Advancing Protocol Fuzzing for Industrial Automation and Control Systems

Steffen Pfrang, David Meier, Michael Friedrich and Jürgen Beyerer
Fraunhofer IOSB, Fraunhoferstr. 1, 76131 Karlsruhe, Germany

Keywords: Security Testing, Fuzzing, Network Protocols, IACS, Industrial Automation and Control Systems, Vulnerabilities, Device Under Test.

Abstract: Testing for security vulnerabilities is playing an important role in the changing domain of industrial automation and control systems. These systems are increasingly connected to each other via networking technology and are faced with new cyber threats. To improve the security properties of such systems, their robustness must be ensured. Security testing frameworks aim at enabling the assurance of robustness even at the time of development and can play a key role in bringing security into the industrial domain.

Fuzzing describes a technique to discover vulnerabilities in technical systems and is best known from its usage in IT security testing. It uses randomly altered data to provoke unexpected behaviour and can be used in combination with regular unit testing. Combined with the power of fuzzing, the effectiveness of security testing frameworks can be increased.

In this work, different fuzzing tools were evaluated for their properties and then compared with the requirements for an application in the industrial domain. As no fuzzer was fully satisfying these requirements, a new fuzzer, combining the strength of different others, was designed and implemented, and then evaluated. The evaluation includes a real-world application where multiple vulnerabilities in industrial automation components could be identified.

1 INTRODUCTION

Security testing is becoming an important part of the development of industrial automation and control systems (IACS). Through new trends, like *Industrie 4.0* or the *(Industrial) Internet of Things*, these systems get increasingly connected with each other, exposing possible network vulnerabilities. Because many of these systems were not intended for this connected world, security testing was formerly neglected.

Fuzzing is successfully used in the blackhat community for finding zero-day-vulnerabilities in code (BlackHat, 2017). In contrast to penetration testing, using randomly altered data to test an interface of a test subject does not need a deep technical understanding of vulnerabilities and how to exploit them.

To provide robust and secure IACS for the future, security testing needs to be performed throughout the development life cycle, as standards like IEC 62443 are already demanding. To improve and simplify such testing, highly optimized testing platforms and frameworks are developed, that are able to carry out predefined tests. But they are still lacking the possibility of incorporating fuzzing into the test cycles, partly because the availability of fuzzers for IACS is limited. Therefore, this work aims at providing an easily applicable and effective fuzzer to enable developers and implementers of IACS to test their systems using fuzzing techniques.

This paper is organized as follows: Starting with an introduction to IACS and security testing in section 2, the ISuTest framework is introduced as host for the newly integrated fuzzing technology. In section 3, a broad overview of fuzzing characteristics is given. A comparative study of available network fuzzers results in a compilation of requirements for our developed solution. The design of the newly integrated fuzzing functionality is being described in section 4. The evaluation in section 5 is split into two parts: Starting with a functional requirement evaluation, the section continues with a practical test of the developed solution in a test bed revealing several vulnerabilities in IACS.

Pfrang S., Meier D., Friedrich M. and Beyerer J.
Advancing Protocol Fuzzing for Industrial Automation and Control Systems.
DOI: 10.5220/0006755305700580
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 570-580
ISBN: 978-989-758-282-0
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 BACKGROUND & RELATED WORK

This section gives a short introduction on the topics addressed in this paper. It briefly describes the current state of industrial automation and security testing of industrial systems. It also describes requirements for industrial security testing, as well as an example for a testing framework fulfilling these requirements. The background is concluded by presenting the basics of fuzzing in security research.

2.1 Industrial Automation and Control Systems

The modern industry landscape shows a high degree of automation enabled by industrial automation and control systems. Nowadays, they are connected via network technology to enable the exchange of control and production data to increase the efficiency of the automation processes. This increasing interconnectivity is made possible by the introduction of standard IT network technology in the industrial domain. The old, proprietary bus systems are replaced by communication protocols based on Ethernet. But, as the industrial domain still has special requirements, specialized protocols need to be used, that are able to, for example, provide real-time communication. Examples for such protocols are Profinet (PNIO) or EtherCAT. All of these established industrial Ethernet protocols have in common that they were designed with no security considerations in mind. This makes networks built on these protocols vulnerable and puts importance on the security properties of the individual devices used in these networks.

2.2 Security Testing for IACS

Security testing is used to reveal vulnerabilities of a system. For this, the Device under Test (DuT) is examined in a test bed. The DuT is provided with input by a testing system and its reaction and behavior (output) is evaluated. When an unexpected reaction is registered, further examination is used to determine which kind of problem was encountered and if this problem is caused by a vulnerability.

When security testing is performed without knowledge about the inner state or function of the DuT, this is called *black box testing*. The testing can be further classified into *positive*, providing the system with input that should lead to an expected reaction, and *negative testing*, where the system is provided with unexpected input and is expected to maintain normal functionality.

Vulnerabilities can be classified by their origin as elaborated in (Pfrang et al., 2017): Vulnerabilities that originate from the *design*, the *implementation* and from the *configuration* of a system.

