      # Move word
E008CE2C 40 E0 87 8C    lda     0x40(fp), g0                # Load address
E008CE30 40 FF FF 09    call    Send_HTTP_WebData_E008CD70  # Call
E008CE30
E008CE34 04 16 80 5C    mov     r4, g0                      # Move word
E008CE38 03 16 88 5C    mov     r3, g1                      # Move word
E008CE3C 34 FF FF 09    call    Send_HTTP_WebData_E008CD70  # Call
E008CE3C
E008CE40 00 00 00 0A    ret                                 # Return
```

Figure 3-7 Part of AC800F's web server code

4 Design

The reverse engineering that is outlined in the previous chapter proved that it is feasible to get trace information out of an industrial controller. Furthermore, it is possible to load and run instrumentation code. The task is now to enable AFL to fuzz the controller and provide feedback from the software to AFL. This gap is not yet closed. Right now, it is not possible to apply coverage-guided fuzzing to industrial controllers. Thus, the tool *embeddedAFL* is developed as part of this work. This chapter provides insight into the design of *embeddedAFL*, the requirements that led to the final design, and the thoughts around the design decisions.

4.1 Assumptions

As pointed out in Chapter 0, there are several challenges to applying coverage-guided fuzzing to industrial controllers. Therefore, the design of *embeddedAFL* must be accompanied by several assumptions. These assumptions have been made carefully with a view to ensuring that the applicability of embeddedAFL is as broad as possible.

First, *embeddedAFL* assumes that the embedded systems (fuzzing target) can maintain trace information in a bitmap. This functionality must not be inherent to the original firmware of the device but could be added by instrumentation code later.

Normally, AFL assumes a compile time random to be used to identify the current branching instruction. As the closed source code cannot be recompiled, another method must be found to identify the branches unambiguously. It is assumed that the coverage measurement of AFL is working correctly, even though the current location in code is not defined by a compile time random, but by the instruction pointer of the current branch, as suggested by Zhang et al. [41].

Even though the source code is not recompiled and automatically instrumented, it is necessary to alter the firmware of the fuzzing target. This means that the instrumentation code must be applied to the firmware and the firmware must be loaded back into the controller.

This work and the requirements from AFL rely on having full control over the target, including the ability to reset the target. Furthermore, full control is needed to retrieve the bitmap out of the device where the bookkeeping of the paths taken happens. This bitmap must be reset from external resources. Eventually, it is required to start and stop tracing individually.

4.2 Design Principles

Thanks to the broad community of researchers, software developers, and other motivated people who have shared their impressive work, this work does not have to start from scratch. In turn, this work lays a foundation on which others can also build. The design principles explained next are followed while developing the software *embeddedAFL*.

The software that is developed in this work should be easy to apply to any other embedded system. Therefore, standard interfaces should be provided or used. The software should be modular so that if other kinds of interfaces are required, it will easy to exchange them. It should be possible to fuzz not only industrial controllers with *embeddedAFL* but any software or firmware that cannot be emulated.

Trace information is necessary to measure the code coverage. Trace accuracy is essential for getting good results. Consequently, tracing should be accurate enough to provide the code coverage information via the bitmap. On top of that, collecting the trace information should not slow down the execution of other tasks in the fuzzing target.

In addition to accuracy, the performance of the fuzzing process is inevitable for a successful fuzzing. The speed per fuzzing round is the most necessary parameter with respect to optimisation, while the time of resetting the device should also be considered. The instrumentation code must not be placed in sections that are part of the normal programme execution to ensure that the normal programme flow is not distorted.

American Fuzzy Lop is continuously developed and receives regular improvements from a large community. It is carefully tested and has been widely applied. Therefore, it is desirable to use the main AFL tools like *afl-fuzz* and others without altering them. This design principle ensures that any further improvement of AFL will lead automatically to an improvement of the fuzzing success of *embeddedAFL*.

Security testing is often required for applications or software whose source code is not publicly disclosed. Thus, the instrumentation of the fuzzing target and its control should be possible, even though there is only very limited knowledge of the operation internals of the device.

The tool that is developed as a result of this work should be able to run on most current operating systems. This should either be achieved by using a programming language that can be compiled against the most relevant operating systems (Windows, Linux, macOS) or by selecting a software platform that is executable on each of the platforms mentioned.

4.3 Approach

The challenges pointed out in Chapter 0 must be overcome to be able to fuzz embedded systems in general and industrial controllers in particular. We should also overcome several shortcomings of AFL. The question is how the gap between AFL and the industrial controller can be filled, as depicted in the figure below:

Figure 4-1 What can fill the gap between AFL and the fuzzing target?

Furthermore, AFL is, in general, not designed to fuzz network protocols. The situation is even worse since we are dealing with embedded systems where we generally have no access to their memory. This work answers this question by filling the gap by developing the *embeddedAFL* tool. Every problem that is faced is tackled with its own interface, so if the fuzzing target changes, an interface can easily be replaced. Furthermore, unit testing is simplified in the development process.

Before the development can start, the assumptions made in Chapter 0 must be fulfilled. The reverse engineering showed that an industrial controller like ABB's AC800F is, in principle, capable of fulfilling these assumptions. The reverse engineering also gained knowledge of where the instrumentation code is placed and how the fuzzer can interact with the fuzzing target. Moreover, the process to load altered firmware onto the controller is defined.

In the next step, the overall design of the gap-filling software is derived from the design principles and requirements.

4.4 System Overview

As previously outlined, AFL is not naturally capable of fuzzing an industrial controller. Therefore, the fuzzing target must be instrumented, and AFL must be extended to *send* the fuzzing data to the target and retrieve the coverage information. The gap between both parties is filled by embeddedAFL and a tool called Kelinci [42], as shown in the figure below:

Figure 4-2 System overview

Kelinci acts as the original fuzzing target from the viewpoint of AFL. It translates the fuzzing input into TCP packages and sends them over to embeddedAFL. The gap filler embeddedAFL is there to handle the fuzzing process, control the fuzzing target, and send over the fuzzing data. There is a tight interaction between embeddedAFL and the instrumentation code of the fuzzing target. After a fuzzing session is completed, the result is sent back to Kelinci. The result consists of information about the paths taken and the status of the device after the fuzzing session is done, e.g. if the device has crashed.

The components of embeddedAFL could have been completely integrated into Kelinci, which is written in C. However, due to the complexity of the state machine, logging requirements, interfaces, and protocols used, the decision is made to use the modern .NET Core framework as the foundation for embeddedAFL. Kelinci could not be replaced by a .NET Core implementation, as AFL is not natively capable of fuzzing .NET Core software.

5 Implementation

It has not so far been possible to apply coverage-guided fuzzing to an embedded system with proprietary software and an operating system that cannot be emulated in a virtualisation environment like QEMU. This gap is now closed by this work. The software developed as part of this work enables AFL with the help of Kelinci to fuzz an industrial controller with proprietary firmware and provide a feedback channel about the paths taken. The system overview is presented in the following figure:

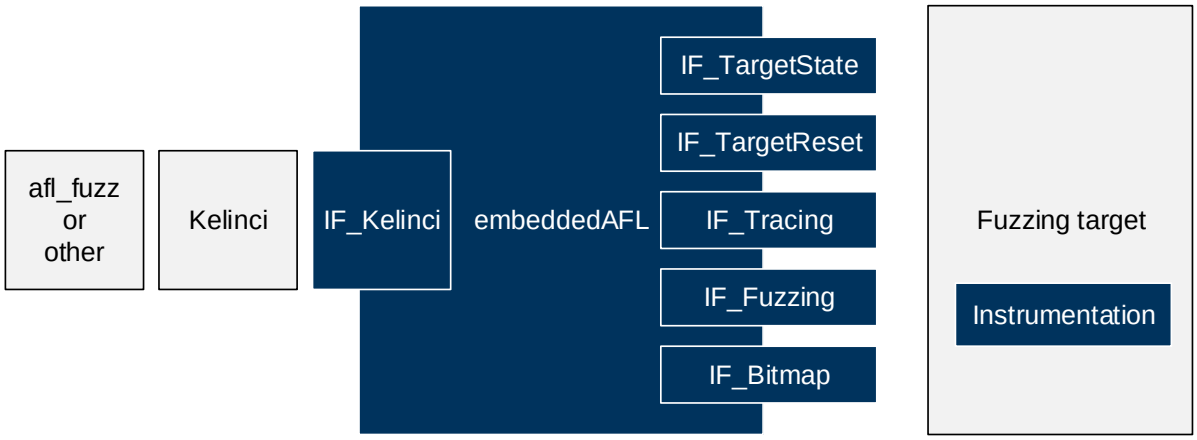


Figure 5-1 System Overview embeddedAFL

The software embeddedAFL consists of several interfaces to bridge the communication with AFL on the one hand and the hardware AC800F controller on the other hand. Furthermore, instrumentation code is needed inside the industrial field controller. Every piece of software that is used in this setup is explained in the following chapters.

5.1 Kelinci

As stated above, the fuzzing data needs to be captured and transmitted to the embedded system. Furthermore, the bitmap from the embedded system needs to be captured and sent back to AFL so it can improve its fuzzing strategy.

Fortunately, a similar task has been solved by Kelinci. Kelinci (the Indonesian word for rabbit) is a project to enable AFL fuzzing for Java projects, as these cannot be fuzzed by AFL natively [42]. This is done by creating a C programme that is fuzzed by AFL but provides the fuzzing data to another webserver. The general setup is depicted in the figure below:

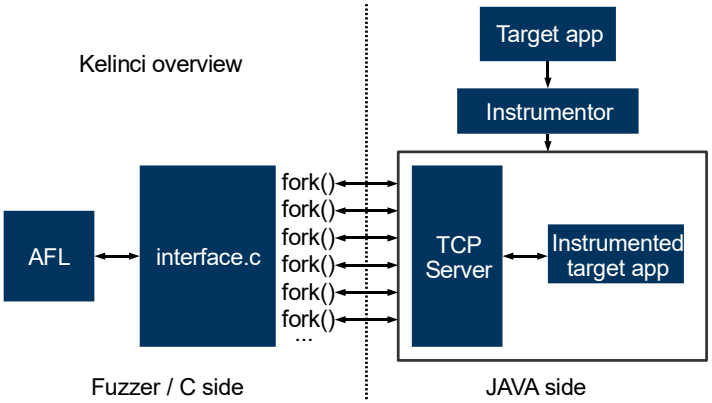


Figure 5-2 Overview of the design of Kelinci [43]

As no Java application is used, the right part is now replaced by embeddedAFL, while the interface is left untouched.

5.2 embeddedAFL

The main outcome of this work is the tool embeddedAFL. This work is intended to not only focus on the current setup but to cover a holistic approach so any other embedded device could be enabled for fuzzing with the help of embeddedAFL. To achieve this high degree of flexibility, embeddedAFL is composed of a variety of interfaces. The functionality and background of these interfaces are presented in the following chapters.

To meet the design principle of operating system independence, embeddedAFL is built in C# with .NET Core 3.1 [44]. The logging functionality is provided by NLog [45].

The main function of embeddedAFL just starts an instance of the interface Kelinci and eventually restarts it if the interface crashes or terminates for any reason. Every interface and task is started asynchronously in embeddedAFL to be able to abort a running execution just by invalidating the cancellation token source [46]. This is done if either a fault is detected or a timeout is exceeded.

5.2.1 The Kelinci Interface

As a first step, Kelinci is used to pack the fuzzing data provided by AFL into TCP packets. As explained above, this is done by the slightly modified version of Kelinci. As a counterpart to the Kelinci instance that is sending TCP requests, a TCP listener is running as part of the Kelinci interface.

The TCP listener receives a Kelinci packet that consists of three pieces of information:

- One mode byte
- Four length bytes
- N bytes of fuzzing data

As soon as this packet is received, the Kelinci interface starts the state machine asynchronously. After returning from the state machine, the bitmap and status are returned to Kelinci. The logical structure is depicted in the figure below:

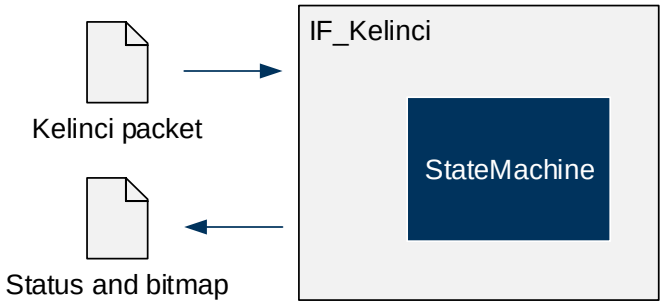


Figure 5-3 Logical Structure of the Kelinci Interface

5.2.2 The State Machine

The state machine handles everything around the complete fuzzing cycle. It is responsible for maintaining the correct state information about the target device and has control over every other interface that is needed in the fuzzing process. The state machine is started every time the Kelinci interface receives a complete Kelinci packet. After completion or in case of an error, the state machine returns the result to the Kelinci interface.

If an error occurs that is not related to a crash event from the fuzzing target, and the complete state machine has not timed out, the fuzzing target is reset, and the fuzzing cycle starts over. The complete fuzzing cycle that is handled by the state machine is shown in the Unified Modelling Language (UML) diagram below:

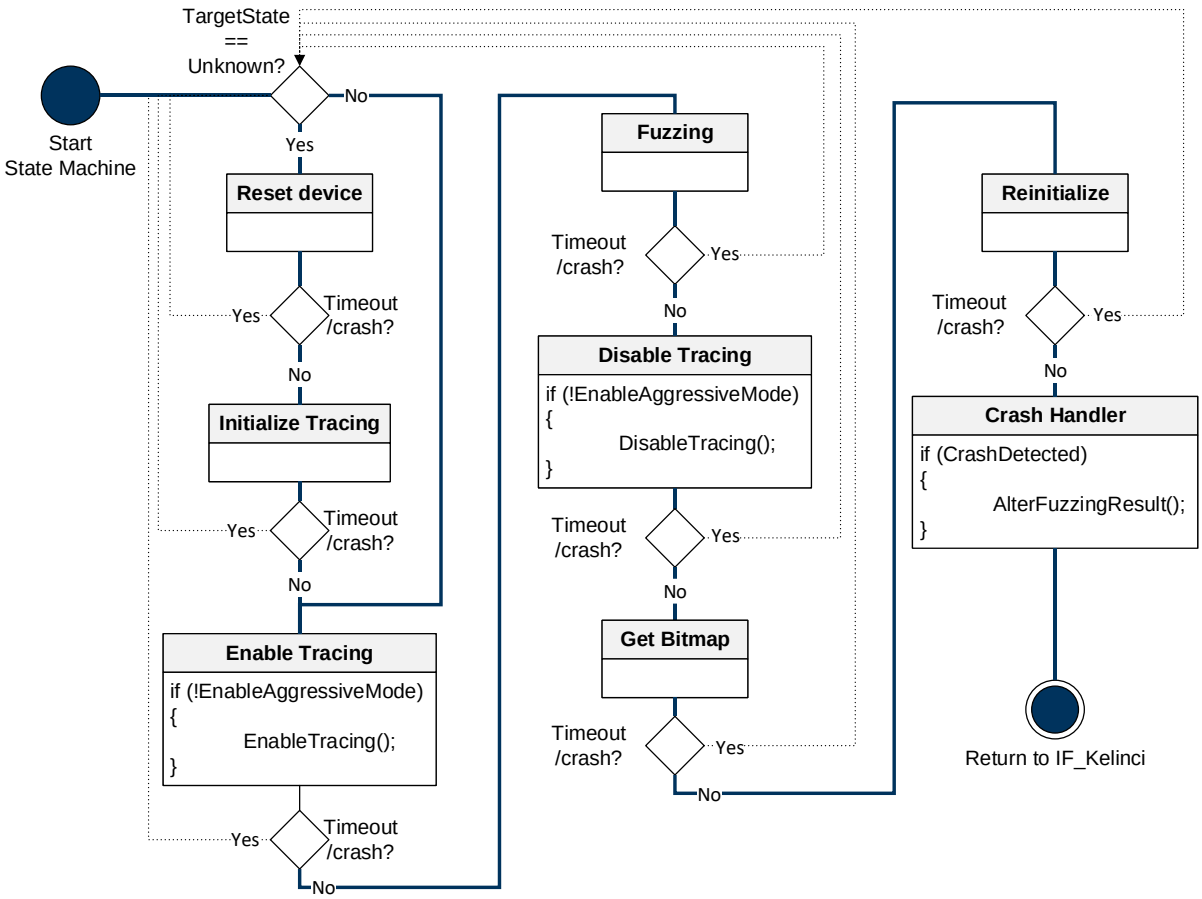


Figure 5-4 UML Diagram State Machine

As soon as the state machine is started, the status of the target is determined. Right after the start of embeddedAFL, the status is set to unknown. If the status is not known, the device is restarted to get a clean starting state. After booting up, the task that enables tracing and allocates memory for the bitmap is executed automatically.

Sometimes a trade-off between accuracy and speed must be made. Therefore, an aggressive mode is introduced.

If the aggressive mode is used, the explicit trace enabling is skipped, and the fuzzing is done immediately. Tracing is disabled in the next step, only if the aggressive mode is not used. Afterwards, the bitmap is downloaded, and the bitmap reinitialisation is done.

If the reinitialisation has finished, the crash handler evaluates whether a crash is detected asynchronously. If a crash is detected, the fuzzing result object is altered accordingly (the status byte is set to *Crash*).

After completion, the state machine returns the fuzzing result to the Kelinci interface. The fuzzing result consists of the 64 kB bitmap and one status byte. The status byte holds the fuzzing status information (*Success*, *Timeout*, *Wait*, *Crash*, *Communication Error*).

5.2.3 Interface Target Reset

If the state of the fuzzing target is unknown, an error occurs, or timeouts are exceeded, then the target reset interface comes into play. The target reset interface controls the physical power supply of the fuzzing target. The interface exposes the two functions *TargetResetIsAvailable* and *TargetReset*. The first function just states whether the target reset functionality is available, while the second function provides the capability of resetting the device.

In a real-world application, this would be done with the remote-controlled power plug *Edimax Sp-1101W*. This plug is connected via Wi-Fi to the embeddedAFL instance.

The reset target interface tests the availability of the plug operation by simply accessing an access protected web page of the internal web server of the plug. If this page is available, the target reset is available. The reset is done by sending two specially crafted XML messages to the plug that switch the plug on and off. The on command is listed below for reference:

