UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Fuzzing of Industrial Control Systems

Relatori:
Prof. Francesco Baiardi
Prof. Luca Deri

Relatore Esterno:
Dott. Martin Scheu

Candidata:
Francesco Pisani

2023

Contents

Chapter 1

Introduction

This thesis aims to offer an automated security testing tool for Industrial Control Systems, which should enable developers and researchers to improve the security of those critical systems.

Industrial Control Systems

Industrial Control Systems (ICS) form the backbone of modern industrial operations, enabling the control, monitoring, and automation of critical processes in various sectors such as manufacturing, energy, water treatment, transportation, and more. These systems encompass a wide range of technologies, tools, and protocols designed to ensure the efficient and safe functioning of complex industrial processes.

At their core, ICS are responsible for managing and regulating the physical processes that drive industrial operations. This includes tasks like managing machinery, regulating temperature and pressure, handling material flow, and maintaining the overall operational integrity of industrial facilities. What sets ICS apart is their ability to integrate various components, such as sensors, actuators, controllers, and networks, into a unified framework that enables real-time decision-making and process optimization.

The evolution of ICS has been closely intertwined with advances in computing, networking, and automation technologies. Early systems relied on simple analog controls, but the digital revolution introduced programmable logic controllers (PLCs) and distributed control systems (DCS), which enabled more sophisticated control and monitoring capabilities. Today, the concept of the Industrial Internet of Things (IIoT) has ushered in a new era of ICS, where sensors and smart devices are interconnected through the internet, allowing for remote monitoring, data analysis, and predictive maintenance.

Industrial Control Systems have played a pivotal role in enhancing efficiency, reliability, and safety across industries. Manufacturing plants rely on ICS to automate production lines, ensuring consistent product quality and minimizing human error. Energy facilities use these systems to manage power generation and distribution, optimizing resource usage and responding to demand fluctuations in real time. Water treatment plants utilize ICS to monitor and regulate purification processes, safeguarding public health by maintaining water quality standards.

Technological advancements like cloud computing, big data analytics, and machine learning have further transformed ICS capabilities. Data collected from sensors and devices are aggregated and analyzed, providing valuable insights for process optimization, predictive maintenance, and informed decision-making. This data-driven approach allows industries to anticipate issues, prevent downtime, and allocate resources more effectively. Furthermore, Industrial Control Systems are gradually being inserted into ever-growing networks, allowing for large-scale automation over whole regions or countries.

However, the increased connectivity and digital integration also bring about new challenges. The security of Industrial Control Systems has become a paramount concern. Cyberattacks targeting ICS can disrupt operations, compromise sensitive information, and even lead to catastrophic incidents. As a result, the field of industrial cybersecurity has emerged to develop strategies and technologies for protecting these critical systems from malicious actors.

As seen with the 2015 Russian cyberattack on the Ukrainian power grid[7], ICS are becoming an interesting target for attackers, allowing cyberattacks to impact the physical world. The field of industrial cybersecurity is critically needed at this time where ICS system security is terribly underdeveloped, having depended on security by obscurity and airgapping until now. Attackers are still catching up, and cyberattacks tend to not use vulnerabilities in the ICS devices, but instead focus on the control architectures, this is a critical moment in which ICS security must be improved.

Unfortunately, the industry still relies on security by obscurity, with firmware images being hard to obtain and no way to verify the security of devices. This appears to have worked until now due to the air-gapped nature of industrial complexes, but with ever increasing networking many more entry points are becoming available to attackers.

Fuzzing

This thesis intends to offer a fuzzer, a tool capable of automatically finding bugs in ICS devices for which no firmware image is available.

Fuzzers are automated testing tools designed to discover vulnerabilities, crashes, and unexpected behaviors in software applications by bombarding them with a barrage of unexpected, malformed, or random inputs. These inputs can be anything from malformed files and network packets to unexpected user inputs. The goal is to expose vulnerabilities that might not have been identified through traditional testing methods.

Unlike structured testing approaches that rely on predefined test cases, fuzzers take a more exploratory approach. They generate a vast variety of inputs, often in a semi-random manner, to probe different execution paths within the software. This enables fuzzers to uncover obscure bugs and vulnerabilities that might have gone unnoticed during manual or scripted testing.

Fuzzers operate under the premise that software should be able to handle unexpected inputs gracefully without crashing, leaking information, or exhibiting erratic behavior. When a fuzzer discovers a vulnerability or triggers a crash, it provides developers with invaluable information about the underlying flaw. This information aids in fixing the problem and improving the software's overall reliability and security.

Fuzzers have played a crucial role in identifying security vulnerabilities in a wide range of software, including operating systems, web browsers, networking protocols, and more. Their ability to find flaws that could potentially be exploited by malicious actors has already led to their integration into many software development and security workflows.

Fuzzers are already available for some ICS protocols, however, modern ICS protocols offer some unique challenges to fuzzing, which haven't been solved before. This creates a void in tooling which we aim to fill by developing a fuzzer for the MMS protocol. Current blackbox fuzzing approaches struggle to handle the complexity of modern ICS protocols, and as such in this thesis we will showcase how we overcame those challenges in developing our own fuzzer.

1.1 Goals

This thesis aims to develop an effective and open-source blackbox fuzzer for the MMS protocol, aimed at IEC61850 Industrial Control Systems. To the best of our knowledge, currently only one[13] open-source MMS fuzzers exists, which we believe presents serious issues. Giving researchers access to an easy-to-use and effective MMS fuzzer would certainly help in improving security of ICS.

1.2 Thesis Structure

In chapter 2 we will write a state of the art on fuzzing and ICS protocols. We will look at various fuzzing approaches and give a taxonomy of fuzzers. Afterwards we will show an overview of ICS communication protocols, as they will be a focus of this document. The second chapter should give all background knowledge needed to understand the thesis.

In chapter 3 we will present an example Modbus fuzzer, using it to show how current fuzzing techniques may struggle against modern protocols. Afterwards we will present our own fuzzer, explaining our implementation choices and how we overcame those issues.

In chapter 4 we will explain our tests and examine their result, with the goal of validating our work.

Finally, in chapter 5 we will present our conclusions.

Chapter 2

State of the Art

2.1 Fuzzing

2.1.1 Introduction

Fuzzing is an automated software testing technique that involves providing random or invalid data to a computer program. It was introduced in 1990 as a way of testing UNIX utilities. The test consisted of the ”fuzz” program that generated a random stream of data, which was then used as an input for various utilities. Even though the technique appears rudimentary today, it achieved over a 25% failure rate [21], proving itself as a valid approach to automated testing.
Since its introduction, fuzz testing has gained popularity and importance, becoming one of the most widely deployed approaches to discovering software vulnerabilities. It proved to be an essential part of automatic bug finding in the 2016 DARPA Cyber Grand Challenge [6], and Google released its OSS-Fuzzer in 2016 [1] which automatically tests popular open source projects for security vulnerabilities.

2.1.2 Use case

Fuzzers are used to automatically find bugs in programs. This is done by running the fuzzer on a target program until a crash is found, therefore fuzzers must not be run in a live environment, as that would risk impacting the availability of the service tested. Fuzzers should instead be used in an appropriately prepared test environment. Fuzzers are used in the first phases of analysis. In particular, a fuzzer is very effective in finding bugs, but any found bugs need to be further analyzed in order to evaluate the impact of the vulnerability, and how it may be escalated.

2.1.3 Structure of a Fuzzer

A fuzzer fundamentally consists of two parts. One that handles the generation of inputs, and one that instruments the Program Under Testing(PUT) and evaluates the given input. A good test input needs to be:

1. Different enough from expected input to discover bugs and vulnerabilities.
2. Similar enough to valid input to pass most sanity checks of the PUT.

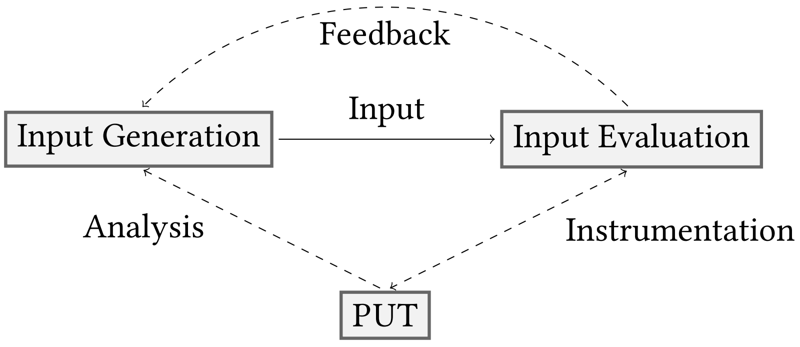

Figure 2.1: Basic fuzzer structure

For example, while completely random data possesses the first property, it will almost always be rejected by the PUT during basic validation. Such inputs wouldn't be very useful in testing, as they would mostly test the relatively simple validation steps. For example a target that rejects inputs not starting with a valid HTTP header would be almost impervious to this type of fuzzing, as we would be very lucky to get one random test case past this check. Completely valid and expected inputs present another problem, while they may test deeper code paths, they will usually stay in already well tested paths. We usually can expect code to function during normal operation, and testing benefits greatly by inserting input that the developers didn't expect.