Based on the IEC 62443 standard on industrial security, the ISASecure EDSA (Embedded Device Security Assurance) certification program (ISA Security Compliance Institute (ISCI), 2016) is a well-known standard in the industry and intended for the testing of IACS. It includes multiple areas of requirements, including secure development (SDA-E), functional security (FSA-E) and device robustness testing (ERT). For the device robustness testing, the Communication Robustness Testing (CRT) defines test cases for multiple network protocols.

The special requirements and conditions of the industrial domain, as described in the previous section, need to be thoroughly considered in the security testing of IACS, as well as requirements originating from security standards like the ISASecure EDSA. In (Pfrang et al., 2017), we gave an overview of multiple security testing frameworks intended for the application in the IACS domain, formulated key requirements and, finally, introduced ISuTest, a new and modular security testing framework for IACS that is intended to fulfill these requirements.

2.2.1 Industrial Security Testing Requirements

Following a review of existing security testing systems and test specification from standards like the IEC 62443, we formulated key requirements for an industrial security testing system. Such a system needs to be able to test arbitrary IACS for vulnerabilities (I), independent from type, manufacturer or technologies used in the IACS. The system also needs to be modular and extensible (II) to account for future application within these evolving technologies. The modularity includes the possibility to test different network protocols in different layers (IIa) and to test the different possibilities of input and output of data an IACS has (IIb). The results of such tests need to be evaluated with suitable methods (IIc). Another important requirement is a precise way to describe security tests through declarative test scripts (IIIa) that allow for an arbitrary combination of input and output handling (IIIb), as well the evaluation of the result against an expected behavior (IIIc). As there is the necessity to execute a high number of individual tests, the system needs to support the fully automated execution of tests (IIId) that also need to be reproducible (IIIe).

2.2.2 ISuTest Framework

The *ISuTest* framework is a novel security testing framework for IACS that is intended to fullfill the requirements presented in the previous section. ISuTest is a highly modular security testing framework that is adaptable to different test environments and scenarios. It is able to incorporate different technologies and therefore able to test systems of very different forms.

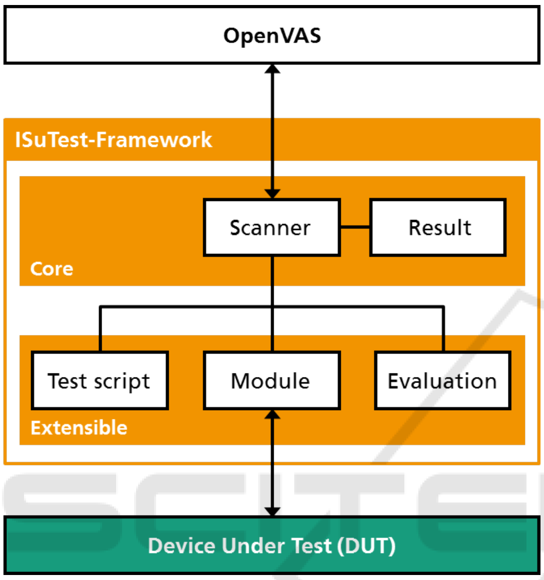

Figure 1: Architecture of the ISuTest framework.

As shown in figure 1, the core of the framework is formed by the scanner, handling the whole testing logic, and the result component that is responsible for the creation of final test results. The extensible part of the framework is consisting of *test scripts*, *modules* and *evaluations*. Test scripts specify individual test cases, including preparation, execution and result evaluation. The test scripts rely on modules to test systems and are executed by the scanner. These modules provide the functionality to interact with a DuT, independent of the used communication technology. For the creation of results, different evaluators can then interpret the reaction of the DuT to detect malfunctions and vulnerabilities. The framework is compatible with the OpenVAS Scanner Protocol and can therefore be used with this vulnerability scanner.

2.3 Fuzzing

Fuzzing characterizes a negative testing methodology driven by random alteration of inputs. The test subject is tested by providing predominantly invalid, unexpected input to it. This approach is highly automatable and offers a high chance of finding unknown vulnerabilities since it explores unexpected input. But fuzzing can only cover a small subset of all possible input in a feasible amount of time. Consequently, one of the main goals is to select the fuzzed data as efficiently as possible, so that the fuzzed data covers a subset with as high probability for detecting vulnerabilities.

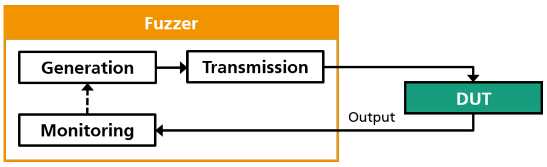

Figure 2: The phases of a fuzzing process.

The fuzzing process, as shown in figure 2, is divided into three phases: Generation of data, transmission of data and monitoring the DuT. Data is created and altered by the fuzzer. The alteration of the data should ensure that the data is still processed by the test subject, i.e. the alterations should not destroy the format completely. Altered data is transmitted to the test subject via arbitrary interfaces. Whilst being tested, the subject is monitored in order to detect anomalies in its behaviour. Detection of crashes, unavailable services or unexpected behaviour is an evidence for a successful fuzzing test.