```
SMARTPLUG id="edimax">
    <CMD id="setup">
       <Device.System.Power.State>ON</Device.System.Power.State>
    </CMD>
</SMARTPLUG
```

Listing 5-1 Interface target reset “ON” command

This command and the respective off-command are sent via a HTTP POST request to the web server of the plug.

Unfortunately, although a complete reset of the device including reflashing the firmware would also be possible, as the firmware loader can be called from the command line, this has not been implemented so far because different fault states require different procedures.

5.2.4 Interface Target State

The target state interface is a little bit more complex due to its asynchronous workflow and its close interdependence with nearly all other interfaces in embeddedAFL. The target state interface instance is started just once when the main embeddedAFL programme is started. After the constructor succeeds, this instance is passed around to all the other interfaces that need information about the target’s state.

The target state interface needs to continuously monitor the state of the fuzzing target. In the case of the AC800F field controller, this is done by observing the serial telnet interface. Therefore, a thread of its own is launched. Although this might seem like a simple task, it took a great deal of time until the monitoring was stable enough for production testing for the following reasons. First, Microsoft’s standard library for serial communication for .NET Core projects was

used. As it turned out, this library is very poorly implemented and has several flaws that make using it impossible.

The main problem is that the AC800F controller relies on the Request-To-Send and Clear-To-Send signals of the serial connection. These signals are completely ignored by Microsoft's System.IO.Ports [47] class. This has also been noted by Ben Voigt [48], as his comment shows:

> The System.IO.Ports.SerialPort class which ships with .NET is a glaring exception. To put it mildly, it was designed by computer scientists operating far outside their area of core competence. They neither understood the characteristics of serial communication, nor common use cases, and it shows. Nor could it have been tested in any real world scenario prior to shipping, without finding flaws that litter both the documented interface and the undocumented behavior and make reliable communication using System.IO.Ports.SerialPort (henceforth IOPSP) a real nightmare. (Plenty of evidence on StackOverflow attests to this, from devices that work in Hyperterminal but not .NET...)
>
> The worst offending System.IO.Ports.SerialPort members, ones that not only should not be used but are signs of a deep code smell and the need to rearchitect all IOPSP usage:
>
> The DataReceived event (100% redundant, also completely unreliable)
>
> The BytesToRead property (completely unreliable) The Read, ReadExisting, ReadLine methods (handle errors completely wrong, and are synchronous)
>
> The PinChanged event (delivered out of order with respect to every interesting thing you might want to know about it)

The effect is that the spatial separation on the wire is not done and both communication partners are sending at the same time. As also no other library seems to handle the signal lines correctly, but tools like HyperTerminal and PuTTY are working reliably, a radical way is now chosen.

Back in 1995, a Microsoft engineer released a manual [49] that provides a broad foundation for serial communication. Together with this article, he released a perfectly crafted C programme that considers all special features of serial communication. This programme showcases multithreaded TTY communication. The original article no longer contains the source code, but the code has been uploaded to GitHub [50]. This C programme is now transformed into a C library as part of this work. Furthermore, the needed functions are exported, so they are available in the .NET Core programme embeddedAFL. The library must be used on Windows as the serial to USB converter driver is only available for Windows.

To convert the MTTTY example into a library, the example is redesigned to run a ReaderThread that listens continuously and a WriterThread that can be invoked via a UDP datagram. The ReaderThread monitors every bunch of characters that is sent by the fuzzing target and forwards this via a callback to the library implementing function. The basic structure of the C library is shown in the figure below:

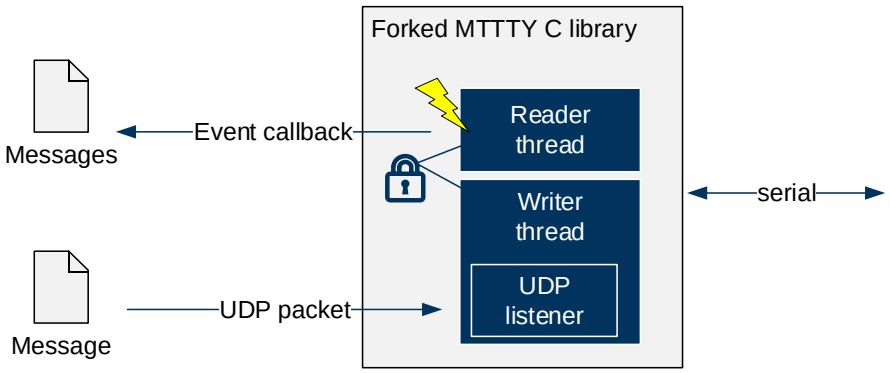


Figure 5-5 MTTTY C library structure

The reader and writer threads are synchronised, so no collision occurs in the serial communication. The UDP package with write requests can be sent at any time and will be buffered if no actual sending is possible.

5.2.5 Interface Bitmap

To improve the fuzzing, the current bitmap must be retrieved out of the fuzzing target. The AC800F field controller is instrumented to provide the bitmap as a 64 kB GIF image via a webserver. The interface bitmap acts like a web browser and downloads the image with an HTTP GET request into a byte buffer. As the buffer is filled with header information from the web request, the first 86 bytes are skipped, and the buffer is returned to the calling function.

Unfortunately, the web interface of the controller is not very stable. This results in an unfinished web response for about 10% of all web requests. As a possible solution, the bitmap transmission is retried until all bytes have been transferred. This often adds up to 4 seconds to the fuzzing cycle. As this is not an acceptable delay, the aggressive mode accepts even partly transferred bitmaps, which are forwarded to Kelinci. In Kelinci the missing bytes are filled with zeros.

5.2.6 Interface Fuzzer

One of the most relevant functions is the fuzzing interface. This interface just has the task of throwing the fuzzing data against the fuzzing target. In the case of the AC800F controller, the fuzzing data is sent to port 9991 via TCP as the DMS communication of the Freelance controller is fuzzed.

The interface opens a TCP socket and tries to send the byte stream to the controller. Whether all bytes are sent, or if any problems occurred or an

exception was thrown, is tracked. If all bytes can be sent successfully, the fuzzer interface just returns to the state machine.

5.3 Instrumentation Code

Coverage-guided fuzzing in general and AFL in particular rely on feedback from the fuzzed software about which code paths have been taken. When using AFL, the instrumented code must maintain a local bitmap. In the local bitmap, all branch tuples are counted. In the AFL whitepaper, the instrumentation code is described as below:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Listing 5-2 Instrumentation code from AFL whitepaper [51]

Besides this fundamental piece of code, some other code snippets must be added to the firmware to be able to control the fuzzing target and to build a foundation for the instrumentation code.

As stated above, the *compile time random* cannot be used because the source code is not available and cannot be recompiled. Therefore, the instruction pointer of the current branching instruction is used.

Unfortunately, no compiler could be found that is capable of translating assembly language to opcodes. Consequently, all the opcode that is needed in the instrumentation and while testing was calculated manually. Additionally, to be able to load the opcode into the fuzzing target, the Intel HEX format must be used. This requires the calculation of the checksum for every 16 bytes of opcode plus their addresses. The opcodes and instruction formats were learned from Intel's i960 Developer's manual [52], and a tool was created to generate the checksums in bulk. Please note that all instructions are in little-endian format.

5.3.1 Instrumenting the menu

Before being able to instrument the firmware of the controller and the fault handling procedure, there must be a way to control the instrumentation code from the outside. The controller has, besides its industrial process interfaces, a serial telnet interface and a web server that can be reached via ethernet. To troubleshoot problems and support commissioning, a menu is available via telnet. Consequently, this menu is used to interact with the controller and run the instrumented procedures.

In the following figure, a switch table is shown that switches on input values sent via telnet. Two of the unused functions are now redirected to call the two functions listed later in this chapter.