Proper input generation requires some information on what constitutes expected input. Common techniques involve mutating some captured valid input, hand defining the input space with a grammar, or automatically inferring interesting input from analysis of the PUT. We will go over those techniques in more detail later.

With input evaluation we intend both the act of executing the PUT with the given input, and the detection of any errors in the execution. This can go from simply executing the PUT with given input and detecting any crashes, to much more involved instrumentation and bug detection. Instrumentation covers a wide range of techniques to collect data from a program. In static instrumentation, a program is recompiled, and code is inserted in crucial parts to collect data. For example at each branch instruction code may be inserted to gather information on the direction taken. This would allow the input evaluation step to calculate code coverage of any run. The data gained from the execution may be used as feedback for further input generation, helping in producing quality test inputs. For example, code coverage of a test run may be gathered and used as feedback to guide input generation, by prioritizing inputs with greater code coverage.

2.1.4 Taxonomy of Fuzzers

Fuzzing approaches can be more or less informed depending on how much information on the PUT is available to the tester. In a real-life situation a tester may be fuzzing an open-source application, for which all the internal workings are publicly known. On the other end, the tester might be trying to fuzz software running on a remote machine and for which nothing more than the given inputs and returned outputs are known. The two cases require very different approaches. In the former the fuzzer may use techniques leveraging the available information, whereas in the latter it must cope with this lack of information, which presents a different challenge entirely.

Fuzzers are usually divided in:

- White-box fuzzers, which have access to complete information over the PUT, like its source code, and can use generative approaches based on analysis of the PUTs internals.

- Black-box fuzzers, which can only observe the I/O of the PUT, and cannot gather information about the PUTs internals.

Referring to figure 2.1, in a black-box fuzzer, the analysis step would be impossible, as would be any instrumentation of the PUT. In a white-box fuzzer, the input evaluation may be guided by an initial static analysis of the PUT, for example TestMiner[28] searches for literals in the PUT to generate interesting test inputs. Furthermore, instrumentation is possible, and the data gathered enables some powerful white-box fuzzing techniques.

2.1.5 White-box Fuzzers

White box fuzzers can make use of highly informed techniques, as they have access to deep instrumentation of the PUT. Instrumentation gives the fuzzer information about the memory contents of the PUT at runtime. It may be implemented statically, by compiling the source code with a modified compiler, or dynamically, by using an emulation tool like QEMU[26]. One of the most popular fuzzers, AFL[31], supports both dynamic instrumentation through QEMU, and static instrumentation through a special compiler. It instruments all conditional branch instructions, enabling it to gather complete coverage information of any PUT execution. The gathered coverage information is used afterwards as a fitness function of inputs in a genetic algorithm, iteratively generating test cases with increasing code coverage.

Input generation

AFL shows that white box fuzzers can use execution feedback to evolve test cases. Another commonly used technique is dynamic symbolic execution. The PUT is executed with symbolic values as input, and at each conditional branch the execution is split in two paths. A formula is generated for each path, which, if satisfiable by a concrete input, will be passed to an SMT solver to generate a test input which will lead to the selected path. This technique is computationally expensive, as it involves instrumenting every single instruction of the PUT, but it can also automatically generate inputs that cover any chosen code path. Mayhem[3], the winner of the 2016 DARPA Cyber Grand Challenge[6] , made heavy use of symbolic execution to automatically find exploitable bugs.

Bug Oracles

Bug oracles are the part of the input evaluation that detects bugs in an execution of the PUT. Simple crash detection doesn't require complex bug oracles, but a good bug oracle will be able to discover a wider array of bugs. They are a crucial part of any fuzzer, as with a poor oracle some triggered bugs may go unnoticed. That said defining a perfect bug oracle is almost impossible, as that oracle would need to perfectly understand how the PUT should behave for any given input. It would then be able to find even small implementation errors. Fortunately, security issues tend to be easier to detect. With white box fuzzing programs can be instrumented so that any unsafe memory access is detected as a bug. Valgrind[29] and similar tools can be used for this purpose. More advanced bug oracles are possible, as demonstrated by the OSS Fuzz[1] project which instrumented execve system calls to detect invalid inputs, which would likely be caused by a command injection discovered through fuzzing[20]. This kind of advanced bug oracles can discover wider classes of bugs, not limiting fuzzing to simple crash discovery.

2.1.6 Black-box Fuzzers

Black box fuzzing has no access to information on the PUT, and may only interact with it using its I/O, without further instrumentation. Therefore, techniques are needed to generate good input even when information on the PUTs internals is not available.

Model-based input generation

In a model-based approach, a fuzzer generates input based on a model of valid input. When this model takes the form of a grammar, it is referred to as grammar-based approach.

The model may be predefined, generated by hand to test a specific application or protocol. Kaksonen et al[15] propose a black box protocol fuzzer which takes a BNF grammar defining the protocol to fuzz as configuration. Grammarinator[12] is an open source fuzzer which instead leverages ANTLRv4 grammars of a protocol to fuzz it . Other fuzzers may be built to work on one specific protocol, having a built-in model. For example, funfuzz[9] uses a grammar defining random but valid javascript code to test javascript engines.

Handwriting grammars defining complex protocols is time consuming and error prone, and methods to automatically infer input models have been studied. Learn&Fuzz[11] uses a machine learning approach to infer a statistical input model from a given set of input files, and to use it for fuzzing. Given a set of input files, it may conduct fuzz testing completely autonomously. Models may also be learned from network traffic. ICPFuzzer[25] analyzes a packet capture of traffic of a proprietary protocol, and through a LSTM model it's able to generate inputs and automatically conduct a fuzz test on a proprietary industrial control protocol for which no data is available.

Mutation based input generation

Mutation based fuzzing generates input by randomly mutating well-structured inputs, called seeds. This approach generates inputs that are to an (often controllable) extent similar to the starting seeds. Seed selection is an important challenge, as we would like a set of seeds that cover as many functionalities of the PUT as possible, while avoiding redundancy which would slow down the fuzzing process.

Mutation can adopt various approaches, like bit flipping, in which a number of randomly selected bits of a seed are flipped, generating a new random input. The number of bits to flip controls how mutated our inputs are. Since the effective mutation amount varies per PUT[4], black box fuzzers usually have to try various mutation amounts, hoping to stumble on one that works well.

Dictionary based mutation is also used, in which specific and likely to cause bugs values are substituted to input values. For example the values -1, 0, 1 can be substituted in integers, and "%s" can be inserted in strings to detect format string vulnerabilities.

Bug Oracles

In a black box approach, the detection of bugs becomes difficult, as only clearly visible bugs may be detected. This is usually limited to detecting crashes, but even detecting those may be difficult if fuzzing over a network. Crash detection in those cases is usually done by seeing whether we still receive responses to our test cases, or to correct requests. However, servers often rate limit or block misbehaving clients, and precautions must be taken to avoid those protections. Attempts may be made to detect less visible bugs from output alone, such as unexpected error

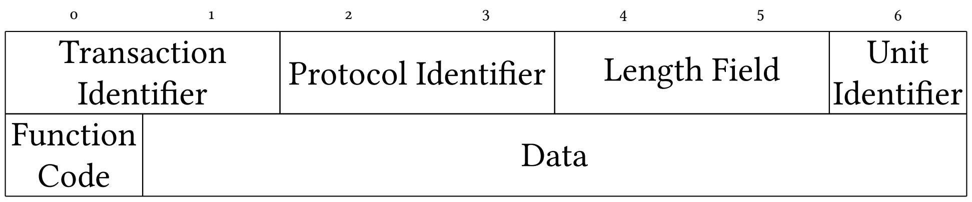

Figure 2.2: Modbus TCP Packet format

responses or entering into invalid states. The automatic detection of those bugs, however, is a difficult problem.

2.2 ICS Protocols

Industrial Control Systems (ICS) are computerized systems used to monitor and control industrial processes. They are essential in the operation of many critical infrastructures, including power generation and distribution, water treatment and distribution, and manufacturing. Industrial Control Systems need to collect data from numerous sensors, interpret it in suitable control units, and eventually send commands to various activators. The challenge of coordinating a large amount of individual devices is met by various networked architectures. These use their own communication protocols, for example Modbus, DNP3, and OPC UA, to enable communication between devices. We will now go in-depth on some communication protocols since those will be the target of our fuzzing.

2.2.1 Modbus TCP

Modbus is a communication protocol used for transmitting information between electronic devices. It was developed in 1979 by Modicon (now Schneider Electric) for use with their programmable logic controllers (PLCs).

Modbus is a simple and open protocol that became a de facto standard in the ICS industry. Since it was designed in the late 70s it presents some limitations, but its simplicity has kept it popular. Many variations of the Modbus protocol exist, such as Modbus RTU which uses a compact binary representation of the protocol, or Modbus TCP[23] which is built to use the network layer.

Packet Structure

We will now look at Modbus TCP more in depth since it has become one of the most widely used versions of the protocol.

As seen in figure 2.2, Modbus TCP packets are composed of a 7byte MBAP header, which gives information specific to Modbus TCP, and a simple Protocol Data Unit (PDU).