3 ANALYSIS

In this section, we provide a terminology for the classification of properties of fuzzers. We then conduct a comparative study of available fuzzers and motivate why there is the need of a new fuzzer for IACS.

3.1 Classification of Fuzzers

In order to compare different approaches and types of fuzzers, it is useful to describe each of them using orthogonal categories. In the following, we present four categories to characterize fuzzers which are derived from (Sutton et al., 2007) and (McNally et al., 2012): *Input crafting*, *operation mode*, *extensibility* and *organizational features*.

3.1.1 Input Crafting

Crafting of fuzzed input data is realizable in many different ways. Depending on the scenario, data is either *generated* based on knowledge about the data format or *mutated* during transmission. Additionally, a fuzzer either generates the input data whilst testing the

DuT (online) or precomputes the fuzzed data (offline). The main reasons for offline generation of data are timing requirements of real time components used in IACS.

3.1.2 Operation Mode

In order to fuzz stateful protocols efficiently, the fuzzer must track the state of the protocol being tested. *Rearranging* fuzzers are sometimes referred to as behavioural fuzzers. They do not fuzz data but data sequences which can trigger a different kind of faults in the DuT compared to the data fuzzing. Rearranging fuzzing is only relevant for stateful protocols because stateful protocols can have a wrong order of valid data. *Adaptive* fuzzers use the reaction of the DuT for adjusting their input in order to increase the probability of provoking an anomaly. If fuzzers are equipped with special means to monitor IACS outputs other then Ethernet-based ones, they provide *IACS monitoring*.

3.1.3 Extensibility

This category deals with the extensibility of the fuzzer. If a fuzzer is *learning*, it is able to extend its knowledge of protocols and data structures automatically by observing a given communication record. In contrast, other fuzzers rely on *Expert knowledge*. Some fuzzers are tailored to a set of specific protocols. Others offer the opportunity to *extend the protocol* knowledge. From a technical point of view, it is important on which network layer of the *ISO/OSI* model (Zimmermann, 1980) the fuzzers are based. There are three common options for network fuzzers: Ethernet (layer 2), IP (layer 3) or TCP/UDP (layer 4). Particularly, if protocols need to be extended or learned, an Ethernet-based protocol cannot be implemented in an IP-based fuzzer.

3.1.4 Organizational Features

Some organizational features are listed in this category. They are particularly important if an existing solution should be extended. So here is listed whether the *source code* of the fuzzer is available online and under which *license* it can be used. Additionally, if known, the programming *language* and the time of the *last update* of the fuzzer are registered.

3.2 Comparative Study of Fuzzers

Based on the classification provided in section 3.1, we performed a comparative study on available fuzzers. The results are presented in table 1 and described in this section. The selection criteria for the studied fuzzers were the following: All fuzzers are either open source or were described comprehensively in scientific papers. Additionally, all fuzzers provide the possibility to create fuzzed network packets.

Two learning fuzzers, *Pulsar* (Gascon et al., 2015) and *AutoFuzz* (Gorbunov and Rosenbloom, 2010), use machine learning algorithms to infer the finite automaton of a stateful packet and heavily depend on the completeness of provided data. While AutoFuzz is not maintained since 2010, Pulsar got recent updates. Nevertheless, Pulsar crashed in our tests when using it with other data then with the included examples. The maintainers confirmed this behaviour.

The General Purpose Fuzzer (*GPF*) learns protocol formats from recorded traffic and displays the protocol format in a human-readable way. Based on the graphical representations, the fuzzer is configurable to alter the displayed data.

Lzfuzz (Bratus et al., 2008) uses the General Purpose Fuzzer (*GPF*) as its fuzzing engine. It uses a compression algorithm to learn data structures and fuzzes data mutatively as a man in the middle when the data is transmitted. Lzfuzz can monitor the DuT, but does not offer generative and rearranging fuzzing.

An example for a commercial fuzzing framework is *Peach* (Peach Fuzzer Website, 2016). It is providing the ability to define data formats and fuzzing scenarios in a xml format. It is available in an open-source community edition with a reduced subset of functions. The code of Peach is written in C#.

Another unmaintained fuzzing framework using data and protocol definitions based on xml is *Snooze* (Banks et al., 2006), which is a research prototype developed in 2006.

A popular fuzzing framework, which has been extended by several research projects, is *Sulley* (Amini and Portnoy, 2007). It has not been maintained since the year 2007 but extended in various projects to adapt it to certain test scenarios. Sulley offers stateful fuzzing of network protocols, using block-based definitions to craft data packets and applying an inexpandable set of heuristics to alter said data. The Sulley framework only offers TCP and UDP as transport protocols, whereas IACS often use Ethernet-based protocols. Monitoring of the DuT is supported by Sulley with automated logging of network traffic and storing it. After every fuzzing step, Sulley resets the connection and iterates through its monitors before continuing fuzzing the DuT in order to ensure a well-defined state of the DuT before sending fuzzed data. This feature is not configurable and causes a low testing rate of one fuzzed data packet per second. Sulley uses an outdated data format to illustrate the proto-

Table 1: Network fuzzer comparison.