```
E0083AEC 14 39 A8 90+   ld      SwitchTable_E0083AF8[g4*4], g5 # switch 7 cases
E0083AEC F8 3A 08 E0
E0083AF4 00 50 05 84    bx      (g5)                           # Branch Extended
E0083AF4
E0083AF4            # ---------------------------------------------------------------------
E0083AF8            SwitchTable_E0083AF8:                    # DATA XREF: Enter_Menu+46C↑r
E0083AF8 50 3B 08 E0+   .long Back_E0083B50                   # 99
E0083AF8 48 3B 08 E0+   .long Output_Destination_E0083B48     # 98
E0083AF8 40 3B 08 E0+   .long Follow_Pointer_E0083B40         # 97 =======>  REDIRECTED
E0083AF8 38 3B 08 E0+   .long Global_Trace_E0083B38           # 96 =======>  REDIRECTED
E0083AF8 30 3B 08 E0+   .long Task_Trace_E0083B30             # 95
E0083AF8 28 3B 08 E0+   .long Keyword_Trace_E0083B28          # 94
E0083AF8 14 3B 08 E0    .long No_way_back_E0083B14
E0083B14            # ---------------------------------------------------------------------
```

Figure 5-6 Original switch table of the menu

The two functions that are called are *the register and start task* and *reinitialize bitmap* function. The first one is called just once, but the second function is called at the start of every fuzzing session. Therefore, performance improvements have a big impact. Normally, the menu options are printed after every ENTER is sent, as shown below:

```
--------------------------------------------------------------------------------
      Main Menue pSOS/960
 back |  output   |follow |global | task  |keyword | lock
      |destination|pointer| trace | trace | trace  |system
  99  |    98     |  97   |  96   |  95   |   94   |  93
   0    MSR-Group                     |   5    free 1
   1    Communication                 |   6    free 2
   2    H&B Libary                    |   7    free_3
   3    Common                        |   8    CGEN-Libary
   4    MSR-OS                        |   9    loaded Modules
 Followpointers = OFF    Globaltrace = OFF
#97INITOK
 ------------------------------------------------------------------------------
      Main Menue pSOS/960
 back |  output   |follow |global | task  |keyword | lock
      |destination|pointer| trace | trace | trace  |system
  99  |    98     |  97   |  96   |  95   |   94   |  93
   0    MSR-Group                     |   5    free 1
   1    Communication                 |   6    free 2
   2    H&B Libary                    |   7    free_3
   3    Common                        |   8    CGEN-Libary
   4    MSR-OS                        |   9    loaded Modules
 Followpointers = OFF    Globaltrace = OFF
#96REINITOK
 ------------------------------------------------------------------------------
```

Listing 5-3 Unoptimised menu output

To speed up the transmission, a short path has been implemented to skip the printing of the menu options and reduce the time before the next command can be sent by nearly 500 ms. The reduced output is shown in the listing below.

```
------------------------------------------------------------------------------
      Main Menue pSOS/960
 Followpointers = OFF    Globaltrace = OFF
 #97INITOK
 ------------------------------------------------------------------------------
      Main Menue pSOS/960
  Followpointers = OFF    Globaltrace = OFF
 #96REINITOK
------------------------------------------------------------------------------
```

Listing 5-4 Optimised menu output

5.3.2 Register and Start a Task

As it is essential to place the bitmap information somewhere in the memory of the ABB AC800F controller, we have to find a safe location to store the trace information. Initial thoughts of just acquiring some memory in an unused area of the RAM failed and resulted in a crashing controller. The reason is that the operating system pSOS+ takes care of the memory and detects memory access to uninitialised memory automatically.

To avoid this situation, the source code of the operating system has been partially reverse-engineered. With the help of the documentation [34,53] and code snippets that were found while reverse engineering the existing firmware, the creation and call of a task can be done in assembly.

To create and start a task, basically, five global registers must be set, and a function must be called. The global registers describe properties of the new task. After the function finished the task creation, it just returns.

The following table describes the global registers, their functions, and their values:

Register	Meaning	Value
g0	Task Name	iAFL
g1	Task Priority	205
g2	Size in stack	128 kB
g3	Tasks first instruction	E008 9190
g4	Task information in stack	4039 985C

Table 5-1 Task properties

The size of the stack that is reserved for the task is being calculated by shifting a '1' bit 17 times to the left and storing the result directly in g2. The size has been extended to 128 kB instead of just the 64 kB that is needed because the placement in the stack is not fully deterministic. With the slightly oversized stack allocation, it is guaranteed that the initialised memory is in the range of the reserved stack memory.

The task priority is chosen at a relatively high priority (maximum is 255) because the processor should not switch the context while global registers are in use.

The instrumentation code is shown in the following listing. In the first part, the used global registers are cached to ensure that every other control flow is not altered afterwards. In the next step, the global registers are initialised and the function to register and start a task is executed by an extended call. And last all used global registers are reverted.

```
0xE0088EF0   10 16 30 5C     mov g0, r6              //Save original global register
0xE0088EF4   11 16 38 5C     mov g1, r7              //Save original global register
0xE0088EF8   12 16 40 5C     mov g2, r8              //Save original global register
0xE0088EFC   13 16 48 5C     mov g3, r9              //Save original global register
0xE0088F00   14 16 50 5C     mov g4, r10             //Save original global register
0xE0088F04   00 30 80 8C     lda "iAFL", g0          //g0 = Task Name stored at E0089190
0xE0088F08   90 91 08 E0     E0089190
0xE0088F0C   CD 00 88 8C     lda 0xCD, g1            //g1 = Task Priority = 205
0xE0088F10   11 5E 90 59     shlo 0x11, 1, g2        //g2 = Size on Stack = 128 kB
0xE0088F14   04 30 98 8C     lda iAFL task, g3       //g3 = Task code stored at E0088F50
0xE0088F18   50 8F 08 E0     E0088F50
0xE0088F1C   04 30 A0 8C     lda 4039985C, g4        //g4 = Stack address = 4039985C
0xE0088F20   5C 98 39 40
0xE0088F24   00 30 00 86     callx Register_Task     //Call “Register and start the task”
0xE0088F28   10 DA 08 E0
0xE0088F2C   06 16 80 5C     mov r6, g0              //Restore original global register
0xE0088F30   07 16 88 5C     mov r7, g1              //Restore original global register
0xE0088F34   08 16 90 5C     mov r8, g2              //Restore original global register
0xE0088F38   09 16 98 5C     mov r9, g3              //Restore original global register
0xE0088F3C   0A 16 A0 5C     mov r10, g4             //Restore original global register
0xE0088F40   00 00 00 0A     ret                     //Return
```

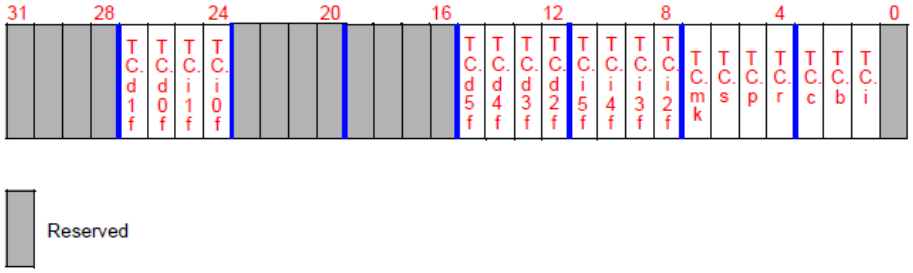
Listing 5-5 Instrumentation to register and start a task

5.3.3 Task to Enable Tracing

The function described in the previous section just registers a task and expects that the task's first instruction is located at 0xE0089190. The code at this location is executed just once after the startup of the controller. It initialises the bitmap area in the stack and enables tracing.

The i960 architecture provides facilities for monitoring processor activity through trace events. A trace event indicates a condition where the processor has just completed executing an instruction and some other cases. When the processor detects a trace event, it generates a trace fault and makes an implicit call to the specific fault handling procedure for trace faults. The fault handling procedure is executed in supervisor mode [52].

Tracing is enabled by modifying the trace controls registers. The process control register must not be altered, even though this is stated by the documentation [52]. In the following figure, the trace controls register is shown.



Hardware Breakpoint Event Flags			Trace Mode Bits		
Mnemonic	**Name**	**Bit #**	**Mnemonic**	**Name**	**Bit #**
TC.i5:2f	Instruction-Address Breakpoint 2 to 5	8-11	TC.i	Instruction Trace Mode	1
TC.d5:2f	Data-Address Breakpoint 2 to 5	12-15	TC.b	Branch Trace Mode	2
TC.i1:0f	Instruction-Address Breakpoint 0 to 1	24-25	TC.c	Call Trace Mode	3
TC.d1:0f	Data-Address Breakpoint 0 to 1	26-27	TC.r	Return Trace Mode	4
			TC.p	Pre-Return Trace Mode	5
			TC.s	Supervisor Trace Mode	6
			TC.mk	Mark Trace Mode	7

Figure 5-7 Trace Controls Register [52]

In the normal execution mode of the controller, all bits are set to zero. AFL requires tracking just the branches that are executed. The following excerpt is from the i960 Developer manual and fits the requirements of AFL perfectly:

Branch Trace

When the branch-trace mode is enabled in TC (TC.b = 1) and PC.te is set, the processor generates a branch-trace fault immediately after a branch instruction executes, if the branch is taken. A branch-trace event is not generated for conditional-branch instructions that do not branch, branch-and-link instructions, and call-and-return instructions.

Figure 5-8 Branch Trace Mode explained [52]

The *branch trace mode* is enabled by setting the trace controls register to 0x4, so the third bit is set.

The following listing shows the instructions for the task. As only local registers are used, there is no need to store the original values and reset them later. The local registers r3–r7 are zero initialised. R8 limits the size of the zero initialised memory. R9 is setting the start address of the bitmap section.

The next four lines initialise the bitmap area. Afterwards, the trace controls register is set to 0x4 and the function waits. After that, the string *INITOK* is printed to telnet. Finally, the task is suspended with a pSOS+ system call so it does not use computing resources but leaves the memory allocated. The return at the end is unnecessary but ends the function in any case if the system call fails.