MBAP

The MBAP header is composed of:

- 2byte Transaction Identifier: Identifies a request/response transaction, as multiple simultaneous request could be made on the same TCP connection.
- 2byte Protocol Identifier: Indicates the protocol used for intra-system multiplexing, MODBUS protocol is 0.
- 2byte Length Field: Number of following bytes, starting from unit identifier until end of message.
- 1byte Unit Identifier: Used for internal routing, identifies a server behind a gateway.

PDU

The PDU is composed of:

- 1byte Function code: If between 1-127 identifies the Function requested from a client to a server, if between 128-255 identifies an exception response from the server.
- Up to 254 bytes of arbitrary Data

The function code identifies the kind of action that the client is requesting from the server, those can go from Reading/Writing various amounts of data on the device, to diagnostic requests. 17 function codes are reserved for User-Defined functions, allowing each user to implement their own functions, with no guarantee of interoperability with other devices. A full list of supported function codes is available in the specifications[22]. The Data field houses up to 254 bytes, and is formatted according to the Function Code.
The same packet format is used for both requests and responses.

2.2.2 IEC 104

IEC 60870–5-104(IEC104) is a communication protocol standard for control of ICS. It was published in 2000 building upon IEC 60870-5-101(IEC101) a binary serial communication protocol, porting it to the TCP/IP stack. It is focused on power systems, defining data objects specific to those applications, and that's where it's mostly adopted. Working on the TCP/IP stack allows for simpler installation compared to serial lines, and enhanced reliability. It is quite a simple protocol, not allowing for advanced features like automatic device discovery, setting up events and automatic callbacks etc. However its simplicity is one of IEC104's strengths, making for easy implementation and extension on a case-to-case basis.

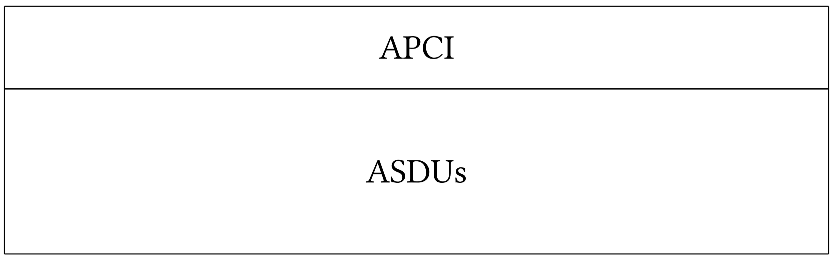

Figure 2.3: IEC 104 Packet format

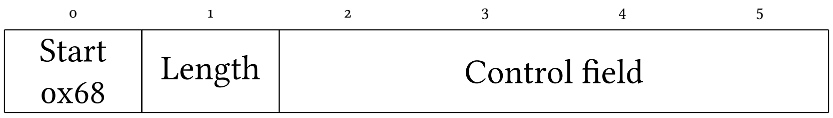

Figure 2.4: IEC 104 APCI format

Packet Structure

As seen in Figure 2.3 an IEC 104 packet is formed by an Application Protocol Control Information (APCI), and zero or more Application Service Data Units (ASDU). The APCI (Figure 2.4) contains a length byte giving the bytecount of following bytes, and a control field, containing information about the kind of packet that was sent, as well as a sequence number, allowing multiple packets to be processed in order[5].
The packets can be of three kinds:

- Information(I): The only packets that contain ASDUs, used for most common functionality.
- Supervisory(S): An empty packet, used to acknowledge I packets.
- Unnumbered control(U): A control packet, allows for stopping/starting data transmission, checking the state of the connection, etc.

Information packets can contain multiple ASDUs, each one being formed of a Data Unit Identifier and zero or more Information Objects. A Data Unit Identifier contains:

- Type ID: The type of the ASDU, and of successive Information Objects. For example, a client may request the reading of a measured value from a sensor, and the type ID would be used to specify the request type.
- Variable Structure Qualifier: Contains the number of Information Objects in the ASDU.
- Cause of Transmission: Indicates what caused the transmission of the ASDU, whether it was a periodic transmission, a response to a request, etc..

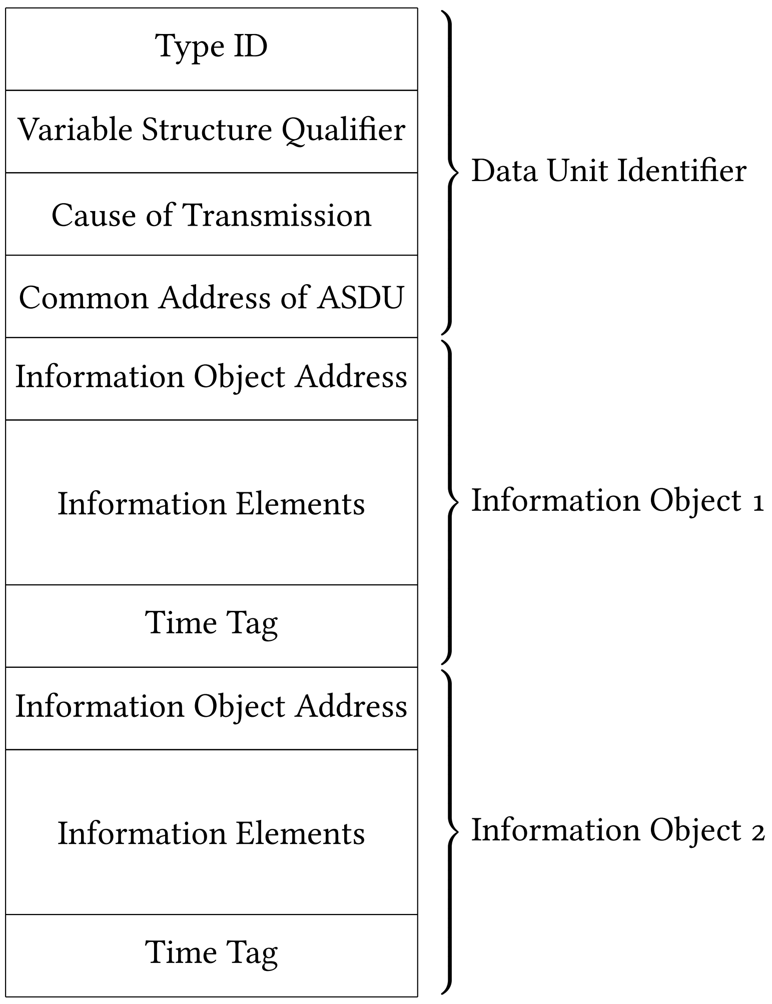

Figure 2.5: IEC 104 ASDU format

- Common Address of ASDU: Identifies the station that should interpret the ASDU, other stations should ignore it.

Information Objects contain the actual data of the packet. This can be the parameters of a request or the data of a response. In IEC104 each ASDU can contain multiple Information Objects, therefore one packet can contain multiple requests or responses. Information Objects have this structure:

- Information Object Address: Identifies the internal address of the subsequent data within a defined station.
- Information Elements: Data formatted accordingly to the Type ID.
- Time Tag: A time tag of when the data was collected.

2.2.3 IEC 61850

IEC 61850 is a standard for communication protocols in the power utility industry which was first published in 2004. It provides a standardized framework for communication in power systems, using object oriented modeling. IEC 61850 defines a set of data models and communication protocols for the transmission of data in ICS. It is quite a large and feature rich standard, offering mappings to 3 different communication protocols, with an aim to expand to web services as well. Those protocols are:

- Generic Object Oriented Substation Events (GOOSE): Offers a fast and reliable mechanism for transmitting time-critical data over entire substation networks. It is a multicast publisher/subscriber protocol.
- Sampled Values (SV): Is a multicast publisher/subscriber protocol, transmitting high-speed periodical updates of the values sampled by a sensor to all subscribed devices on the lan.
- Manufacturing Message Specification (MMS): A unicast protocol allowing data transfer and supervisory functions.

Those protocols are often used in tandem, with SV being used in the lower levels for collecting data from various sensors, GOOSE allowing coordination between units, and MMS mostly being used for management and control functions.

GOOSE

GOOSE is a peer-to-peer publisher/subscriber communication protocol. It enables high-speed exchange of time-critical information between devices. GOOSE is built on the ethernet stack, with a focus on fast and reliable multicast communication. When an event happens in a device, such as a breaker tripping, the device sends multiple transmissions of the same packet to all other devices on the LAN. Sending multiple packets per event enhances the reliability of the protocol. Those packets contain information regarding the data of the event, which would be formatted according to the IEC 61850 standard. Furthermore, they contain information regarding the topic of the transmission. As GOOSE is a multicast protocol, all devices on the LAN receive the packets, but they filter them and process only those of the topics they are subscribed to. GOOSE manages to provide sub-millisecond reliable transmissions, allowing it to be used in time sensitive and critical fields like load-shedding[27].

SV

SV is a communication protocol which specializes in transmitting digital representations of analog signals. It takes samples of an analog signal at a fixed sampling rate, and constantly sends them to all devices on the LAN. SV is widely used as a standardized way of digitizing and collecting analog signals from power devices to control or monitor devices.