	Pulsar	*Autofuzz*	*GPF*	*lzfuzz*	*Peach*	*Snooze*	*Sulley*	*ProFuzz*	*MTF*	*PropFuzz*	**Ideal**
Input crafting											
Generative	✓		✓		✓	✓	✓	✓	✓		✓
Mutative		✓	✓	✓						✓	
Online	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Offline			✓		✓					✓	✓
Operation mode											
Stateful		✓			✓	✓	✓		✓		✓
Rearranging	✓				✓	✓					✓
Adaptive		✓		✓					✓		✓
IACS monitoring										✓	✓
Extensibility											
Learning	✓	✓	✓	✓					✓	✓	(✓)
Expert Knowl.			✓		✓		✓		✓		✓
Protocol extens.	✓[2]	✓	✓		✓		✓			✓	✓
ISO/OSI layer	3	4	3	3	2	4	4	2	4	4	2
Organizational											
Source code	✓	✓	✓[1]	–	✓	–	✓	✓	–	–	✓
License	BSD	n/a	GPL	–	MIT	–	GPL	GPL	–	–	Open
Language	Pyth./R	Java	Pyth.	n/a	C#	Pyth.	Pyth.	Pyth.	Pyth.	Pyth.	–
Last updates	2017[2]	2010	2007	2008	2016[3]	2006	2015[4]	2012	2016	2017	–

[1] via archive.org, [2] known bugs, see text, [3] Community Edition, [4] officially unmaintained

col state machine which cannot be opened easily, impeding development of individual fuzzers within the framework.

ProFuzz (HS Augsburg, 2012) and *Modbus/TCP Fuzzer (MTF)* (Voyiatzis et al., 2015) are fuzzers distinctly developed for fuzzing automation protocols. They are targeted to one specific protocol, Profinet and respectively Modbus, and are not extendible for usage with other data formats.

PropFuzz (Niedermaier et al., 2017) is a framework using statistical computation to analyze the structure of proprietary protocols during the DuT's communication with the vendor's integrated development environment (IDE). It is able to fuzz such communication, which has to be based on TCP. This generates a limited range of fuzzed data since the input is limited to only the valid communication with the IDE.

Concluding, none of the considered fuzzers and fuzzing frameworks shown in table 1 fit our scenario perfectly and include the properties required for providing an efficient and extendable fuzzing framework.

3.3 IACS Fuzzer Requirements

Following the comparative study of different fuzzers presented in the previous section and based on the requirements for security testing of IACS described in section 2.2.1, the following requirements for an ideal fuzzer of IACS can be formulated.

(A) The fuzzer must be able to use existing protocol definitions and therefore be *generative*. This enables the integration of expert knowledge into the fuzzing process for enabling the test of arbitrary systems based on varying technologies.

(B) The fuzzer must be able to craft the desired input *online* and *offline*. Online input crafting is needed to be able to exercise adaptive fuzzing, whereas offline crafting might be needed to prepare input to achieve high transmission rates.

(C) The fuzzer must be able to be aware of the *state* of the network protocols. There are many stateful industrial protocols and the fuzzer needs these states to be able to achieve a high coverage of possible device conditions.

(D) The fuzzer must be able to use knowledge of the protocol procedures to create invalid protocol sequences, enabling fuzzing by *rearranging* (behavioral). This increases the effectiveness in the testing of network protocol implementations.

(E) The fuzzer must be *adaptive* in its interaction with the examined DuT. This enables the fuzzer to react on the responses of the device and is needed to reach certain states in stateful protocols.

(F) The fuzzer must be able to *monitor* the DuT as thoroughly as possible, i.e. on all available interfaces. Especially IACS can exhibit multiple interfaces of different nature, like network or electrical I/O.

(G) The fuzzer must be *extensible* to enable the adaption to different test scenarios and different network technologies. The extensibility includes simple ways for adding new protocols by *expert knowledge*, and optionally by automated *learning*. *Source code availability* and suitable, open *licensing* is also part of the extensibility properties of a fuzzer.

(H) The fuzzer must be able to craft network packets on the *Ethernet-Layer* (ISO/OSI layer 2), to enable the fuzzing of industrial Ethernet protocols that are often used in the IACS domain.

(I) The fuzzer shall create *reproducible* test cases and data. This is important for the availability of repeated testing in order to localize found vulnerabilities and test mitigations.

An *ideal* fuzzer for IACS needs to meet all these requirements. Its properties, resulting from the requirements, are also depicted in table 1. The results in the table also show, that the examined existing fuzzers cover only a subset of the presented requirements. In order to offer an improved and extendable solution, this work aims at providing a fuzzer more suitable to the IACS domain.