```
0xE0088F50   00 30 18 8C     lda 0x0, r3           //Initialize r3 to 0x0
0xE0088F54   00 00 00 00
0xE0088F58   00 30 20 8C     lda 0x0, r4           //Initialize r4 to 0x0
0xE0088F5C   00 00 00 00
0xE0088F60   00 30 28 8C     lda 0x0, r5           //Initialize r5 to 0x0
0xE0088F64   00 00 00 00
0xE0088F68   00 30 30 8C     lda 0x0, r6           //Initialize r6 to 0x0
0xE0088F6C   00 00 00 00
0xE0088F70   00 30 38 8C     lda 0x0, r7           //Initialize r7 to 0x0
0xE0088F74   00 00 00 00
0xE0088F78   00 30 40 8C     lda 0x1000, r8        //Initialize r8 to 0x1000
0xE0088F7C   00 10 00 00
0xE0088F80   00 30 48 8C     lda 0x402F4DC0, r9    //Initialize r9 to 0x402F4DC0
0xE0088F84   C0 4D 2F 40     402F4DC0
LOOP:
0xE0088F88   03 5E 22 B2     stq r4, (r9) [r3*16]  //Store r4-r7 at (r9 + (16 * r3))
0xE0088F8C   81 C8 18 59     add 0x1, r3, r3       //Increment r3
0xE0088F90   F8 1F 1A 3C     cmpibl r3, r8, -0x3   //If r3<r8 goto LOOP
0xE0088F94   00 30 78 8C     lda 0x4, r15          //Initialize r15 to 0x4
0xE0088F98   04 00 00 00
0xE0088F9C   0F C2 7B 65     modtc r15, r15, r15   //Modify the trace controls register
0xE0088FA0   8F C0 7B 58     and r15, r15, r15     //Wait
0xE0088FA4   8F C0 7B 58     and r15, r15, r15     //Wait
0xE0088FA8   8F C0 7B 58     and r15, r15, r15     //Wait
0xE0088FAC   00 30 80 8C     lda "INITOK\n\r", g0  //Load string “INITOK\r\n” in g0
0xE0088FB0   A0 91 08 E0     E00891A0
0xE0088FB4   00 30 00 86     callx FEFC704C        //Call function to print g0 on telnet
0xE0088FB8   4C 70 FC FE
0xE0088FBC   06 1E E8 5C     mov 0x6, g13          //Initialize g13 to 0x6
0xE0088FC0   0B 38 00 66     calls 0xB             //pSOS+ System call to suspend a task
0xE0088FC4   00 00 00 0A     ret                   //Return
```

Listing 5-6 Task to enable tracing

5.3.4 Reinitialize Bitmap function

The previous function initialises the bitmap just once. To ensure good coverage results, the bitmap must be initialised before starting a new fuzzing phase. Therefore, the fuzzer needs to have the ability to reinitialise the bitmap with an external command. Also, in this case, the menu command via telnet is used. The command calls a function that is shown in the listing below.

The functionality is exactly the same as in the task function, except that the trace controls register is left untouched and the message that is printed to telnet is exchanged. Furthermore, the function returns instead of suspending.

```
0xE0089040   00 30 18 8C     lda 0x0, r3              //Initialize r3 to 0x0
0xE0089044   00 00 00 00
0xE0089048   00 30 20 8C     lda 0x0, r4              //Initialize r4 to 0x0
0xE008904C   00 00 00 00
0xE0089050   00 30 28 8C     lda 0x0, r5              //Initialize r5 to 0x0
0xE0089054   00 00 00 00
0xE0089058   00 30 30 8C     lda 0x0, r6              //Initialize r6 to 0x0
0xE008905C   00 00 00 00
0xE0089060   00 30 38 8C     lda 0x0, r7              //Initialize r7 to 0x0
0xE0089064   00 00 00 00
0xE0089068   00 30 40 8C     lda 0x1000, r8           //Initialize r8 to 0x1000
0xE008906C   00 10 00 00
0xE0089070   00 30 48 8C     lda 0x402F4DC0, r9       //Initialize r9 to 0x402F4DC0
0xE0089074   C0 4D 2F 40     402F4DC0
LOOP:
0xE0089078   03 5E 22 B2     stq r4, (r9) [r3*16]     //Store r4-r7 at (r9 + (16 * r3))
0xE008907C   81 C8 18 59     add 0x1, r3, r3          //Increment r3
0xE0089080   F8 1F 1A 3C     cmpibl r3, r8, -0x3      //If r3<r8 goto LOOP
0xE0089084   10 16 30 5C     mov g0, r6               //Save original global register
0xE0089088   00 30 80 8C     lda "REINITOK\n\r", g0   //Load String “REINITOK\r\n” in
g0
0xE008908C   B0 91 08 E0     E00891B0
0xE0089090   00 30 00 86     callx FEFC704C           //Call function to print g0 on
telnet
0xE0089094   4C 70 FC FE
0xE0089098   06 16 80 5C     mov r6, g0               //Restore original global
register
0xE008909C   00 00 00 0A     ret                      //Return
```

Listing 5-7 Reinitialize Bitmap function

5.3.5 Instrumentation of the fault handler

The core AFL functionality — the bookkeeping of the bitmap — is packed into the fault handler. As stated in Chapter 3.4, finding the fault handler and the fault table was not an easy task, but as soon as the fault handler was found, the instrumentation could take place. When a fault occurs (e.g. a trace fault), the processor looks up the fault type in the fault table. The definition of the fault table is shown in the figure below.

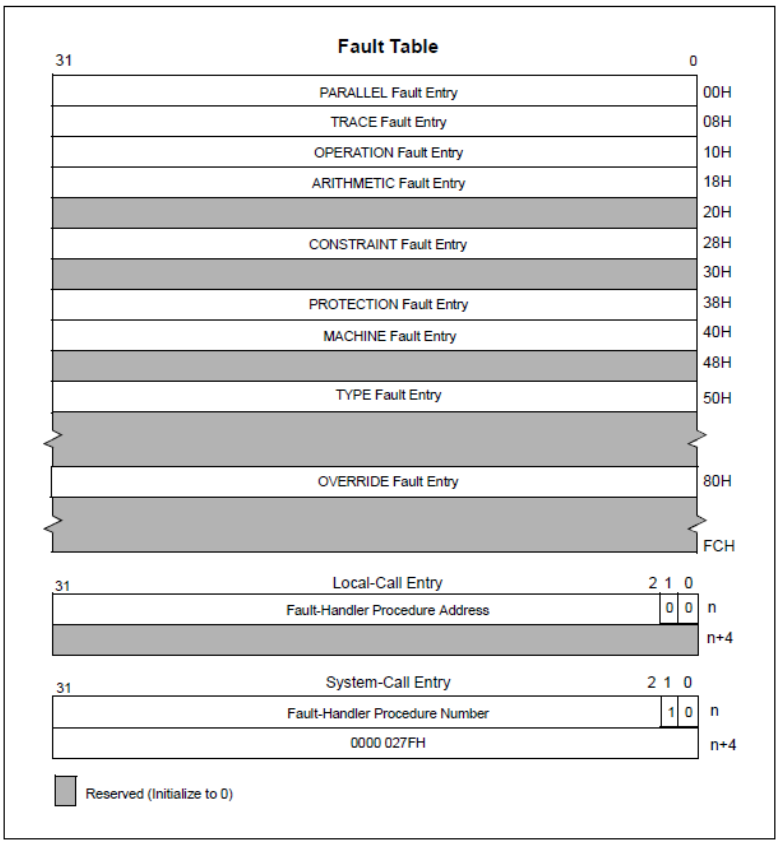


Figure 5-9 Fault table reference [52]

Different fault entries exist for the different fault types, which can be either local-call or system-call entries. In the following figure, the original fault table from the bootloader is shown.

```
400F1190 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F1194 00 00 00 00    .long 0
400F1198 3A 00 00 00    .long unk_3A
400F119C 7F 02 00 00    .long unk_27F
400F11A0 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11A4 00 00 00 00    .long 0
400F11A8 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11AC 00 00 00 00    .long 0
400F11B0 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11B4 00 00 00 00    .long 0
400F11B8 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11BC 00 00 00 00    .long 0
400F11C0 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11C4 00 00 00 00    .long 0
400F11C8 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11CC 00 00 00 00    .long 0
400F11D0 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11D4 00 00 00 00    .long 0
400F11D8 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11DC 00 00 00 00    .long 0
400F11E0 48 79 FC FE    .long DetermineFault_SubType_FEFC7948
400F11E4 00 00 00 00    .long 0
```

Figure 5-10 Original fault table from firmware binary

The entry for trace faults is a system-call entry, while all other entries point to one single function. This function is a standard fault handler, which calculates the

fault type by parameters in the fault record. To handle trace faults only the second entry needs to be overwritten with a pointer to a self-crafted function.

As the least significant bits of the fault table entry are '00' in case of a local call, the first instruction of the fault handling procedure must also be aligned to an address with '00' least significant bits.

In the next listing, the code of the fault handler is shown. This procedure is called every time the processor is branching in the control flow. Therefore, this code has been highly optimised and contains only two memory operations and a mix of calculation and address calculation instructions that can be performed simultaneously in the execution pipeline of the i960 processor.