MMS

MMS(Manufacturing Message Specification) was first published in 1990 with the goal of offering standardized and interoperable communication between devices from different manufacturers. IEC61850 later adopted this protocol as a part of its software stack, by creating a mapping of IEC61850 objects to MMS objects. MMS is designed as to be independent of the underlying data and hardware, and only defines a set of generic objects and operations that a server should implement. Those operations go from reading and writing data, to more complex management, such as starting processes, uploading or downloading large amounts of data (for example a software update), or defining automatic actions that a server should take on receiving an event. With its advanced features MMS is perfectly suited for high level management, as it gives operators powerful tools to manage whole power plants. While MMS doesn't define any domain specific objects, it allows the definition of arbitrarily complex data types, simplifying its adoption in distinct fields. This enables the mapping of IEC 61850 power specific objects to MMS. ASN.1 (Abstract Syntax Notation) is used the specification to define the syntax of its messages. As such, MMS messages can be seen as a sequence of ASN.1 objects MMS is defined independently of its communication protocol, and doesn't specify how it's messages should be routed or encoded. Within IEC61850 MMS packets are encoded using ASN.1 BER and routed over the TCP/IP stack.

Basic Encoding Rule

Basic Encoding Rule (BER) is a rich encoding protocol, which represents ASN.1 objects as a Type Length Value (TLV) triple. In each triple, the type and length

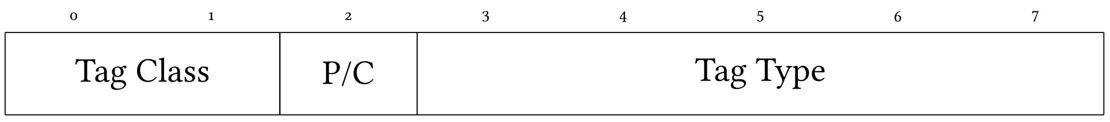

Figure 2.6: BER Tag field

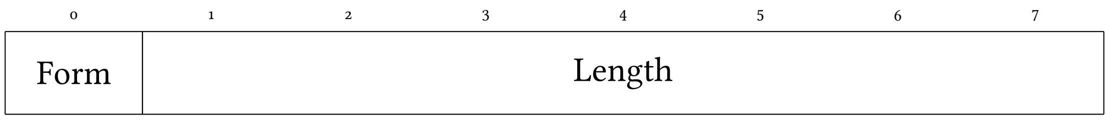

Figure 2.7: BER Length field

indicate how to interpret the successive value field. As an MMS packet is a list of ASN.1 objects, it is encoded to a sequence of TLV triples. The Type encodes the ASN.1 tag field, can be composed of one or more bytes, and represents the type of the object, which can for example be String, Integer, or any kind of abstract ASN.1 object. It also identifies whether the object is "constructed" and contains further objects as its value. In figure2.6 we can see how a tag is encoded:

- 2byte Tag Class: Contains the "class" of the tag, identifying whether the tag is native to ASN.1 or in which context it was defined if not.
- 1byte Constructed bit: Is set if the type is constructed.
- 5byte Tag Type: Identifies the type. If all bits are set (Type = 31) specifies a long encoded tag, with following octets identifying the actual type.

In figure2.7 we can see the encoding of a BER Length:

- 1byte Form bit: If set indicates a long form length, otherwise indicates a short form.
- 7byte Length: In short form contains the length of the value field. In long form contains the number of following length octets.

In long form, the first octet indicates the number(1-127) of following octets, which will contain the actual length of the value field. Therefore, a value field can be as long as $2^{127*8} - 1$.

The Value field contains length bytes of arbitrary data. The data is interpreted according to its type, and as MMS allows defining arbitrary types can take any form. If the type is constructed, however, the Value field actually contains a sequence of TLV triples, which can in turn be of constructed type. This allows the encoding of complex types, such as lists.

2.2.4 OPC-UA

OPC-UA (Open Platform Communications Unified Architecture) is a widely used industrial communication standard. It was first released in 2006 with the objective of providing a more modern approach to industrial automation. OPC-UA provides a standardized framework for information exchange, offering a high degree of abstraction from the underlying hardware. This allows interoperability and integration of various devices and systems. With its wide range of functionality, it has also been used for cloud and IOT devices. Some of its most prominent features are its support for security features like encryption and message signing, its open source nature, and its extensibility. OPC-UA has a different approach to ICS compared to other standards. For example, IEC61850 takes a hierarchical approach. It separates ICS systems in low-level devices, such as PLCS, a SCADA layer which collects data and coordinates them, and a high level human interface which allows operators to efficiently control the whole process. OPC-UA instead looks to put all devices on the same network[17], with sensors communicating directly with human interfaces. This puts it at the forefront of the switch to Industry 4.0, with smarter IOT devices being integrated in production processes. At its core OPC-UA defines a number of services that a server might implement. Not all servers need to support all services, they instead can choose which profiles (sets of services) to implement. A service is an abstract remote procedure call that a OPC-UA Client might make to a server, examples might be a Read/Write of data, or GetEndpoints which returns a list of endpoints supported by a server[24]. This last kind of advanced service discovery shows the power of OPC-UA, and how it distances itself from older and simpler protocols like Modbus and IEC104. OPC-UA also gives the possibility of querying historical data from servers, effectively giving access to logs of system activity in all devices. OPC-UA supports various encodings. Requests and responses may be encoded in binary data, XML or JSON, and be transported over TCP, HTTP/S, or even WebSockets. This wide array of supported technologies allows OPC-UA to adapt to varied network environments.

Chapter 3

Architecture and Implementation

We chose to implement a blackbox fuzzer for MMS as MMS is a crucial part of the widely adopted IEC61850 protocol, and its use as a high level control language makes it the first point of attack against an Industrial Control System. In IEC61850 lower layers are not exposed to the internet, but are instead only connected to upper control layers, therefore an attacker would need to enter the system through its human machine interface, which uses MMS. The fuzzer has to be blackbox as usually researchers will not have access to firmware images, but only to the finished machine with which they may interact over the net. If firmware images were available then current whitebox fuzzers such as AFL[31] would probably be effective. Our fuzzer looks to be simple to use, allowing users to, once they set up a target in a lab environment, automatically conduct fuzzing just by launching the tool against the target.

We will first present the architecture of an example fuzzer for the Modbus protocol, as we believe this to be a good example of how ICS fuzzing has usually been approached. Afterward, we will go in depth on our own fuzzer, and see which challenges we had to overcome in fuzzing MMS.

We will split ICS protocols in "old-generation" and "modern" protocols. This somewhat arbitrary categorization underlines the changes in protocols going into industry 4.0. We categorize as old-generation protocols those characterized by simple implementations and relatively limited functionality. They usually support some simple function calls, and leave most of the handling to the users. This works perfectly well for simple systems where standardization is not essential and devices do not need to be particularly smart or independent. However, those protocols don't support the level of abstraction needed for building large dynamic networks, a necessity for the switch to industry 4.0.

In contrast to these modern protocols, those built with an eye for industry 4.0, tend to be much more complex. They are characterized by rich functionality and great abstractions from the underlying implementation. This is justified by

Listing 3.1: Read coil request
```
#Defines a read coil request
s_initialize("modbus_read_coil")
with s_block("modbus_head"):
    #Each s_word or s_byte identifies a different field,
  those are the fields of the MBAP
    s_word(0x0001,name='transId',fuzzable=True)
    s_word(0x0000,name='protoId',fuzzable=False)
    s_word(0x06,endian='>',name='length')
    s_byte(0xff,name='unit Identifier',fuzzable=False)
    #Here we define the PDU fields
    with s_block('pdu'):
        s_byte(0x01,name='funcCode read coil memory',
  fuzzable=False)
        s_word(0x0000,name='start address')
        s_word(0x0000,name='quantity')
```

the modern need to build a web of interconnected and independent standardized smart devices.

3.1 Architecture of a Modbus fuzzer

Modbus along with IEC104 are clear examples of old-generation protocols. As seen in the state of the art, messages of those protocols have a simple structure, and the structure itself stays mostly unchanged between different messages. An effective approach to fuzz those protocols is to identify the structure of those messages, and then define a model of those messages. Once the model is available specialized libraries can be leveraged, which allow generative fuzzing of any protocol by just specifying its structure. Fuzzing old-generation protocols is usually done with this straightforward and effective generative approach. We will examine this common approach in order to showcase how it struggles against more complex ICS protocols.

We will now analyze the open-source fuzzowski[10] protocol fuzzer as a representative example of a Modbus fuzzer. Fuzzowski is a black box generational fuzzer based on a fork of Boofuzz[2]. The generative approach is done by identifying the fields present in a subset of Modbus requests and then iteratively generating bytes to fill them. This is easily done by hand as requests have a simple structure, which doesn't change much between different requests. This is a common and effective approach and allows for a deep coverage of the Modbus protocol.

The snippet of code in listing 3.1 defines a "modbus_read_coil" request for the Boofuzz framework. It identifies a series of fields (with s_word), and chooses whether they should be fuzzed or not. This definition tells Boofuzz everything it needs to know to successfully fuzz this part of the protocol. It will later use its internal algorithms to generate a series of requests by mutating the fields identified as fuzzable. For crash detection, fuzzowski sends a simple modbus query on a new connection after each test case, thus checking whether the last test case made the target crash.