4 DESIGN

Based on the identified requirements, we present in this section a generic fuzzer which meets the requirements as well as allows the exchanging of its components in order to extend or replace functionality of the framework. Integrated in the ISuTest framework, the fuzzing framework is composed of different ISuTest modules: data generation, protocols, data transmission and monitoring modules. The modules and their interaction are shown in figure 3.

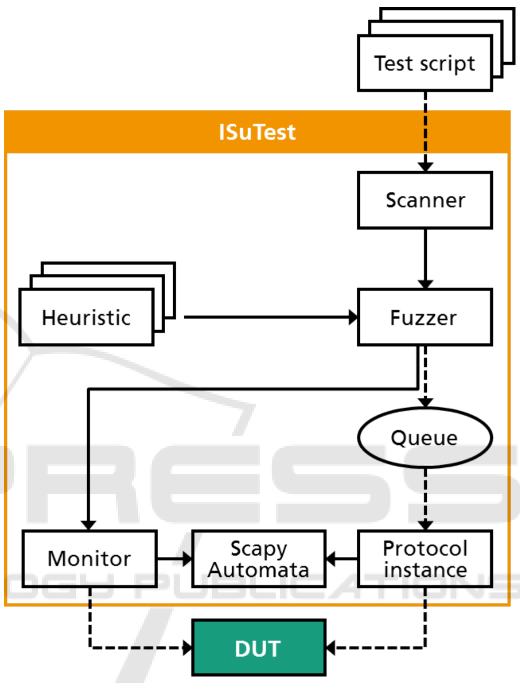

Figure 3: Architecture of the fuzzing framework.

The ISuTest *scanner* interprets a test script which defines a fuzzing test. It initializes the *fuzzer* accordingly to the specified testing functionality. On the one hand, the fuzzer makes use of the *heuristics* to generate the requested input data and puts resulting fuzzing instructions into the *queue*. On the other hand, the fuzzer instructs the *monitor*. Once the fuzzing test is being started, the *protocol instance* pulls fuzzing instructions from the queue and crafts network packets by using *scapy/automata* (Biondi, 2010) definitions. Concurrently, the protocol sends the packets to the DuT while the monitor observes the DuT's reactions.

All tests conducted with the fuzzing framework must be reproduceable. Therefore, a pseudo random number generator is used, which provides deterministic values depending on a provided seed.

In the next subsections, the design is being explained following the phases of a fuzzing process.

4.1 Data Generation

The packet formats for the protocols, which are to be fuzzed, are encoded in packet definitions. Modelling of stateful protocols is done with a finite automaton. The automaton must enable state transitions without sending and receiving the packets which are normally necessary for state transitions in order to realize rearranging fuzzing. For generative fuzzing, the state transitions must enforce sending and receiving data according to the protocol. This enables a targeted fuzzing of protocol states which have to be reached before fuzzed data is sent.

The data generation itself is twofold: Fuzzing instructions are generated in the fuzzer module with the help of heuristics, and alteration of valid data in the protocol module within the finite automaton. The fuzzing instructions contain information which field is to be fuzzed and in which manner, i.e. it contains the value the given field is to be applied by the protocol module. The protocol module uses a packet definition to craft a data packet containing the altered value. Lose coupling between fuzzer and protocol with well defined interfaces in-between makes the framework highly flexible. The fuzzer is also logging seed values for every generation of pseudo-random values, enabling to recreation of the same values again.

4.1.1 Heuristics

The fuzzer component uses *heuristics* to determine how a field of a given data type is to be fuzzed, whereas the protocol module crafts the altered data. The need for using heuristics results from the impossibility to test every possible value in a reasonable time.

Therefore, promising values have to be chosen. The used heuristics rely on experience from previous projects, e.g. Sulley, and are extendable. Expert knowledge should additionally be used to tailor the heuristics to a given test scenario in order to maximize the chance for finding a vulnerability.

The fuzzing framework currently makes use of four different heuristics. The *Integer heuristic* produces numbers like 0, the minimum and the maximum value as well as boundary cases at power of 2 minus 1. The *String heuristic* provides, for example, escape sequences, terminal commands and SQL statements. The *MAC heuristic* creates randomized Mac addresses while the *UUID heuristic* chooses randomized Universally Unique Identifiers.

4.2 Transmission of Data

The finite state machine, modeling the protocol to be tested, alters fields of given data packets crafted from the packet definition according to the instructions received from the fuzzer. Instructions are transmitted between the fuzzer and the protocol instance via a queue. This enables the fuzzer and the protocol modules to be separate entities within the ISuTest framework. The execution of the two entities is loosely decoupled to enable running them asynchronously and increase overall performance of the testing framework. Furthermore, the fuzzer and protocol modules are reusable for other kinds of security tests within the ISuTest framework.

The communication between the fuzzing framework and the DuT is realized by a communication module which provides transparent connectivity on a certain layer. For example, a communication module providing a TCP connection handles the establishment and maintaining of said connection. Based on that connection layer, the protocol modules only need to manage data based on the connection and forward it to the communication module.