```
0xE00890B0   81 8D 20 59   shri 1, RIP, r4          //Load previous location id in r4
0xE00890B4   00 F4 1F 90   ld -4(fp), r3           //Load current location id in r3
0xE00890B8   FC FF FFoFF
0xE00890BC   04 C3 18 58   xor r3, r4, r3          //XOR of IPs to determine tuple
0xE00890C0   00 30 20 8C   lda 0x10000, r4         //Load max bitmap size in r4
0xE00890C4   00 00 01 00
0xE00890C8   84 C4 18 74   modi r4, r3, r3         //Fit tuple location into bitmap
0xE00890CC   00 F4 28 80   ldob 0x402F4DC0(r3), r5 //Load previous tuple counter value
0xE00890D0   C0 4D 2F 40   402F4DC0
0xE00890D4   81 48 29 59   addi 1, r5, r5          //Increment the tuple counter r5
0xE00890D8   00 F4 28 82   stob r5,0x402F4DC0(r3)  //Store incremented tuple counter
0xE00890DC   C0 4D 2F 40   402F4DC0
0xE00890E0   00 00 00 0A   ret                     //Return
```

Listing 5-8 Trace fault handler

The original AFL code from Listing 5-2 is slightly altered to improve the performance. In the first step, the *previous location* right shifted by one is stored in the local register r4. In the next step, the current instruction pointer is read into r3. After that, both registers are combined by an xor operation.

Further on, the calculated value is trimmed to fit inside the bitmap. To achieve that, a modulo operation is used. Afterwards, the current value of the tuple counter is read, incremented by one, and written back to the memory location. The effective address is calculated by the start address of the bitmap (0x402F4DC0) plus the remainder of the modulo operation of the xor result.

After the incremented value is stored in the bitmap, the fault handler just returns.

5.3.6 Strings

The tool embeddedAFL communicates the complete fuzzing process to the external industrial controller. Therefore, it is necessary to get feedback from the hardware if several procedures are executed successfully. The following listing shows all necessary strings.

```
0xE0089190   69 41 46 4C                                  //"iAFL"
0xE0089194   00 00 00 00
0xE0089198   00 00 00 00
0xE008919C   00 00 00 00
0xE00891A0   49 4E 49 54                                  //INITOK\r\n
0xE00891A4   4F 4B 0A 0D
0xE00891A8   00 00 00 00
0xE00891AC   00 00 00 00
0xE00891B0   52 45 49 4E                                  //REINITOK\r\n
0xE00891B4   49 54 4F 4B
0xE00891B8   0A 0D 00 00
0xE00891BC   00 00 00 00
```

Listing 5-9 Strings

First, the name of the task is stored. This string is not null terminated, as every task name is exactly four characters long. Both strings INITOK and REINITOK are null terminated and ended by a carriage-return linefeed (0x0A 0x0D).

5.3.7 Exfiltrate GIFs

As the bitmap is maintained in the controller but evaluated in AFL, it must be transferred to the instance of AFL. The transmission of the bitmap is done by embeddedAFL. Initially, the plan was to make the transfer via telnet, but with a transmission rate of only 9600 Baud, this alone would take about one minute. To overcome this limitation, another way was sought.

The ABB AC800F controller provides a web page to show status information for the controller and also present some images. The reverse engineering process showed that the web server is technically answering a request to these images by just starting at a given address and sending over the bytes as long as not the size of the image is reached.

To make use of this surprisingly easy mechanism, the following GIF is altered to send over the 64 kB of the bitmap if the *write.gif* image is requested from the webserver. The following figure shows the original parameters of the image.

```
E0181FBC 10 1E 18 E0    .long aWriteGif                   # "write.gif"
E0181FC0 60 D9 17 E0    .long 0xE017D960
E0181FC4 23 02 00 00    .long 0x223
```

Figure 5-11 GIF name, location and size

These parameters have been changed to transfer the complete bitmap:

Address	Parameter	New value
0xE0181FBC	Requested name	write.gif
0xE0181FC0	Start address	0x402F4DC0
0xE0181FC4	Length	0x00010000

Table 5-2 Parameters of the gif

6 Evaluation

Even though no source code for the industrial controller is available, the instrumentation code together with embeddedAFL must enable a reasonable coverage-guided fuzzing. In this chapter, the performance of the overall system and its components is measured and assessed. The instrumentation code is compared to the original firmware, and the fuzzing result is evaluated.

6.1 Performance

One main indicator for the outcome of the fuzzing result is the number of executions per second. The more executions are done within a timeframe, the more code will be covered, and the more inputs can be tested. Especially in cases with no or few seed values, speed is an important factor. In the next few subchapters, the performance of the complete fuzzing cycle and the performance impact of the different components is measured and interpreted.

6.1.1 Short Circuit Testing

To measure the performance impact of embeddedAFL, it is necessary to measure the base performance of the components that already exist. This is done by implementing short circuits. Each short circuit is set at the first possible point in code, so the impact of the subsequent code is minimal. The two short circuits are shown in the figure below.

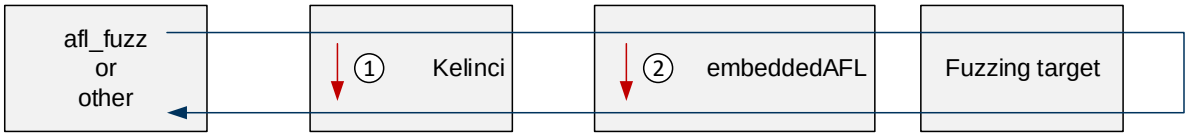


Figure 6-1 Short circuits for performance testing

The first short circuit ① is made directly in the Kelinci source code. The second short circuit ② is set in embeddedAFL. With this testing, the communication protocols are also included in the performance testing. As afl-fuzz and Kelinci are running inside the same VM, the performance is mainly limited by the VM's CPU resources. The communication between Kelinci and embeddedAFL is built on TCP/IP, as embeddedAFL must run on Windows due to driver limitations. Therefore, an impact on the performance in this section can be expected.

The performance is measured by executions per second. To compare the performance components, this value is converted into time per execution.

The test case results are listed in the table below:

Testcase	Throughput (exec/s)	Speed (ms/exec)
① Kelinci short circuit	1980	0.5
② embeddedAFL short circuit	8.3	119

Table 6-1 Short circuit testing performance

Obviously, the instrumentation of Kelinci is very efficient, while the performance overhead added by the TCP connection is present but reasonable.