This is a standard approach to blackbox generative fuzzing, which we also find in some published fuzzers such as the framework proposed by Ilgner and Fujdiak[14].

This is not to say that fuzzing of old-generation protocols is a solved problem. Many interesting ideas are being explored on how to effectively guide the generation of such fuzzers. MTF[30] and its successor MTF-Storm[16] implement an interesting reconnaissance phase, and manage to automatically tailor their fuzzing to the target even in a blackbox environment. ICPFuzz[25] proposes the use of LSTM to learn the features of a proprietary protocol from traffic data and to further guide the generation through analysis of ongoing traffic.

3.2 Architecture of our MMS fuzzer

In contrast to old-generation protocols, IEC61850 is more modern and directed towards industry 4.0. Furthermore, the MMS protocol itself is much richer in functionality than Modbus. Because of this MMS offers some peculiar challenges to a generative approach. The structure of an MMS message is much more complex than that of Modbus. Furthermore, it changes drastically with each message, and even with each implementation. MMS messages support arbitrary user defined objects, and therefore a generative approach clearly becomes difficult to implement. Writing definitions by hand would necessarily cover only a small part of the protocol, and the process would require a large amount of man-hours, with all associated human errors. On the other hand, a mutational approach, where seed messages are mutated, would be easy to implement, but would most likely lead to inefficient testing. By randomly mutating messages we would often modify the wrong fields, and make many of our test cases completely invalid, and quite uninteresting. If we were to randomly modify a BER encoded message, we would likely modify the length field of a TLV triple and completely invalidate all following triples. Furthermore, this lack of knowledge of the packet structure would completely impede our fuzzer from creating coherent packets where, for example, a value field was extended or shortened.

3.2.1 Our solution

In order to solve those issues, we needed to develop an automatic and structure aware fuzzer. Our fuzzer extracts the structure of MMS messages from a set of seed requests. Afterwards, it is able to set up a generational approach over those structures, allowing for total protocol coverage as it can analyze any legal MMS request. This approach, which straddles the line between mutational and generational approaches, is a good fit for MMS. Uninformed mutational approaches would be limited by their lack of knowledge of the underlying protocol. While this makes them faster to implement, they would tend to generate wildly invalid packets, easily rejected by the PUT. Thus they would spend most of their time on testing input validation instead of deeper parts of the code. By setting up a proper generational approach dynamically, however, we manage to generate valid or (close enough to valid) packets that tend to pass the first validation of the PUT. This setup phase is the heart of our approach, and makes for a fast and automatic equivalent to a hand built generational approach.

3.2.2 Architecture

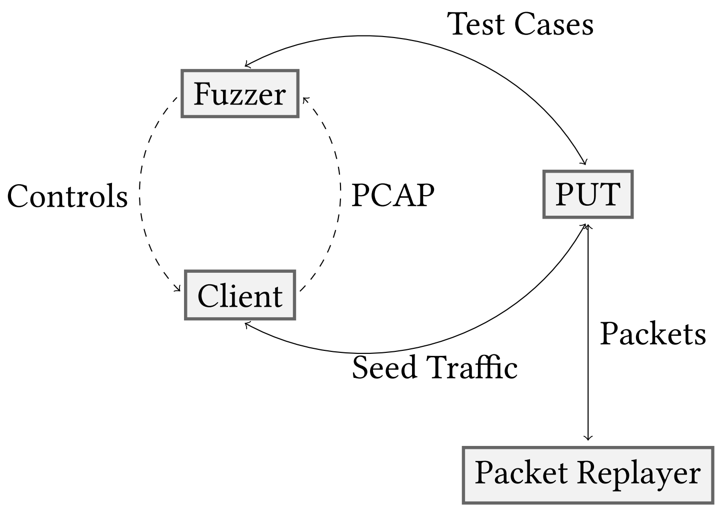

Figure 3.1: Basic Architecture

We developed:

1. A python fuzzer built on top of the boofuzz framework.
2. A custom MMS client written in C.
3. A python packet replayer allowing resending of test packets.

Apart from the main fuzzers, we developed both a custom client and a packet replayer. Those are useful utilities that make our fuzzer easier to use. In order to develop an easy to use fuzzer with this approach, we needed a way to automatically generate seed traffic. We could have left this to the users, requiring packet captures as setup for the fuzzer, but we believe that this would have made our fuzzer quite cumbersome. We instead elected to develop a custom MMS client, that automatically makes a series of requests which we can capture. When our fuzzer is launched without seed files, it automatically launches the client to make some requests against the target, which it collects in a packet capture. This way fuzzing a target is as simple as launching ./fuzzer.py -t ip:port. The fuzzer also supports external traffic files, if that is preferred by the user. Having both choices allows our fuzzer to be both simple to use out of the box, while keeping open to full customization of seed traffic if needed. We also developed a packet replayer, which allows to replay any test case from a run. This enables users to efficiently replicate any found bugs, and should help in investigating vulnerabilities.

3.2.3 The Fuzzer

The MMS fuzzer functions as a blackbox generational fuzzer. It first generates seed traffic by launching the client and capturing its traffic with the target. Afterwards, it analyzes the captured traffic and extracts the structure of the captured requests. At last, this structure is used to define a generative fuzzing process, which is then handled by the boofuzz framework.

3.2.4 The MMS Client

A client used for seed traffic generation. Once launched it makes a series of valid requests to the server, which are used as seeds for further fuzzing. This automatic traffic generation is needed to simplify the usage of the fuzzer, but the fuzzer also supports PCAP traffic files generated independently. The client supports various coverage options, generating only the required subset of requests. This makes it possible to select more interesting requests depending on the target.

3.2.5 The Test Case Replayer

We offer a test case replaying tool to help in repeating and analyzing found crashes. It allows one to resend any chosen test case exactly as during the fuzzing process. In this way, users can trigger the crash again during debugging, simplifying the investigation of found bugs.

3.3 Implementation

3.3.1 The Fuzzer

The fuzzer is written as a python command line tool. The fuzzing functions are implemented using the open source boofuzz[2] fuzzing framework. Boofuzz was chosen as it is a mature python framework, with a focus on black box protocol fuzzing. As already seen it is used in fuzzowsky[10], and many other ICS blackbox fuzzers. It independently handles many of the common parts of fuzzing, such as saving test results and logging. Boofuzz also natively contains mature generation

Listing 3.2: Handshake extraction

```
cap = pyshark.FileCapture(pcap, display_filter="cotp")
handshake1 = packetToBytes(cap[0].tcp.payload)
handshake2 = packetToBytes(cap[2].tcp.payload)

handshake1 = Request("handshake1",
    children=[Static(name="handshake1", default_value=
   handshake1)])
handshake2 = Request("handshake2",
    children=[Static(name="handshake2", default_value=
   handshake2)])
cap.close()
```

algorithms for most fuzzable types allowing us to focus on other matters. The fuzzer takes many options as command line arguments, which are parsed using the argparse library, allowing customization of the fuzzing process.

We will now describe the fuzzer in further detail, following it step by step in a standard execution.

Traffic Generation

Once launched, if no seed traffic is provided to the fuzzer, it will automatically generate some. To do so, it uses shell commands to start a tcpdump traffic capture, start the MMS client against the server, and stop the capture once it's done. This guarantees ease of use, with a completely automated traffic generation step. In fact running our fuzzer only requires the target specification, and is otherwise completely automatic (while still offering command options to customize the process).

Packet Analyzer

Once the seed traffic is available, it needs to be analysed to setup the fuzzing process. To do so we use the pyshark python package which supports an in depth exploration of the seed PCAP file. Here we collect some handshake packets which will be needed to establish each MMS connection. Those do not change according to the connection so they are reused as-is. In listing 3.2 we can see the first pass on the PCAP using pyshark. The handshake packets are set as Static as to not be fuzzed. The defined requests will be sent before any test case to establish a new connection.

Afterwards, we analyze all seed packets in the PCAP, as seen in listing 3.3. The seed packets are first split according to the TLV format described in the BER encoding state of the art. They are divided in Tag, Length Value triples and recursively in TLV sub-objects in constructed types. In this way, a tree representing

Listing 3.3: Seed packet analysis

```python
filter = f"(mms) && (tcp.dstport == {target[1]})"
cap = pyshark.FileCapture(pcap, display_filter=filter, include_raw=True, use_json=True)
for i, pack in enumerate(cap):
    print(f"Analyzing {i}")

    # We split the packet in a static header which we won't fuzz, and the actual MMS payload.
    header, mms = splitMMS(pack)
    mms = packetToBytes(mms)
    header = packetToBytes(header)
    head = Static(name="header"+str(i), default_value=header)

    # Here we dynamically generate a fuzzable block based on our current MMS payload.
    fuzzBlock = setupFuzzTLV(analyzeTLV(mms))

    # Here we generate a human readable name for the packet based on the kind of request made
    name = getName(pack)
    print(f"Name: {name}[{i}]")

    # Lastly we generate the boofuzz request and connect it to the session.
    curReq = Request(f"{name}[{i}]", children=(head, fuzzBlock))
    session.connect(handshake2, curReq)
    print(f"Success {i}")
cap.close()
```

each packet is created which we will then use to inform our fuzzing process. This step can be seen in listing 3.4.