4.3 Monitoring of the DuT

White box fuzzers have access to the internal state of the subject under test, e.g. by using a debugger to observe the reactions of a program to fuzzed input. Industrial devices however do not offer the possibility to observe internal reactions to a certain input. Instead, the state of the DuT has to be inferred by monitoring interfaces which it provides. This includes the Ethernet interface over which it communicates with the fuzzing framework, as well as other data buses and I/O.

By running a known program on the DuT, the expected output of the DuT can be compared with the actual output in order to detect vulnerabilities of the DuT. Monitoring of analogue and digital I/O requires appropriate measurement and processing of the signals in order to be logged and used in the fuzzer. In particular, the rate of false positives arising from measurement inaccuracies must be kept to a minimum.

4.4 Fuzzing Procedure

The fuzzer operates in two modes: synchronous or asynchronous. When operating synchronously, the fuzzer checks the state of the DuT after each instruction, respectively after each sent data packet. A more time efficient way of monitoring the DuT's health is monitoring it asynchronously, after a bulk of data packets has been sent. Since the fuzzer and the protocol is decoupled, the fuzzer requests the health information about the DuT instead of being notified and can only map anomalies to messages which have been sent since the last check. A tradeoff between efficiency and accuracy is found here in order to increase the number of performed tests to a reasonable number.

After every monitoring check with a positive result, a well-defined state must be reached by the DuT. Resetting the DuT enables the fuzzing to be targeted to a certain state. The efficiency of fuzz testing is increased by selectively fuzzing states of a stateful protocol instead of sending fuzzed data without consideration of the DuTs state. Without considering the protocol state machine, a deep exploration of said state machine is not feasible since most of the fuzzed data resets the DuT into an initial state.

5 EVALUATION

In this evaluation, the properties of the presented fuzzer will be compared to the requirements, as presented in section 3.3. For a practical evaluation, the fuzzer was also used to test multiple IACS, which will also be described in the following sections, including the test results.

5.1 Requirement Evaluation

The presented fuzzer design uses protocol descriptions, including structure of individual packets and logic of the protocol sequences (section 4), and is able to generativly create the test data, satisfying requirement (A). The test data can be generate online, during the the test execution, or offline, as demanded by requirement (B).

The design enables the fuzzing of stateful protocols as well as behavioral fuzzing by rearranging packet sequences (section 4.1) and thus satisfies requirements (C) and (D). Through the usage of Scapy/Automata, as well as modules from the ISuTest framework, the responses of the DuT are available to the fuzzer (section 4.2) and enable the adaption to these reactions, fulfilling requirement (E).

As described in section 4.3, the fuzzer is able to monitor a wide range of the output possibilities of the DuT, ranging from network responses to digital and analog I/O, satisfying requirement (F). This form of protocol description format also enables the fuzzer to adapt to new test scenarios, including additional protocols, as demanded in requirement (G).

The fuzzer is based on open source technology, enabling open adaption to new technologies (requirement (G)). Based on Scapy/Automata, the fuzzer is able to craft packets from the Ethernet layer upwards and enabling the implementation of typical automation protocols (requirement (H)). Requirement (I) states the reproducibility of test cases, which the presented fuzzer can satisfy by logging configuration and seed values, as described in section 4.1.

5.2 Test Bed Evaluation

The fuzzer has been implemented in Python 3 as ISuTest modules. It makes use of Scapy definitions for network and automation protocols. States of automation protocols, in particular PNIO, have been implemented using Automata in Scapy.

Four PNIO bus couplers were selected as DuTs to perform the evaluation. Bus couplers are used to operate and control remote periphery in terms of digital and analog signals. In PNIO terminology, the bus couplers act as PNIO devices which are managed by a PNIO controller.

5.2.1 Testing Methodology

The test bed in which the evaluation took place, following the ISuTest specification, consists of the devices depicted in figure 4. Depicted in the center, the industrial network *Switch* is providing communication.

Above the switch, the infrastructure components are shown: The *Test device*, a Linux-based PC, runs the ISuTest framework. A programmable logic controller (PLC) acts as *IO device* which monitors the digital outputs of the DuTs. Finally, the *Remote controllable power supply (RCPS)* is used to reset the DuTs automatically. Below the switch, there are the four bus couplers labelled as *DuT 1 to 4*.

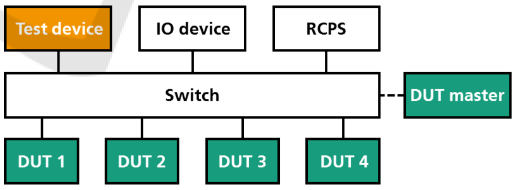

Figure 4: Test bed evaluation setup.

Beside the switch, the *DuT master* is depicted with a dashed connection. Initially, the DuT master is used as PNIO controller which runs the automation process and controls the four DuTs as PNIO devices. The automation process emits a reference signal on one digital output of every DuT. It consists of three repeating cycles in which the first one is a 1 and the next two ones are a 0 each. This cycle is based on the EDSA CRT definition, as mentioned in section 2.