6.1.2 embeddedAFL Interfaces

Based on the explanations above, one can see that embeddedAFL consists of several interfaces. The advantage of this approach is that the performance of every functionality provided by each interface can be measured individually. While measuring the performance of the individual interfaces, it turns out that there is room for improvement.

First, the time to retrieve the bitmap has been measured with debugging tools in a web browser to get an idea of the performance impact of the instrumented trace fault handler. The time to get the bitmap nearly doubles when the tracing is enabled. This fact led to the idea of only enabling tracing during the fuzzing step. To achieve this, telnet communication needs to be initiated, and the result must be transmitted back to embeddedAFL. The process is shown in the figure below:

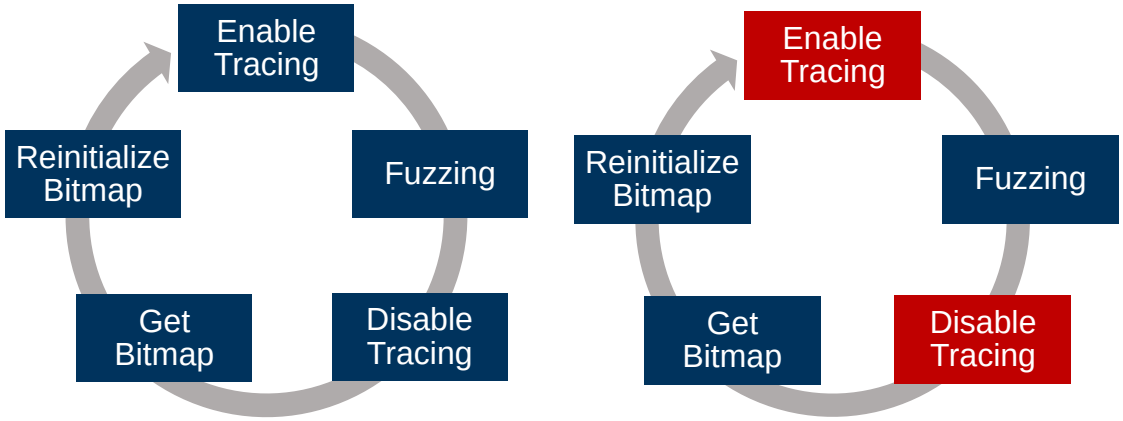


Figure 6-2 Fuzzing process with and without TraceOn / TraceOff

The measurement showed that enabling and disabling the tracing added much more overhead than the tracing impacted the bitmap retrieving function. Furthermore, due to the increased load on the serial communication channel, even the bitmap reinitialisation was highly impacted. The results are shown in the table below. The small deviation in the fuzzing step is related to the slightly different length of input data.

	ms per step with switched	ms per step without

Process step	TraceOn / TraceOff	permanently TraceOn
Enable Tracing	303.80	0
Fuzzing	38.79	32.96
Disable Tracing	329.96	0
Get Bitmap	161.26	277.13
Reinitialize Bitmap	322.84	28.97
Total	**1156.65**	**339.07**

Table 6-2 Time consumption with TraceOn/TraceOff

Due to the measurement results, the tracing is left enabled over the complete fuzzing cycle. This reduced the time per fuzzing session by more than 70%.

6.1.3 Conclusion

The general design of embeddedAFL and its interfaces allows exact measurement of the runtimes of the different steps. As expected, AFL and Kelinci only add a very small amount of overhead to the complete fuzzing process, while the fuzzing target is the real bottleneck. In the graph below, one can see the different contributions that add up to about 459 ms per fuzzing session.

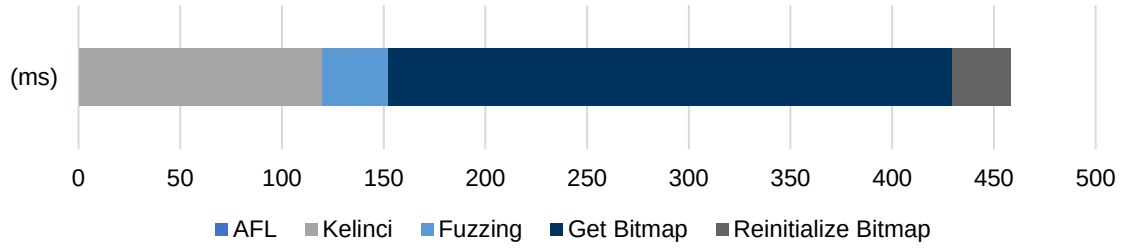


Figure 6-3 Performance impact of fuzzing steps

This results in about 2.18 executions of AFL per second. The greatest impact has the transfer of the bitmap back to embeddedAFL. The transmission is limited by the controller speed and bandwidth and cannot be improved easily. One option would be to reduce the size of the bitmap (e.g. to 32 kB) [51], but this has not been tested so far for this fuzzing target.

6.2 Instrumentation Code

The instrumentation code must be crafted in such a way that the fuzzed code is not interfered with. It is not known if there are free sections to place the code. As a result, the code has been placed in a section that is understood not to be needed at all. A 400 Byte area has been identified, as the code belongs to a menu entry that is overwritten anyway. This area can now be safely used to place the instrumentation.

In general, the instrumentation code is only 99 Bytes. Compared to the original size of the bootloader with 576 kB and the operating system with 2049 kB, this is less than 0.017% of the original code.

6.3 Fuzzing

Even though the performance of the fuzzing process is orders of magnitude worse compared to fuzzing sessions on servers or other multipurpose systems, it still gets reasonably good fuzzing results. Also, the graph generated by afl-plot is like other AFL graphs that show a fast saturation of the graph count overall, which flattens over an increased runtime. Furthermore, the number of pending paths is decreasing compared to the total amount of paths.

The execution speed is also relatively stable and in the range of the expected value. The dips are mainly related to necessary resets of the device, which cause up to 40 seconds of downtime. A total of about 100,000 executions per day is achieved.

Hangs and crashes are reported accurately, and the software recovers properly from hang states. To be able to recover from fault states that have changed the state of the controller, an acoustic alarm has been implemented that goes off when embeddedAFL is not able to recover on its own.

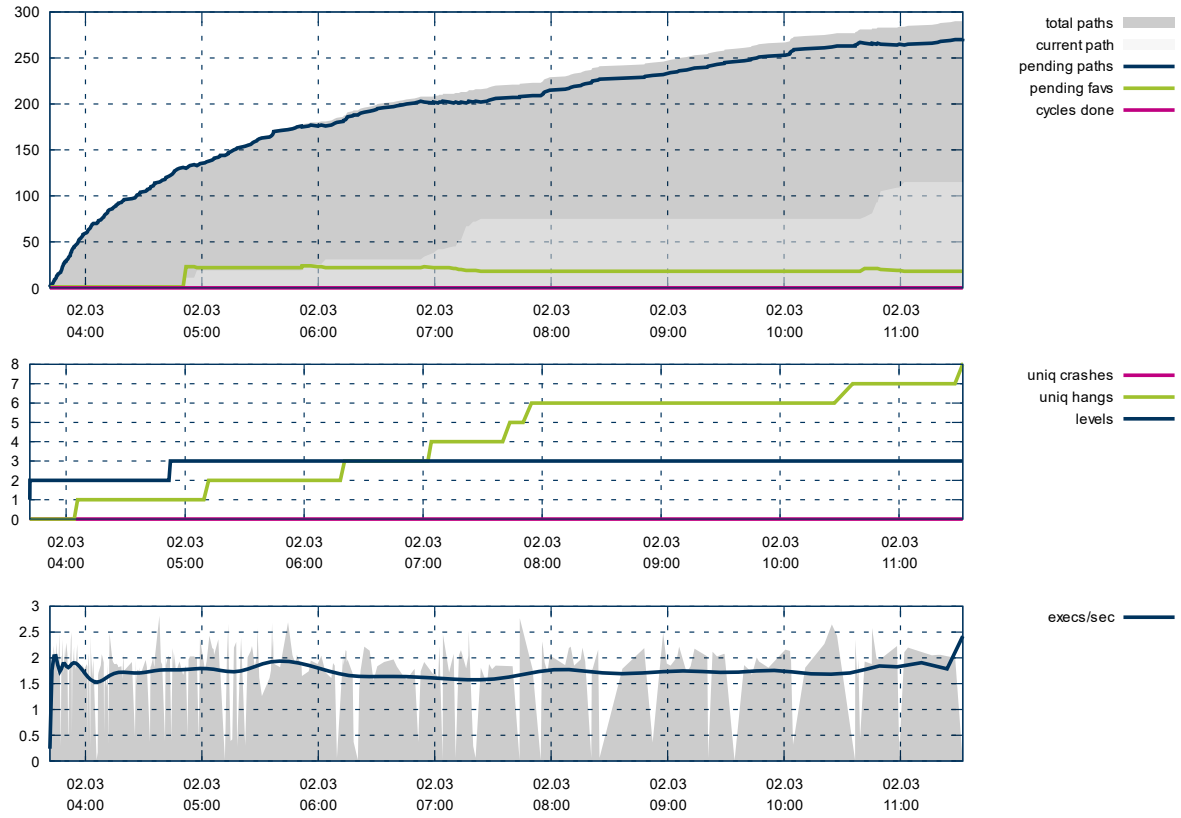


Figure 6-4 afl-plot graphs of about 8 hours embeddedAFL runtime

The detailed output of the AFL console in the figure below gives further information. The execution speed is marked red as expected. The stability is also highlighted, as other threads of the multithreading operating systems are spilling in the bitmap.

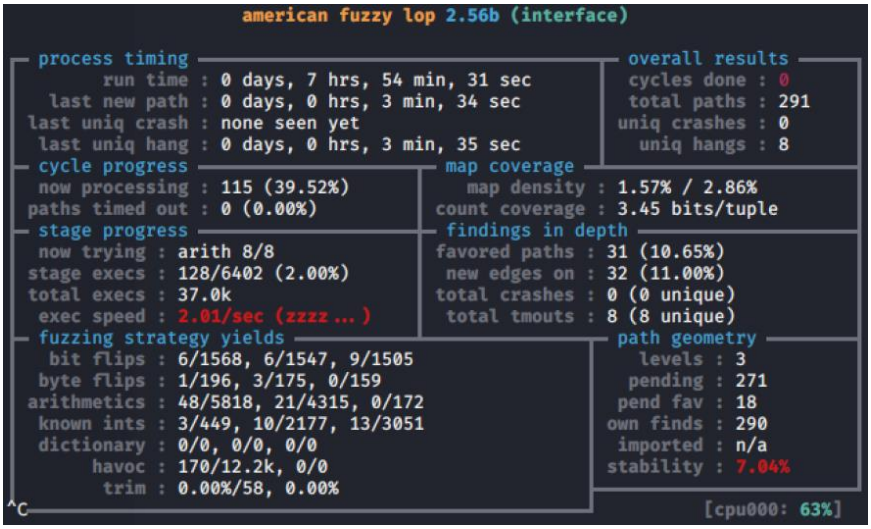


Figure 6-5 AFL console output after about 8 hours of runtime

These results are showing that the instrumentation code and the overall design are working as expected. The further limitations are described in detail in Chapter 0.

The accuracy of the tracing function was tested thoroughly in the development process by populating the bitmap with trace information in a ring buffer–like setup. The call trace was then verified by following the traces in IDA Pro. It turned out that every branching instruction is correctly recorded.

6.3.1 A Valid Project out of Thin Air

The DMS protocol, which is an enhanced MMS (Machine Message Specification) protocol, describes the communication between the industrial controller and a supervising engineering or operator station [15,54]. The understanding of this protocol that is targeted by the fuzzing setup in the proof of concept exceeds the scope of this work. Therefore, as a seed value, just a single '0' is chosen, and it is up to the fuzzer to determine acceptable input data that will eventually cause a crash.

Unexpectedly, AFL was able to load a valid project file into the controller after just about 70,000 iterations. As in 'the project', there was a redundancy link configured, and embeddedAFL was no longer able to reset the device because a check is implemented in the startup procedure of the controller that fails if there is no redundant controller available.

The shortened output of the controller via telnet is shown in the listing below. Obviously, the input data from the fuzzer set a project name and version. The IP address, station number, time zone beside buffer, and variable sizes are configured as well.