Fuzzing Setup

We then setup the fuzzing process for the boofuzz framework. This is done by traversing the generated tree for each packet, and setting up each node as a block to be fuzzed in boofuzz. Each object contains a Tag field, a Length field, and either a Value field or further sub-objects in a constructed type. Each field is fuzzed in a separate way. Length fields are not fuzzed, and are instead linked to their value fields as to stay consistent. As such they will always correctly represent the length of the object, even as it changes during the fuzzing process. This avoids generating invalid Length fields, which tends to make the BER packets completely incoherent. Since a BER encoded packet is a succession of TLV objects, if a Length is incorrect a parser will start reading the successive object at the wrong position, leading to a completely random Tag and Length, quickly making the whole packet invalid. Tag and value fields are instead fuzzed using their base values as seed data. This results in an informed test case generation, which ensures that fuzz cases pass at least the first steps of validation. In fact, the generated test cases will always have a correct BER encoding, even if they don't represent a correct MMS message. This allows the test cases to always pass the first sanity checks, and reach deeper parts of the code. This setup step can be seen in listing 3.5. We extended boofuzz with two classes specific to MMS, MMSType which represents a Tag field and MMSLength which represents a Length field. MMSLength extends a Size boofuzz block, which always has a value corresponding to the length of a linked block. Our class extends this functionality by supporting a BER specific encoding, which splits the Length field in multiple bytes. Long length encoding in BER supports a Length field of at most 126 bytes In this encoding, the first byte has a flag bit set identifying the encoding, and the number of following bytes in the remaining seven bits. Further bytes contain the actual length field.

Support for this encoding is useful as it's an underused feature of BER within MMS. As such, we expect it to appear in a path that is seldom executed and that should contain many bugs. However this is also sometimes unsupported by MMS servers, and they may completely reject messages containing this encoding. Hence, we use it sparingly, in only about 1% of cases, to find a sweet spot.

We also developed a custom MMSType class, which extends the Bytes boofuzz class. This class is needed to support long form Tag encodings in BER. Long form encoded Tags have a first byte which identifies this type of encoding, and a bit set in all further bytes until the last, signaling that the Tag continues. This custom class is needed to always generate correctly formatted Tag fields. The encode function enforces some constraints of the BER encoding. In particular, it keeps the constructed bit set if the Tag was constructed in the seed traffic, keeping our message structure correct. If our fuzzer were to modify a constructed bit, then

Listing 3.4: TLV Analysis

```python
def analyzeTLV(block):
    res = []
    cur = 0
    l = len(block)

    while cur < l:
        tagLen = 1
        tag = block[cur].to_bytes(1, "big")
        # We need to handle long tags, marked by setting the last 5 bits of the packet.
        if getBits(tag[-1], 0,4) == 31:
            tagLen += 1
            tag = block[cur:cur + tagLen]
            print(f"Long tag: Len {tagLen} Tag: {tag}")
            while getBit(tag[-1], 7) == 1:
                tagLen += 1
                tag = block[cur:cur + tagLen]
                print(f"Long tag: Len {tagLen} Tag: {tag}")

        # This bit marks a constructed type, which we identify separately.
        if getBit(tag[0], 5) == 1:
            res.append((FieldType.CONSTRUCTED_TAG, tag))
        else:
            res.append((FieldType.TAG, tag))


        length = block[cur + tagLen]
        res.append((FieldType.LENGTH, length.to_bytes(1, "big")))
        value = block[cur +tagLen + 1:cur + tagLen + 1+length]
        cur += length + tagLen + 1

        # If the tag is constructed, we need to recursively analyze the value as a TLV.
        if getBit(tag[0], 5) == 1:
            res.append((FieldType.TLV, analyzeTLV(value)))
        else:
            res.append((FieldType.VALUE, value))

    return res
```

Listing 3.5: Fuzzing Setup

```python
nameCount = 0
def setupFuzzTLV(TLV):
    global nameCount
    curCount = nameCount
    children = []
    # Values in our tree are always triples of (type, value, name)
    for t, el in TLV:
    # Tags are fuzzed by our custom MMSType class, here we need to set the isConstructed flag appropriately.
        if t == FieldType.TAG:
            children += [MMSType(name="Tag"+str(nameCount), max_len=3, size=len(el), default_value=el, fuzzable=True)]
        elif t == FieldType.CONSTRUCTED_TAG:
            children += [MMSType(name="Tag"+str(nameCount), max_len=3, size=len(el), default_value=el, fuzzable=True, isConstructed=True)]

    # Length fields are fuzzed by our custom MMSLength class which extends the boofuzz Size class.
        elif t == FieldType.LENGTH:
            children += [MMSLength(name="Length"+str(nameCount), block_name="Value"+str(nameCount), length=1, max_length=126, fuzzable=False)]

    # Base Value fields are fuzzed as bytestrings.
        elif t == FieldType.VALUE:
            children += [Bytes(name="Value"+str(nameCount), max_len=0xff, default_value=el, fuzzable=True)]
            nameCount +=1
    # Constructed Value fields are recursively fuzzed as TLVs.
        elif t == FieldType.TLV:
            nameCount += 1
    # We need to update the block name of the last Length field to match our inner TLV block.
            children[-1].block_name = "TLV"+str(nameCount)
    # And then recursively generate our inner blocks.
            children += [setupFuzzTLV(el)]

    res = Block(name="TLV"+str(curCount),children=children)
    return res
```

Listing 3.6: MMSLength

```python
# Long length encoding uses multiple bytes to encode the length, with the first byte having the highest bit set.
def longLengthEncoding(value, max_length):
    res = [el for el in value]
    # We choose a random amount of bytes in which to encode our length.
    l = random.randint(1, max_length)
    # In the first byte, we set the highest bit to 1, and encode the number of bytes chosen in the remaining 7 bits.
    res[0] = l
    res[0] = setBit(res[0], 7)
    fuzzLogger.log_info("Long length: " + str(l))
    # Afterwards we fit the actual length into the remaining bytes.
    tmp = value[0].to_bytes(l, "big")
    for el in tmp:
        res.append(el)
    return bytes(res)
# This class is used to fuzz an MMS length.
class MMSLength(Size):
    def __init__(self, name=None, block_name=None, request=None, offset=0, length=4, endian="<", output_format="binary", inclusive=False, signed=False, math=None, max_length=0, *args, **kwargs):
        if(max_length == 0):
            self.max_length = length
        else:
            self.max_length = max_length
        super().__init__(name, block_name, request, offset, length, endian, output_format, inclusive, signed, math, *args, **kwargs)
    # We override the encode function to use our custom length encoding.
    def encode(self, value, mutation_context):
    # Randomly choose a long length encoding.
        value = super().encode(value, mutation_context)
        if random.random() <= 0.01:
            value = longLengthEncoding(value, self.max_length)
        return value
```

Listing 3.7: MMSType

```python
class MMSType(Bytes):
    def __init__(
        self,
        name: str = None,
        default_value: bytes = b"",
        size: int = None,
        padding: bytes = b"\x00",
        max_len: int = None,
        isConstructed: bool = False,
        *args,
        **kwargs
    ):
        # We need to know if the type is constructed, so we can set the correct bit in the tag.
        self.isConstructed = isConstructed
        super().__init__(name=name, default_value=default_value, size=size, padding=padding, max_len=max_len, *args, **kwargs)

    def encode(self,value,mutation_context):
        value = super().encode(value, mutation_context)
        if value is None:
            value = b""
            return value
        res = [el for el in value]
        # We set the constructed bit in the tag if the type is constructed, as this may have been changed by the fuzzer breaking our message structure.
        if self.isConstructed:
            res[0] = setBit(res[0],5)
        else:
            res[0] = clearBit(res[0],5)

        # If our tag is longer than one we need to mark the first byte as a long tag.
        l = len(value)
        if l > 1:
            res[0] = setBits(res[0], 0,4)
            # And set the highest bit in the remaining bytes(apart from the last) to 1.
            for i in range(1,l-1):
                res[i] = setBit(res[i], 7)
        return bytes(res)
```

Listing 3.8: Ping

```python
def ping(target:Target, fuzz_data_logger, session,
   test_case_context=None, *args, **kwargs):
    target.open()
    fuzz_data_logger.log_info("Sending handshake")
    # The sent packets are hardcoded, as they are not
   fuzzed.
    target.send(handshake1Pack)
    target.send(handshake2Pack)
    target.recv()
    fuzz_data_logger.log_info("Pinging")
    # pingPack contains an MMS identify request.
    target.send(pingPack)
    res = target.recv()
    target.close()
    if(len(res) > 0):
        fuzz_data_logger.log_pass("Ping successful")
    else:
        # A log_fail call will cause the previous test case
    to be marked as failing
        fuzz_data_logger.log_fail("Ping failed")
```

further fields would be interpreted incorrectly as nested values. This would make our whole packet invalid. Furthermore, the fuzzer keeps the length of our tag correct, by setting the appropriate continuation bits. This again is needed to keep our packet coherent. This long form encoding of Tags is actually used within the MMS specification, as such long tags are usually supported by MMS servers and supporting them ourselves is needed.