In the setup phase, the Test device observes the communication setup between the DuT master and the DuTs. During the testing phase, the Test device spoofs both the MAC and the IP address of the DuT master while the DuT master gets disconnected physically from the switch. The spoofing enables the Test device to act as the legitimate PNIO controller for the DuTs. This is important because it allows for setting up a valid communication between the Test device and the DuTs, which ensures that the DuTs process the fuzzed packets as intended.

5.2.2 Essential Services Monitoring

ISuTest monitors the functionality of the DuTs while those are being tested. In table 2, the essential services of the DuTs 1 to 4 that are taken into account are listed. A check mark represents the fact whether the DuT supports the respective service.

Table 2: Essential services provided by the DuTs and checked by ISuTest.

Service	**DuT 1**	**DuT 2**	**DuT 3**	**DuT 4**
Ping	✓	✓	✓	✓
Web server	✓	–	✓	–
Digital Output	✓	✓	✓	✓
CM Connect	✓	✓	✓	✓

Ping tests are performed by the ICMP module from ISuTest. It sends ICMP echo requests and waits for echo replies. Checking the availability of the web-based management interface is performed by ISuTest's HTTP module. It requests the home page of the DuT and checks the HTTP status code.

In order to check for the digital output, the reference signal is being used. The digital output pin of the DuT is connected to a digital input pin of the IO device. The Digital IO module checks for both the existence of the test signal and its jitter.

Checking for the PNIO-CM Connect functionality is the most complex part, because it consists of more than one single request and response packet but includes different protocols. It makes use of the PNIO module being described in the following and checks whether the real-time communication lasts at least three seconds.

5.2.3 Test Execution

The fuzzing tests have been performed on each DuT for the following four protocols: PNIO-DCP, PNIO-CM and PNIO-RT, as an example for industrial automation protocols, and IPv4, as an example for an internet protocol.

Table 3 represents an example for a PNIO-DCP network packet requesting the identification of a PNIO device with the respective nameOfStation as the identifier. Starting with the name and the length (in bit) of a field, the following columns represent the applied heuristics, the number of iterations performed while testing and the resulting coverage of the search area in percent.

Table 3: Example of the fuzzing configuration of a DCP message requesting the identification of a PNIO device (IdentReq). Length in bit, coverage in %.

Name	**Len.**	**Heur.**	**Iter.**	**Cover.**
FrameID	16	Integer	2000	3,05
ServiceID	8	Integer	256	100
Service-Type	8	Integer	256	100
xid	32	Integer	2000	$\ll 0{,}1$
ResponseDel.	16	Integer	2000	3,05
DCPDataLen.	16	Integer	2000	3,05
blockType	16	Integer	2000	3,05
dcpBlockLen.	16	Integer	2000	3,05
blockInfo	16	Integer	2000	3,05
nameOfStat.	1152	String	2500	$\ll 0{,}1$
		Sum	17012	

As one can see, the bigger the fields, the smaller the coverage of the overall value space. Since there are defined multiple block types for getting and setting configuration options, the number of fuzzing iterations for the PNIO-DCP protocol multiplies.

Table 4: Excerpt of the fuzzing configuration of a PNIO-CM Connect request message. Length in bit.

Name	**Len.**	**Heur.**	**Iter.**
args_max	32	Integer	2000
args_length	32	Integer	2000
max_count	32	Integer	2000
offset	32	Integer	2000
actual_count	32	Integer	2000
block_type	16	Integer	2000
block_length	16	Integer	2000
block_version_high	8	Integer	256
block_version_low	8	Integer	256
ARType	16	Integer	2000
ARUUID	128	UUID	2000
SessionKey	16	Integer	2000
CMInitiatorMacAdd	48	MAC	2000
CMInitiatorObjectUUID	128	UUID	2000
...			

Table 4 represents the fuzzing of PNIO-CM as an example for a stateful network protocol. The depicted network packet shows an excerpt of a Connect request from the DuT master to the DuT. The PNIO-CM-related part of the packet has 80 distinctive fields with a total length of 1,232 bit. In order to fuzz all possible configurations, one had to fuzz $2^{1,232}$ packets ($\approx 7.4 \cdot 10^{370}$). Using the heuristics, this amount of test cases can be reduced to 106,000, which can be conducted in a reasonable amount of time.

5.2.4 Vulnerability Discoveries

During the execution of the fuzzing tests, we discovered multiple vulnerabilities in each of the DuTs. On the one hand, there were expected vulnerabilities which are present in the DuTs because they originate from weaknesses in the design of PNIO. A prominent example for this, which is also mentioned in (Pfrang and Meier, 2017), is a DCP Set request which changes the name of the PNIO device (nameOfStation). Since there is limited use in testing if the communication breaks if the device is renamed to *a* or *b* or *c*, we limited those test cases.