```
Boot Configuration:
  Main IP-Addr:  200.120.252.254  [200.120.252.254]
  2nd IP-Addr:   200.120.252.254  [200.120.252.254]
  Red IP-Addr:   98.69.69.69
  Station No:    60
  Project Name:  "E\EXEE▒2EdE▒EEE\EXEE▒^", Version 1162200389
  ObjDir Size:   50454
  GlobVar Size:  13621 KByte
  RedMem Size:   25669 KByte

  Connections:   197/94
  Tasks:         22/22
  Network Buff:  5654 KByte
  TimeZone:      -6175 min
…
Event Log:
176338 BR00  : Event 0x10001013 in BTRPSOS.   Line 1805   26.02.20  5:10:07.348
…
176337 BR00  : Event 0x10001013 in REDMNT.d   Line 1058   26.02.20  5:10:07.348
176339 BR00  : Event 0x10001013 in REDMNT.d   Line 1058   26.02.20  5:10:07.677

No redundancy link available!

*** Fatal Error in Startup, 26.02.20 07:21:27 ***

E-Code: 0x80000308
```

Listing 6-1 Valid project out of thin air reduced error message

7 Related Work

The security of the critical infrastructure is more at risk than ever. Nevertheless, security research regarding the security of their most important components is still very limited. In recent years, some low hanging fruits have been harvested in terms of the security of industrial controllers. In the following section, the current research and trends are discussed.

7.1.1 Security of Industrial Controllers

Industrial controllers are a special type of embedded devices that do their job perfectly well, but only when a malicious adversary is not trying to abuse them. Due to the attacker model of past decades, it was not necessary to implement security controls in industrial controllers [55]. Only after Stuxnet has there been a change of mind in the industry, and security has become more and more important.

Due to their very long lifecycle, industrial controllers that are sold today will still be in operation in 20–30 years. This means that the neglect of security will have an impact on the security of critical infrastructure for a very long time.

Thanks to the diversity of industrial controllers, the security situation is also very diverse [56], not only regarding current actively sold controllers but also with respect to the controllers that are out there in the field. In the following two sections, research is presented that targeted industrial control system security in the past and trends that will affect the future of industrial systems security.

7.1.2 Traditional Security Testing

Traditionally, security research on industrial controllers is mainly done if the industrial controller consists of components that are known from standard computers [57,58] or if a protocol can be analysed by statistical or other remote observation techniques [56,59,60].

Work dedicated to the specifics of industrial controllers is likely to stop at a very superficial level [61] or focus on the logic applications that are executed by these controllers [62]. Also, theoretical work is often done [63].

On top of the research efforts, IEC 62443 conformance certification is available that requires industrial controllers to be tested for vulnerabilities. Unfortunately, obtaining a certificate with such a strong message for an embedded device requires only a scan for known vulnerabilities [64].

7.1.3 The Future of Industrial Controller Security

A transition in security for industrial control systems and their controllers is happening right now. Vendors of industrial devices understand that security features are a selling point. Furthermore, the regulatory authorities are pushing security controls down to the field level, as the critical infrastructure definitions and guidelines are maturing, for example.

Nevertheless, the security of industrial controllers will most likely be undermined by the installation of millions of IOT or IIOT assets with questionable security promises [65,66]. The main task for operators of industrial plants will be to handle the complete diverse portfolio and lifecycle of assets in their plant [67].

The security of industrial controllers will most likely be affected by research that is related to embedded devices in general. As the border between IT and OT will further vanish, it is very likely that more and more IT approaches will also find their way into embedded devices. This includes, for example, Trusted Execution Environments (TEE) [68], Secure Boot [69], and new frameworks for embedded software development [70]. Eventually, more secure operating systems or kernels [71,72] will be used to reduce the attack surface from scratch.

Additionally, research regarding protection approaches for control systems is slowly emerging [26,73,74]. Also, the specifics of embedded devices are subject to security research [28].

Research that targets parts of the security analysis of industrial controllers is already available [66,75].

Despite an intense search, no research has been found that applies intelligent or feedback-driven vulnerability detection to industrial controllers or even embedded devices that could not be emulated [76].

8 Conclusion

In this chapter, the results of this work are presented, and the research questions are evaluated. Moreover, the limitations of the approach, design, and other factors will be considered thoroughly. As a last step, possible future work is laid out.

8.1 Summary of Contributions

Critical infrastructure is more at risk than ever before. Nevertheless, the vulnerability landscape of its most indispensable part, industrial controllers, is still unclear. Several pieces of research have tried to examine the security of industrial controllers, but no case is known where coverage-guided fuzzing is applied directly to an embedded device that has firmware that cannot be emulated.

The goal of this work was to provide a proof of concept demonstrating that coverage-guided fuzzing could and eventually should be applied to industrial controllers. To enable coverage guidance in industrial controllers, the first research question was:

RQ1 Is it possible to retrieve the firmware of an industrial controller and analyse it, to get an understanding of its main parts, structure, and functionality?

As pointed out in Chapter 3, it is possible to retrieve the firmware of an industrial controller with proprietary firmware. In the case of the ABB AC800F controller examined, the firmware consists of different parts that must be obtained with different methods. Even though the firmware is not encrypted, combining the different parts in IDA Pro still requires great effort. Furthermore, with the help of reverse engineering techniques, vital parts of the firmware could be attributed to different functionalities, even though the firmware can only be disassembled and not decompiled.

In a second step, the firmware must be instrumented. Thus, the second research question tackles the instrumentability of the industrial controller:

RQ2 Is it possible to instrument the firmware of an industrial controller and run the instrumented firmware in the controller?

In case of the proof of work, this question can be answered with a definite yes. As shown in Chapter 5.3, the firmware of an industrial controller can be instrumented, and the firmware can still be loaded back into the controller. Moreover, the normal operation of the controller is not interfered with. The instrumentation code is very small compared to the overall firmware.

Instrumenting an industrial controller with a processor that is not widely used in multipurpose computers still requires an incredibly high level of dedication, as the instrumentation code must eventually be written in bytecode.

As an industrial controller can run code that is implanted by a third party, it is now necessary to get trace information out of the controller, as the next research question indicates:

RQ3 Is it possible to extract trace information out of the controller and use this information as coverage guidance for AFL?

Chapter 6.2 shows that it is not only possible to provide trace information to an external party, but that it is possible to retrieve very good trace information from an industrial controller. The trace information recording, furthermore, does not affect the performance of the controller in any way that disturbs it. These are very good conditions to apply a coverage-guided fuzzer to the industrial controller if further requirements are met. These further conditions are determined by the following research question:

RQ4 Is it possible to monitor the current state of an industrial controller and control the hardware at every moment?

As depicted in Chapter 0 and 5.3, the current state of an industrial controller can be monitored in general. With regard to the detection of memory corruption, there is still space for improvements, as shown in Chapter 0. Furthermore, state monitoring requires initial reverse engineering and a good understanding of the hardware and structure of the industrial controller examined. Unfortunately, this is also a task that needs to be done for every other fuzzing target.

All the work described above is targeted towards the goal of applying coverage-guided fuzzing to industrial controllers, as stated in the final research question:

RQ5 Is it possible to run coverage-guided fuzzing against any industrial controller?

As shown by the entirety of this work, it is possible to apply coverage-guided fuzzing to industrial controllers. On top of this fact, which is proved in this work, the overall performance is enough to find the input data that is crashing the controller.

8.2 Limitations

Although this work has overcome all challenges within the proof of concept, there are limitations. First, the exact instrumentation code is only valid for exactly one CPU type. This code is only a few lines long, however this also applies to the concept of the trace information collection in general. For every other CPU type, another way must be found to collect trace information, do bookkeeping in a bitmap, and provide the bitmap to embeddedAFL.

Also, this work is limited by all the ins and outs of embedded devices. The main downside is the speed of only about 2.5 executions per second, which is only achieved in an aggressive mode and due to massive optimisations throughout the whole development process of the instrumentation code and embeddedAFL. It is possible that the developer of the industrial controller is capable of replacing slow serial communication with faster communication via ethernet.

Another problem is caused by the fact that the chosen industrial controller uses a multithreading operating system. As tasks were running in parallel to the investigated thread, the bitmap does not only cover the interesting code areas. This pollution has a significant impact on the stability score of AFL. The problematic embedded web server that is not responding reliably is also limiting either the performance of the fuzzing cycle or accuracy of the bitmap.

As stated by Muench et al. [28], the detection of memory corruption is very limited, and crashes or hangs cannot be directly attributed to input values. Furthermore, it cannot be guaranteed that the fuzzed service is stateless as required. Normally the device is reset before every fuzzing step, but in this case, it would take minutes.

Even though the performance of embeddedAFL does not affect the fuzzing speed very much, the logging overhead is not fine-tuned, and its resource consumption is ultimately too high.

8.3 Future Work

As already mentioned, this work solves many problems, but there is still room for improvement. These improvements can be classified into two major sections. Global improvements are contributions that make a difference to all future fuzzing of embedded devices or fuzzing in general, while local improvements enhance the fuzzing of ABB's AC800F controller.

8.3.1 Local Improvements

ABB's AC800F industrial controller is still actively supported by the vendor. Nevertheless, the interfaces of this controller are not very fast. Eventually, it could help to exchange the serial port communication for faster communication over ethernet.

On the circuit board of the controller, and with the help of the documentation for the Intel i960HT processor and the operating system pSOS+, several other interfaces are available that could optionally be used. Some interfaces may have better performance or could be used for better detection of memory corruption.

Furthermore, the bitmap quality could be improved by preventing other threads from spilling into the bitmap.

As a result of this work, some input data has been found that caused the targeted device to enter a fault state. Eventually, fuzzing could be applied as part of the standard release testing for this controller.

8.3.2 Global Improvements

The software embeddedAFL in combination with Kelinci limits the maximum speed per fuzzing session due to the need to send TCP packets in both directions. If embeddedAFL is developed in C, the impact of this component can then be reduced to nearly zero. Whether this would improve the performance in a way that justifies this complex software project is debatable, but it could bring improvements.

AFL expects a bitmap size of 64 kB, but this is not fixed and can be reduced. Downloading the bitmap together with the cyclic reinitialisation and reducing the bitmap size could improve the fuzzing performance. The pros and cons must be carefully researched.

Eventually, any other coverage-guided fuzzer could deliver better results for different fuzzing targets [77]. Therefore, whether fuzzers other than AFL would have a better performance or detect more crashes should be examined.

Acronyms

.NET Core	Open software framework for Windows, Linux, and macOS
ABB	ASEA Brown Boverie
AFL	American Fuzzy Lop
CPU	Central Processing Unit
DMS	Digimatik Message Specification
DSAC	Device Security Assurance Center
embeddedAFL	AFL for embedded devices
FTP	File Transfer Protocol
GIF	Graphics Interchange Format
GMU	Guarded Memory Unit
HMI	Human–Machine Interface
HTTP	HyperText Transfer Protocol
ICS	Industrial Control System
IDA Pro	Interactive Disassembler by Hex-Rays
Intel HEX	Intel's Hexadecimal Object File Format Specification