Crash Detection

Crash detection is implemented by sending a valid, simple identify MMS request to the server on a new connection. This should always be answered, and if the server doesn't accept new connections or answer a simple identify request, we assume it to have crashed. This step is ran after every test case. If it fails, the test case sent just before is assumed to be crashing, and marked as such. Using an MMS request rather than a simple ping is necessary as a ping server will often be implemented in a separate process from the main MMS server. Checking if the server is still responding after each test case is the only way to surely detect a crash. Servers will sometimes silently drop connections if they deem the requests corrupted or invalid, which is often the case during fuzzing and not to be interpreted as a crash. More advanced error detection may be possible by checking whether the received responses match the specification. However, such an in-depth analysis would certainly be complex, and require access to the specification which was not available to us.

The Fuzzing

At last, the fuzzing is handed over to boofuzz. Once launched and properly setup, boofuzz's algorithms handle the test case generation for us. Furthermore, it natively supports all other steps of fuzzing, such as saving all tried test cases in a database for further investigation. It also supports a web interface from which the fuzzing process can be comfortably monitored.

3.3.2 The MMS Client

The client is written in c, in order to use the bindings for libiec61850[18]. libiec61850 is a c library for writing server and clients, and has full support for the MMS protocol. The client supports multiple coverage options, which makes it possible to select a subset of available requests. Once launched it will connect to the target server, and make the selected requests. Some requests are influenced by the results of queries. For example, it will query the target for a list of variables and then use the found variables in successive read/write requests. Targeting existing variables is important as in IEC61850 they are identified by a human-readable name. As such it would be almost impossible for our fuzzer to randomly find valid variable names from seed traffic that didn't include this information. It supports all requests offered by libiec68150, and as such offers almost complete coverage of the protocol. This level of coverage allows our fuzzer to test little used commands, which will be more likely to contain bugs.

3.3.3 The Test Case Replayer

Boofuzz saves all ran test cases in an output sqlite database. This includes all sent and received bytes in a connection. As such our implementation of the packet replayer connects to the sqlite database, and queries it for all sent and received data in a chosen test case. It then reenacts the test case, sending all data as it was sent during fuzzing. Furthermore, it checks whether the received data matches with the test case. Otherwise, the user is notified, as this could imply that the crash was influenced by a previous test case.

All code is publicly available at `https://github.com/Sofnya/MMS_Fuzzer`.

Chapter 4

Validation

4.1 The Lab

We tested our fuzzer against a commercial PLC running an IEC61850 server, a fairly commonly used commercial product in ICS. The device is a PLC that offers support for various ICS protocols, on which a IEC61850 server license was activated. It was set up as a simple server, with no complex internal state or external peripherals. While this does not allow for complete testing, the server still supports most commands, so it remains an interesting target. We had internet access to the PLC through a VPN, and were able to restart manually it from a web interface.

4.2 Our Tests

Our tests consisted of 3 runs of our fuzzer against the target. The first run was launched with standard coverage options. However it only fuzzed the first packet it found, an InitiateRequestPDU, since fuzzing that was quickly able to crash the target. In the other runs, we manually specified a single seed packet to fuzz. The second run was launched on a Write request and the third run on a Read request.

In all of the runs, our fuzzer was able to crash the target within 5 minutes, by finding three separate bugs in the server. After our runs found the bugs they stopped as there was no way to automatically restart the remote target. This is a limitation to our tests as it prevents us from collecting information such as the number of crashes found per hour. However, we believe that being able to quickly find separate crashes is a pretty strong result on its own. In order to extrapolate some further interesting data from our tests we will now define some metrics.

4.2.1 Metrics

We collected some data on our runs in Figure 4.1.

Levenshtein distance

The Levenshtein lev(a,b) distance between two strings a and b can be defined with the following recurrence relation[8]:

$$lev(a,b) = min \begin{cases} lev(tail(a), tail(b)) & if a[0] = b[0] // copy, \\ lev(tail(a), tail(b)) + 1 & if a[0] <> b[0] // substitution, \\ lev(tail(a), b) + 1, & // deletion \\ lev(a, tail(b)) + 1 & // insertion \end{cases} \tag{4.1}$$

This is a commonly used measure of the edit distance between two strings, counting substitutions, deletions and insertions. We measured the average Levenshtein distance between the original seed value and all the produced test cases as a way to measure how much the fuzzers were mutating the seed packets. For example an average Levenshtein distance of 16 means that on average the fuzzer modified, inserted, or removed 16 bytes from the seed.

The average edit distance is an interesting metric when regarding the efficiency of a fuzzer, as it shows how much a fuzzer tends to mutate the input. Manes et al[19] defined fuzzing as ”The execution of the PUT using input(s) sampled from an input space (the “fuzz input space”) that protrudes the expected input space of the PUT”. With this definition in mind the average Levenshtein distance can be seen as a measure of how much the fuzz input space protrudes the expected input space.

The issue of how much a mutational fuzzer should mutate an input has been widely discussed, and it seems that the mutation ratio of fuzzers holds a great importance to fuzzer performance. Cha et al[4] found that the number of bugs found in various PUTs was influenced by the mutation ratio of the fuzzer, with separate PUTs often having different optimal ratios. Intuitively, inputs that are too similar to their seed are likely to be uninteresting in finding bugs, whereas inputs that differ too much tend to be rejected in the input validation.

Response rate

We define the response rate as:

$$responserate = \#responses / \#fuzzcases \tag{4.2}$$

This measures the likelihood of one of our fuzz cases receiving a response from the target. Since validation steps silently drop any requests that are incorrect, we take this to be a measure of how likely it is for one of our fuzz cases to pass the validation step.

This is especially useful in conjunction with the average distance. Intuitively, high edit distances in conjunction with high response rates would indicate that a

Seed packet fuzzed:	Initiate	Write	Read
Run Length:	188s	316s	63s
Number of fuzz cases:	130	302	78
Average Levenshtein distance:	16.75	14.16	7.33
Response Rate:	26.36%	84.44%	42.31%
Length of seed:	187	58	55
Average length of fuzz cases:	202.4	70.6	61.2
Min-Max length of fuzz cases:	184-482	57-390	55-181
Found bugs:	1	1	1

Figure 4.1: Our runs

fuzzer was able to modify its seed more skillfully. Clearly, if a packet has a high distance and was still accepted by the PUT then our fuzzer would have modified it in way that kept it recognizable, while still differing enough from the seed to be interesting.

Furthermore, we measured how much the various runs modified the length of our packets, by collecting the length of the seed and the minimum and maximum lengths of our fuzz cases. This holds some interest for us, as minimum and maximum values that differ little would show that a fuzzer wasn't modifying its seed length by much.

4.2.2 Comparison with available fuzzer

We also tested two runs of the only other open-source MMS fuzzer[13] we could find. The fuzzer is a mutational blackbox fuzzer that requires separately generated traffic in a specific format. We setup the fuzzer with seed traffic of a Write request, as this was the only requests supported by both the fuzzer and the server. It seems that the fuzzer supports a rather small subset of MMS requests. The first run lasted 35 minutes without incident, and didn't find any bugs. The second run was supposed to last for an hour, but crashed at the 44 minute mark with no bugs found. It seems that the fuzzer crashes whenever the server doesn't reply to one of its fuzz cases. This is a serious oversight as servers will often ignore incorrect requests, and a fuzzer should expect this to happen. The fact that the fuzzer only encountered this after 40 minutes of fuzzing makes us believe that the sent packets where not mutated in interesting ways. While the edit distance shows that mutations certainly took place, those probably happened in relatively safe spots. Unfortunately, we believe this limitation to be rather serious, impeding any kind of long run. Furthermore, it seems that the fuzzer has some limitations in regards to bug checking, as it runs all its test cases on a single MMS connection, and does no pings in between. After a run it outputs a pdf that contains some information on the seed packets, but no report on any found bugs, or the test

Seed packet fuzzed:	Write	Write
Run Length:	35m	44m
Number of fuzz cases:	20k	24k
Average Levenshtein distance:	7.85	7.85
Response Rate:	100%	99.99%
Length of seed:	58	58
Average length of fuzz cases:	58.9	58.9
Min-Max length of fuzz cases:	53-83	22-83
Found bugs:	0	0

Figure 4.2: Benchmark runs

cases run. Therefore a user needs to separately collect all the generated traffic, and look through it by hand to find any crashing packets.

We collected the run data in Figure 4.2. By analyzing the traffic generated it seems that the fuzzer varied the values much less than our own fuzzer on the same Write seed packet. While it had an almost perfect 100% response rate in both runs, versus our 84.44%, it had an average edit distance of 7.85, almost half of our own 14.16. While the lengths varied a fair amount, the relatively low edit distance seems to show that the fuzzer wasn't modifying its seed enough to find our same bug. In fact, being able to modify our packets by an average of 14 bytes and still achieving a 84% response rate seems to confirm the success of our informed fuzzing approach.

This clearly isn't a rigorous measure of fuzzer performance, as such a comparison holds many difficulties which we didn't have the means to solve. In particular, we were only able to test the fuzzers on one target, which may be more favorable to one fuzzer rather than another. Furthermore, we couldn't run the fuzzers for a long enough period, as finding a single bug in a short time may be just result of luck, and is much less statistically significant than finding a certain amount of bugs per hour over 24 hours or more. However, the only other available open source fuzzer showed great limitations, and we were able to find and report three bugs. As such we feel we can comfortably say that at least in this instance our fuzzer outperformed the competition.