On the other hand, we discovered vulnerabilities which are located in the implementation. As an example, we will describe a vulnerability in the PNIO-CM protocol which two DuTs have in common. When setting up an application relationship, both communication partners configure the frequency in which they exchange real time data messages. The respective parameter is called *SendClockFactor* and defined in a 2 byte field.

When setting this parameter to the values of 8, 9, 10, 11 or 65524, two DuTs crash and reboot with all status LEDs flashing shortly. Usually, devices loosing a connection have to send alarm frames. This does not work either. The monitoring also confirms that neither are the devices pingable nor can their web server be reached. Their digital outputs stop working and the a CM connect is not possible for half a minute.

In real factories, it is common to fix failures of automation equipment by power cycling the respective device. But this vulnerability cannot be fixed by this means, because the device has lost its device name which is indispensable in a PNIO communication. Probably those values cause a memory corruption. In order to cure this behaviour, one has to reset the device name using the PNIO-DCP protocol.

6 CONCLUSION

Security testing for IACS is crucial. By detecting vulnerabilities and fixing them before they can be abused by attackers, manufacturers can improve the overall security of automation systems. The ISuTest framework enables developers to perform security tests on their IACS automatically. Fuzzing is another, interesting technique to discover vulnerabilities. By adding this functionality to the ISuTest framework, its effectiveness in finding vulnerabilities could be improved extraordinarily.

Within this work, first, available fuzzers and fuzzing frameworks have been characterized and compared. Then, requirements for effective fuzzing of IACS, based on existing security testing requirements, were developed. By comparing the studied fuzzers to these requirements, it was shown that none of the fuzzers fulfilled them completely. As a result, a new fuzzer was developed and described in this paper, which can be integrated into the ISuTest framework as well as meet the requirements for an ideal IACS fuzzer.

During the evaluation, it was shown that the new fuzzer is able to fulfill these requirements. The fuzzer was evaluated further by setting up a test bed and conducting fuzzing test against multiple Profinet devices. During this evaluation, several vulnerabilities could be discovered. The fact that two devices suffered from the same vulnerability stresses the importance of security tests not only for developers but also for integrators of IACS. As shown, the fuzzing framework proved its usefulness successfully.

6.1 Future Work

During this work, it became clear that future work should focus on the precise description and classification of the vulnerabilities. This is important to be able to clearly classify detected vulnerabilities and report them to the manufacturer of the device. This includes formal methods taking into account the special requirements for IACS.

Further work needs also to be put into the definition of new protocols following the presented fuzzer definition. As the IACS domain consists of highly heterogeneous technology, it is important to cover as many technologies and protocols as possible.

REFERENCES

Amini, P. and Portnoy, A. (2007). Sulley: Fuzzing framework. http://www.fuzzing.org/wp-content/SulleyManual.pdf. [Online; accessed 2016-10-23].

Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. (2006). SNOOZE: toward a stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*, pages 343–358. Springer.

Biondi, P. (2010). Scapy documentation. http://www.secdev.org/projects/scapy/doc/. [Online; accessed 2017-11-07].

BlackHat (2017). Fuzzing for vulnerabilities. https://www.blackhat.com/us-17/training/fuzzing-for-vulnerabilities.html. [Online; accessed 2017-11-07].

Bratus, S., Hansen, A., and Shubina, A. (2008). LZfuzz: a fast compression-based fuzzer for poorly documented protocols. Technical Report TR2008-634, Dartmouth College.

Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., and Rieck, K. (2015). Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer.

Gorbunov, S. and Rosenbloom, A. (2010). Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239.

HS Augsburg (2012). ProFuzz. https://github.com/HSASec/ProFuzz. [Online; accessed 2017-11-07].

ISA Security Compliance Institute (ISCI) (2016). Embedded Device Security Assurance (EDSA) - version 2.0.0. http://www.isasecure.org/en-US/Certification/IEC-62443-EDSA-Certification. [Online; accessed 2017-05-18].

McNally, R., Yiu, K., Grove, D., and Gerhardy, D. (2012). Fuzzing: the state of the art. http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA558209. [Online; accessed 2016-09-30].

Niedermaier, M., Fischer, F., and von Bodisco, A. (2017). Propfuzz-an it-security fuzzing framework for proprietary ics protocols. *International Conference on Applied Electronics (AE)*.

Peach Fuzzer Website (2016). Peach fuzzer: Discover unknown vulnerabilities. http://www.peachfuzzer.com/. [Online; accessed 2016-11-01].

Pfrang, S. and Meier, D. (2017). On the Detection of Replay Attacks in Industrial Automation Networks Operated with Profinet IO. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, pages 683–693.

Pfrang, S., Meier, D., and Kautz, V. (2017). Towards a modular security testing framework for industrial automation and control systems: Isutest. In *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017*.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.

Voyiatzis, A. G., Katsigiannis, K., and Koubias, S. (2015). A modbus/TCP fuzzer for testing internetworked industrial systems. In *20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–6. IEEE.

Zimmermann, H. (1980). Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432.