IO	Input and Output
IP	Instruction Pointer
IP	Internet Protocol
MMS	Machine Message Specification
PC	Process Controls Register
PROFIBUS	Process Field Bus
pSOS+	Portable Software on Silicon
QEMU	Quick Emulation
RAM	Random Access Memory
RTOS	Real-Time Operating System
SCADA	Supervisory Control and Data Acquisition
SoC	System-on-a-Chip

TC	Trace Controls Register
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
UDP	User Datagram Protocol
UML	Unified Modelling Language
VM	Virtual Machine

List of Figures

List of Tables

List of Listings

References

[1] EU Science Hub, Critical infrastructure protection, 2019. https://www.us-cert.gov/sites/default/files/documents/MAR-17-352-01%20HatMan%E2%80%94Safety%20System%20Targeted%20Malware˙S508C.pdf (accessed 26 February 2020).

[2] NCCIC/ICS-CERT, MAR-17-352-01 HatMan—Safety System Targeted Malware (2017).

[3] Electricity Information Sharing and Analysis Center, Analysis of the Cyber Attack on the Ukrainian Power Grid (2016).

[4] N. Falliere, L.O. Murchu, E. Chien, W32.Stuxnet Dossier (2011).

[5] Kaspersky Lab ZAO, Five myths of industrial control systems security (2014).

[6] C. Walls, Guidelines for using C++ as an alternative to C in embedded designs: Part 1 - Embedded.com, 2008. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-798.pdf (accessed 26 February 2020).

[7] B. Stroustrup, Abstraction and the C++ machine model (2005).

[8] M. Barr, Real men program in C - Embedded.com, 2009. https://www.embedded.com/real-men-program-in-c/ (accessed 27 February 2020).

[9] P. Akritidis, Practical memory safety for C (2011).

[10] D. Papp, Z. Ma, L. Buttyan, Embedded systems security: Threats, vulnerabilities, and attack taxonomy, 2015, pp. 145–152.

[11] A. Gaynor, Modern C++ Won't Save Us, 2019. https://media.kaspersky.com/pdf/DataSheet˙KESB˙5Myths-ICSS˙Eng˙WEB.pdf (accessed 26 February 2020).

[12] M. Sutton, A. Greene, P. Amini, Fuzzing: Brute force vulnerability discovery, Addison-Wesley, Upper Saddle River, NJ, London, 2007.

[13] M. Zalewski, american fuzzy lop, 2020. http://lcamtuf.coredump.cx/afl/ (accessed 26 February 2020).

[14] The International Association of Oil & Gas Producers, Obsolescence and life cycle management for automation systems - Recommended practice — IOGP bookstore, 2020. https://www.iogp.org/bookstore/product/obsolescence-and-life-cycle-management-for-automation-systems-recommended-practice/ (accessed 26 February 2020).

[15] ABB, ABB confirmed as the #1 in Distributed Control Systems globally, 2017. https://new.abb.com/news/detail/3104/abb-confirmed-as-the-1-in-distributed-control-systems-globally (accessed 27 February 2020).

[16] The MITRE Corporation, CVE - Search Results, 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=abb (accessed 26 February 2020).

[17] Blackberry, QNX Operating Systems, 2020. https://blackberry.qnx.com/en/products/neutrino-rtos/index (accessed 28 February 2020).

[18] Wind River Systems Inc., VxWorks, 2019. https://resources.windriver.com/vxworks/vxworks-product-overview (accessed 28 February 2020).

[19] PROFIBUS Nutzerorganisation e.V., PROFIBUS, 2020. https://www.profibus.com/ (accessed 28 February 2020).

[20] Carnegie Mellon University, Automating Vulnerability Discovery in Critical Applications, 2020. https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel˙datapageid˙4050=6487 (accessed 17 March 2020).

[21] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, T. Holz, REDQUEEN: Fuzzing with Input-to-State Correspondence, in: Symposium on Network and Distributed System Security (NDSS), 2019.

[22] J. Li, B. Zhao, C. Zhang, Fuzzing: a survey, Cybersecur 1 (2018) 343. https://doi.org/10.1186/s42400-018-0002-y.

[23] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, T. Holz, GRIMOIRE: Synthesizing Structure while Fuzzing, in: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA, 2019, pp. 1985–2002.

[24] libFuzzer – a library for coverage-guided fuzz testing. — LLVM 10 documentation, 2020. https://llvm.org/docs/LibFuzzer.html (accessed 28 February 2020).

[25] H. Peng, Y. Shoshitaishvili, M. Payer, T-Fuzz: Fuzzing by Program Transformation, in: 2018 IEEE Symposium on Security and Privacy: SP 2018 21-23 May 2018, San Francisco, California, USA proceedings, San Francisco, CA, IEEE, Piscataway, NJ, 2018, pp. 697–710.

[26] A. Abbasi, Race to the bottom: embedded control systems binary security an industrial control system protection approach, Technische Universiteit Eindhoven, 2018.

[27] ABB Limited, contact: The customer magazine of ABB in India, Middle East & Africa (2011).

[28] M. Muench, J. Stijohann, F. Kargl, A. Francillon, D. Balzarotti, What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices, 2018.

[29] ABB, Freelance Distributed Control System: System Description (2019).

[30] ABB, System Hardening & Systemkonfiguration - Cyber Security (System 800xA — ABB), 2020. https://new.abb.com/control-systems/de/system-800xa/cyber-security/system-hardening-systemkonfiguration (accessed 4 March 2020).

[31] ABB, ABB PM 802F, Base Unit 4 MB, battery-buffered RAM, 2020. https://new.abb.com/products/3BDH000002R1/pm-802f-base-unit-4-mb-battery-buffered-ram (accessed 2 March 2020).

[32] Wikiversity, Computer Architecture Lab/WS2007/OhHaHeTa/ThreeISAs, 2020. https://en.wikiversity.org/wiki/Computer˙Architecture˙Lab/WS2007/OhHaHeTa/ThreeISAs (accessed 2 March 2020).

[33] G. Shvets, Intel 80960HT microprocessor family, 2018. http://www.cpu-world.com/CPUs/80960/TYPE-80960HT.html (accessed 2 March 2020).

[34] Integrated Systems Inc., pSOSystem Advanced Topics, 1999.

[35] National Security Agency, Ghidra, 2020. https://www.ghidra-sre.org/ (accessed 2 March 2020).

[36] Hex-Rays SA, Decompilation vs. disassembly, 2020. https://www.hex-rays.com/products/decompiler/compare˙vs˙disassembly/ (accessed 2 March 2020).

[37] R. Kurtz, Frequently asked questions · NationalSecurityAgency/ghidra Wiki, 2020. https://github.com/NationalSecurityAgency/ghidra/wiki/Frequently-asked-questions (accessed 2 March 2020).

[38] S. Bergmans, SB-Projects - File Formats - Intel Hex Files. https://www.sbprojects.net/knowledge/fileformats/intelhex.php (accessed 7 February 2020).

[39] SRecord 1.64, 2014 (accessed 7 February 2020).

[40] Virustotal, YARA - The pattern matching swiss knife for malware researchers, 2020. https://virustotal.github.io/yara/ (accessed 2 March 2020).

[41] G. Zhang, X. Zhou, Y. Luo, X. Wu, E. Min, PTfuzz: Guided Fuzzing with Processor Trace Feedback, IEEE Access 6 (2018) 37302–37313. https://doi.org/10.1109/ACCESS.2018.2851237.

[42] R. Kersten, GitHub - isstac/kelinci: AFL-based fuzzing for Java, 2020. https://github.com/isstac/kelinci (accessed 26 February 2020).

[43] R. Kersten, K. Luckow, C.S. Păsăreanu, POSTER: AFL-based Fuzzing for Java with Kelinci, in: Proceedings of the 2017 ACM SIGSAC Conference on

Computer and Communications Security, Dallas, Texas, USA, ACM, New York, NY, 2017, pp. 2511–2513.

[44] R. Lander, Announcing .NET Core 3.1 — .NET Blog, 2020. https://devblogs.microsoft.com/dotnet/announcing-net-core-3-1/ (accessed 24 February 2020).

[45] NLog Project, NLog, 2020. https://nlog-project.org/ (accessed 24 February 2020).

[46] M. Dotnet-bot, CancellationTokenSource Class (System.Threading) — Microsoft Docs, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource?view=netframework-4.8 (accessed 25 February 2020).

[47] Microsoft, SerialPort Class (System.IO.Ports), 2020. https://docs.microsoft.com/en-us/dotnet/api/system.io.ports.serialport?view=netframework-4.8 (accessed 25 February 2020).

[48] B. Voigt, If you *must* use .NET System.IO.Ports.SerialPort, 2014. http://www.sparxeng.com/blog/software/must-use-net-system-io-ports-serialport (accessed 25 February 2020).

[49] A. Denver, Serial Communications: Multithreaded TTY, 1995. https://docs.microsoft.com/en-us/previous-versions/ff802693(v=msdn.10)?redirectedfrom=MSDN (accessed 25 February 2020).

[50] B. Moran, Multi-Threaded TTY program provided by Microsoft, originally from 1995!, 2017. https://github.com/bmo/mttty (accessed 25 February 2020).

[51] M. Zalewski, Technical "whitepaper" for afl-fuzz, 2017. http://lcamtuf.coredump.cx/afl/technical˙details.txt (accessed 21 February 2020).

[52] Intel Corporation, i960ff Hx Microprocessor: Developer's Manual, 1998.

[53] Integrated Systems Inc., pSOSystem System Calls, 1997.

[54] B. Schliessmann, What protocol does Freelance/DIGIVIS use to communicate with AC800F?, 2020. https://forum-controlsystems.abb.com/19752133/What-protocoI-does-Freelance-DIGIVIS-use-to-communicate-with-AC800F (accessed 2 March 2020).

[55] M. Gjendemsjø, Creating a Weapon of Mass Disruption: Attacking Programmable Logic Controllers, 2013.

[56] J.-B. Bedrune, A. Gazet, F. Monjalet, WHITEPAPER-Reversing-Proprietary-SCADA-Tech.pdf. https://conference.hitb.org/hitbsecconf2015ams/wp-

content/uploads/2015/02/WHITEPAPER-Reversing-Proprietary-SCADA-Tech.pdf (accessed 3 March 2020).

[57] G. Bonney, H. Hoefken, B. Paffen, ICS/SCADA Security Analysis of a Beckhoff CX5020 PLC, 2015.

[58] D. Peck, D. Peterson, Leveraging ethernet card vulnerabilities in field devices (2009).

[59] S.A. Milinkovic, L.R. Lazic, Industrial PLC security issues, in: 20th Telecommunications Forum (TELFOR), 2012: 20-22 Nov. 2012, Belgrade, Serbia, Belgrade, Serbia, IEEE, Piscataway, NJ, 2012, pp. 1536–1539.

[60] M. Niedermaier, F. Fischer, A. von Bodisco, PropFuzz — An IT-security fuzzing framework for proprietary ICS protocols, in: 22nd 2017 International Conference on Applied Electronics: Pilsen, 5-6 September 2017, Pilsen, Czech Republic, IEEE, [Piscataway, NJ], 2017, pp. 1–4.

[61] J. Mulder, M. Schwartz, M. Berg, J. van Houten, M. Urrea, A. Pease, Reverse Engineering Industrial Control System Field Devices. https://www.osti.gov/servlets/purl/1294423 (accessed 4 March 2020).

[62] G. Frey, M.B. Younis, A re-engineering approach for PLC programs using finite automata and UML, in: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IRI-2004): November 8-10, 2004, Hilton, Las Vegas, Nevada, USA, Las Vegas, NV, USA, IEEE Systems Man and Cybernetics Society, Piscataway N.J., 2004, pp. 24–29.

[63] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A.M. Zanchettin, S. Zanero, An Experimental Security Analysis of an Industrial Robot Controller, in: 2017 IEEE Symposium on Security and Privacy - SP 2017: 22-24 May 2017, San Jose, California, USA proceedings, San Jose, CA, USA, IEEE, Piscataway, NJ, 2017, pp. 268–286.

[64] The International Society of Automation, IEC 62443-4-2 - EDSA Certification, 2020. https://www.isasecure.org/en-US/Certification/IEC-62443-CSA-Certification#tab1 (accessed 4 March 2020).

[65] H. Boyes, B. Hallaq, J. Cunningham, T. Watson, The industrial internet of things (IIoT): An analysis framework, Computers in Industry 101 (2018) 1–12. https://doi.org/10.1016/j.compind.2018.04.015.

[66] Z. Bakhshi, A. Balador, J. Mustafa, Industrial IoT security threats and concerns by considering Cisco and Microsoft IoT reference models, in: 2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW): 15-18 April 2018, Barcelona, IEEE, Piscataway, NJ, 2018, pp. 173–178.

[67] A. Keliris, Automated Formulation of Attack Vectors for Industrial Control Systems Security Assessment, 2019.

[68] H. Janjua, M. Ammar, B. Crispo, D. Hughes, Towards a standards-compliant pure-software trusted execution environment for resource-constrained embedded devices, in: Proceedings of the 4th Workshop on System Software for Trusted Execution - SysTEX '19, Huntsville, Ontario, Canada, ACM Press, New York, New York, USA, 2019, pp. 1–6.

[69] C. Profentzas, M. Gunes, Y. Nikolakopoulos, O. Landsiedel, M. Almgren, Performance of Secure Boot in Embedded Systems, in: 15th Annual International Conference on Distributed Computing in Sensor Systems: Proceedings 29-31 May 2019, Santorini Island, Greece, Santorini Island, Greece, Conference Publishing Services, IEEE Computer Society, Los Alamitos, California, Washington, Tokyo, 2019, pp. 198–204.

[70] B. Hamid, J. Geisel, A. Ziani, J.-M. Bruel, J. Perez, Model-driven engineering for trusted embedded systems based on security and dependability patterns, in: International SDL Forum, 2013, pp. 72–90.

[71] D. Elkaduwe, G. Klein, K. Elphinstone, Verified protection model of the seL4 microkernel, in: Working Conference on Verified Software: Theories, Tools, and Experiments, 2008, pp. 99–114.

[72] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, others, seL4: Formal verification of an OS kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 207–220.

[73] A. Cui, S.J. Stolfo, Defending embedded systems with software symbiotes, in: International Workshop on Recent Advances in Intrusion Detection, 2011, pp. 358–377.

[74] Z. Basnight, J. Butts, J. Lopez, T. Dube, Firmware modification attacks on programmable logic controllers, International Journal of Critical Infrastructure Protection 6 (2013) 76–84. https://doi.org/10.1016/j.ijcip.2013.04.004.

[75] A. Keliris, M. Maniatakos, ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries, 2018.

[76] N. Anh Quynh, Virtualizing IoT with Code Coverage-guided Fuzzing, 2018. https://conference.hitb.org/hitbsecconf2018dxb/materials/D1T1%20-%20Virtualizing%20IoT%20With%20Code%20Coverage%20Guided%20Fuzzing%20-%20Lau%20Kai%20Jern%20and%20Nguyen%20Anh%20Quynh.pdf.

[77] Google LLC, FuzzBench: sample report, 2020. https://www.fuzzbench.com/reports/sample/index.html (accessed 3 March 2020).

Eidesstattliche Erklärung

Ich erklaere, dass ich keine Arbeit in gleicher oder aehnlicher Fassung bereits für eine andere Pruefung an der Ruhr-Universitaet Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemaess auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erklaere mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Official Declaration

Hereby I declare that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universitaet Bochum or any other institution or university.

I officially ensure that this paper has been written solely on my own. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

....

Datum / Date Unterschrift / Signature