4.2.3 Found Vulnerabilities

We found that by sending a modified Initiate-RequestPDU, Write or Read packet, one could completely crash the server, stopping it from accepting any new connections until a manual power cycle occurred.

The bugs require no authentication or privilege apart from a connection to the server to exploit, and no advanced knowledge is needed to implement it. In fact, once the triggering packets were known we were able to reliably crash the target just through packet replaying. Unfortunately no further analysis of the bugs could be made as we had no access to the code running on the target. Such bugs may escalate in gravity to full on RCE under the proper conditions. That being said we certainly have a Denial of Service vulnerability, which is still a serious threat against ICS.

While DoS vulnerabilities are sometimes considered not critical, they become so in the field of ICS. Shutting down an Industrial Control System means shutting down a physical process such as a factory, or in the worst cases, an entire power plant. This is much more severe than the temporary shutdown of a server as such an attack may disconnect thousands of people from the electrical grid.

Report Timeline

The bugs were reported to the vendor on 11/09/2023. They were confirmed on 14/09/2023, and given a preliminary CVSS score of 7.5/10, as a no authentication DoS vulnerabilities, pending further analysis. On 26/09/2023 the vendor notified us that the bugs triggered the same vulnerability, for which CVE-2023-5188 was reserved. We are currently awaiting for a patch to be developed and an advisory to be published, where the score will be finalized.

4.3 Results

The fuzzer was able to discover a serious vulnerability on a commercial target, with a preliminary 7.5 CVSS score. The vulnerability was reserved a CVE Identifier, which confirms that we found a real zero day vulnerability on a commercial system. Furthermore, in this instance our fuzzer outperformed the only other available open source fuzzer, which wasn't able to find any vulnerabilities in the target. The collected data also seems to suggest that our approach produced better test cases than a standard mutational fuzzer.

Chapter 5

Conclusions

In front of the security crisis Industrial Control Systems are facing, there is a deep need for the work of security researchers on the field. Our fuzzer offers an open-source tool which has already proved effective on one commercial PLC. We hope it will be used by developers of MMS servers to validate their products, as this would certainly improve the security of this protocol in the future. As such we feel that we successfully met the goals of this thesis.

5.1 Future Works

The fuzzer may use some improvements, mostly in its speed. While the limiting factor is certainly the time that the targets take to respond, we could take measures to reduce the number of requests per test case. At the moment, for every test case, a new connection needs to be established, which contain two requests each, and for the ping packet sent on a new connection, three different packets need to be sent. So for each test case we need to send 5 accessory requests, which is quite bad when targets are so resource limited and take so long to respond to each request.

Having access to the MMS specification would also allow us to test whether the responses matched our expectations, possibly flagging some more subtle bugs. In the future we hope to run some further tests on our fuzzer seeing that we only had one target available in our lab. Furthermore, we would love to put our fuzzer to use, in finding and reporting other vulnerabilities.

5.2 Thanks

I would like to thank Professor Baiardi and Professor Deri for their support and patience. Deep thanks also go to Martin Scheu, as without his invaluable help laboratory tests would not have been possible. I would also like to thank my friends and my bestie for their support during this year of work.

Bibliography

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software.* 2016. URL: `https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`.

[2] *boofuzz.* URL: `https://boofuzz.readthedocs.io/en/stable/index.html`.

[3] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code". In: *2012 IEEE Symposium on Security and Privacy.* 2012, pp. 380–394. DOI: `10.1109/SP.2012.31`.

[4] Sang Kil Cha, Maverick Woo, and David Brumley. "Program-Adaptive Mutational Fuzzing". In: *2015 IEEE Symposium on Security and Privacy.* 2015, pp. 725–741. DOI: `10.1109/SP.2015.50`.

[5] G. Clarke, D. Reynders, and E. Wright. *Practical Modern SCADA Protocols: DNP3, 60870.5 and Related Systems.* Engineering : instrumentation & control. Elsevier Science, 2004. ISBN: 9780750657990.

[6] *Cyber Grand Challenge.* DARPA. URL: `https://www.darpa.mil/program/cyber-grand-challenge`.

[7] *Cyber-Attack Against Ukrainian Critical Infrastructure.* URL: `https://www.cisa.gov/news-events/ics-alerts/ir-alert-h-16-056-01`.

[8] AnHai Doan, Alon Halevy, and Zachary Ives. "4 - String Matching". In: *Principles of Data Integration.* Ed. by AnHai Doan, Alon Halevy, and Zachary Ives. Boston: Morgan Kaufmann, 2012, pp. 95–119. ISBN: 978-0-12-416044-6. DOI: `https://doi.org/10.1016/B978-0-12-416044-6.00004-1`. URL: `https://www.sciencedirect.com/science/article/pii/B9780124160446000041`.

[9] *funfuzz.* Mozilla Security. URL: `https://github.com/MozillaSecurity/funfuzz`.

[10] *fuzzowski.* URL: `https://github.com/nccgroup/fuzzowski`.

[11] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: Machine learning for input fuzzing". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 50–59. DOI: 10.1109/ASE.2017.8115618.

[12] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: A Grammar-Based Open Source Fuzzer". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* A-TEST 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 45–48. ISBN: 9781450360531. DOI: 10.1145/3278186.3278193. URL: https://doi.org/10.1145/3278186.3278193.

[13] *IEC61850-MMS-Fuzzer*. URL: https://github.com/Skill3t/IEC61850-MMS-Fuzzer.

[14] Petr Ilgner and Radek Fujdiak. "Fuzzing ICS Protocols: Modbus Fuzzer Framework". In: *2022 IEEE International Carnahan Conference on Security Technology (ICCST)*. 2022, pp. 1–6. DOI: 10.1109/ICCST52959.2022.9896405.

[15] Rauli Kaksonen, Marko Laakso, and Ari Takanen. "Software Security Assessment through Specification Mutations and Fault Injection". In: *Communications and Multimedia Security Issues of the New Century: IFIP TC6 / TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01) May 21–22, 2001, Darmstadt, Germany*. Ed. by Ralf Steinmetz, Jana Dittman, and Martin Steinebach. Boston, MA: Springer US, 2001, pp. 173–183. ISBN: 978-0-387-35413-2. DOI: 10.1007/978-0-387-35413-2_16. URL: https://doi.org/10.1007/978-0-387-35413-2_16.

[16] K. Katsigiannis and D. Serpanos. "MTF-Storm: a High Performance Fuzzer for Modbus/TCP". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2018, pp. 926–931. DOI: 10.1109/ETFA.2018.8502600.

[17] Michael C. Krutwig. "Suitability of OPC UA for distributed energy monitoring". In: *Proceedings of the International Conference on Business Excellence* 13.1 (2019), pp. 399–410. DOI: doi:10.2478/picbe-2019-0035. URL: https://doi.org/10.2478/picbe-2019-0035.

[18] *libiec61850*. MZ Automation. URL: https://libiec61850.com/.

[19] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2018. DOI: 10.48550/ARXIV.1812.00140. URL: https://arxiv.org/abs/1812.00140.

[20] Jonathan Metzman, Dongge Liu, and Oliver Chang. *Fuzzing beyond memory corruption: Finding broader classes of vulnerabilities automatically.* 2022. URL: `https://security.googleblog.com/2022/09/fuzzing-beyond-memory-corruption.html`.

[21] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: `10.1145/96267.96279`. URL: `https://doi.org/10.1145/96267.96279`.

[22] *Modbus Application Protocol V1.1b3.* Modbus Organization, Inc. URL: `https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf`.

[23] *Modbus Messaging on TCP/IP Implementation Guide V1.0b.* Modbus Organization, Inc. URL: `https://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf`.

[24] *OPC UA Specification 1.04.* OPC Foundation. 2017. URL: `https://reference.opcfoundation.org/`.

[25] Lin Pei-Yi, Chia-Wei Tien, Ting-Chun Huang, and Chin-Wei Tien. "ICP-Fuzzer: proprietary communication protocol fuzzing by using machine learning and feedback strategies". In: (2021). DOI: `10.1186/s42400-021-00087-5`.

[26] *QEMU, a generic and open source machine emulator and virtualizer.* URL: `https://www.qemu.org/`.

[27] Nicholas C. Seeley. "Automation at Protection Speeds: IEC 61850 GOOSE Messaging as a Reliable, High-Speed Alternative to Serial Communications". In: (2008).

[28] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. "Saying 'Hi!' is not enough: Mining inputs for effective test generation". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2017, pp. 44–49. DOI: `10.1109/ASE.2017.8115617`.

[29] *Valgrind Home.* URL: `https://valgrind.org/`.

[30] Artemios G. Voyiatzis, Konstantinos Katsigiannis, and Stavros Koubias. "A Modbus/TCP Fuzzer for testing internetworked industrial systems". In: *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA).* 2015, pp. 1–6. DOI: `10.1109/ETFA.2015.7301400`.

[31] Michał Zalewski. *American Fuzzy Lop.* URL: `https://lcamtuf.coredump.cx/afl/`.