OIOFuzz: A Guided Model-based Blackbox Fuzzer for OIORASP Schematron Validation

Project Report
cs-23-ds-10-06

Aalborg University
Computer Science

Copyright © Aalborg University 2023

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
http://www.aau.dk

Title:
OIOFuzz: A Guided Model-based Blackbox Fuzzer for OIORASP Schematron Validation

Theme:
Fuzzing OIORASP

Project Period:
Sprint Semester 2023

Project Group:
cs-23-ds-10-06

Participant(s):
Emil Fulei Lykke Aagreen
Frederik Arnfeldt Jensen

Supervisor(s):
Danny Bøgsted Poulsen
René Rydhof Hansen

Copies: 1

Page Numbers: 61

Date of Completion:
June 16, 2023

Abstract:

In this project we explored the potential for fuzzing OIORASP. OIORASP is an protocol for exchange of e-business documents and is an integral part of the Danish IT Infrastructure. The protocol uses the OIOUBL document standard for the documents sent. Fuzzing is an automatic test method where unexpected inputs are constructed and passed to the target program to observe if it trigger unexpected behavior. The target is narrowed in to the Schematron validation of the documents. We made OIOFuzz which is a proof-of-concept implementation of a guided model-based blackbox fuzzer targeting OIORASP Schematron validation. OIOFuzz managed to find an error in the Schematron validation. Therefore we concluded that it is functional, but it also has room for improvement.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

Preface **v**

1 Introduction **3**

2 OIORASP **6**
2.1 E-business Standards 7
2.1.1 OpenPeppol 7
2.1.2 NemHandel 8
2.2 OIORASP 8
2.2.1 Protocol Overview 8
2.2.2 Protocol Details 9
2.3 OIOUBL - Universal Business Language 12
2.3.1 OIOUBL Properties 13
2.3.2 Validation of OIOUBL Documents 14
2.4 Document Processing in the OIORASP Library 15

3 Fuzzing Theory **17**
3.1 Fuzzing Approaches 18
3.1.1 Blackbox Fuzzing 18
3.1.2 Whitebox Fuzzing 19
3.1.3 Greybox Fuzzing 20
3.2 Fuzzing Concepts 20
3.2.1 Mutation- and Generation-based Fuzzing 21
3.2.2 Smart and Dumb Fuzzer 22
3.2.3 Code Coverage 22
3.2.4 Power Scheduler 22
3.3 American Fuzzy Lop 22

4 Implementing OIOFuzz **24**
4.1 Delimiting the OIORASP Fuzzing Target 25
4.2 Fuzzing Approach 26
4.3 OIORASP Library Setup 26

4.4 OIOUBL documents . . . 27
4.5 Fuzzer Structure . . . 29
4.6 Components . . . 30
4.6.1 Invoice Model . . . 30
4.6.2 Parser . . . 32
4.6.3 Fuzzer . . . 32
4.6.4 Mutator . . . 33
4.6.5 Runner . . . 36
4.7 Classification of OIOFuzz . . . 36

5 Fuzzing OIORASP 37
5.1 Initial Exploration . . . 37
5.2 Running OIOFuzz . . . 38
5.2.1 Observations From Early Iterations . . . 38
5.2.2 Experiment Setup . . . 39
5.2.3 Different Variations . . . 40
5.3 Schematron Error . . . 41

6 Related Work 45

7 Discussion 47
7.1 Instrumentation of the ClientExample . . . 47
7.2 C# Fuzzer . . . 48
7.3 Schematron Guided Mutations . . . 48
7.4 Generation-based Fuzzer . . . 49

8 Conclusion 51

9 Future Work 52

Bibliography 54

A OIOUBL Invoice Document 57

Preface

This master thesis is written by the group cs-23-ds-10-06 in the spring semester of 2023 at Aalborg University.

Figures, listings, etc., are numbered by order of appearance.

The Harvard reference style are used for citations. All citations is in the Bibliography at the end of the report.

We would like to thank our two supervisors Danny Bøgsted Poulsen and René Rydhof Hansen for their guidance on our project.

Aalborg University, June 16, 2023

Emil Fulei Lykke Aagreen
<eaagre18@student.aau.dk>

Frederik Arnfeldt Jensen
<faje18@student.aau.dk>

Summary

It is as important as ever to secure software from threats from hackers and foreign powers trying to gain political information. If the Danish IT infrastructure was compromised it would have detrimental consequences to the Danish population and the government. An important part of the Danish IT infrastructure is the OIO-RASP protocol used for exchange of e-business documents. Therefore we aimed to test OIORASP with fuzz testing, which is an automatic test method.

OIORASP is maintained by Nemhandel which is a branch of Erhvervsstyrelsen. It supports business-to-business and business-to-government relationships and is used for sending e-business documents that follow the OIOUBL document standard. OIOUBL consists of standards for 15 types of business documents, including invoices and orders. We use fuzzing to test the OIORASP protocol.

Fuzzing is an automatic test method that focuses on finding errors and unexpected behavior that can easily be missed by other test methods. It does so by constructing random inputs and passing those to the target program. Fuzzing has become increasingly more popular over the recent years, and has evolved to be a popular bug finding method.

We implemented a guided model-based blackbox called OIOFuzz to fuzz the OIORASP protocol. It was focused on the Schematron validation of the protocol and it mutated and generated parts of OIOUBL documents. It is a mutation-based fuzzer with generation-based functionality. We ran different experiments with OIOFuzz which showed that our approach was a little bit naïve, but it did manage to find an error in the Schematron validation of OIORASP. The found error was unexpected as the input passed the schema validation of OIORASP, happening just before the Schematron validation, and therefore should be of the correct format to be validated with the Schematron. The error happened when a field in an example invoice document was duplicated, which is schema valid but the Schematron rules do not account for that.

With the found error, we concluded that OIOFuzz is functional and that it showed the potential of using fuzzing to test the OIORASP protocol. However we also concluded that it has room for improvement and more functionality that could enable it to find more errors could be added. We assessed that the areas that

could improve OIOFuzz the most, are functionality that could provide a broader exploration of Schematron rules.

Chapter 1

Introduction

The modern world is more and more reliant on IT. Everything gets digitalized from systems for companies and systems for governments infrastructure, making the need of these systems being secure really important. As of last year Danish companies has been target of Russian hacker attacks [15]. One of the precautions that can be taken is the testing, verification and validation of software to find vulnerabilities.

A method for automatic software testing that is gaining traction lately and has been used to find thousands of vulnerabilities in different software is fuzz testing, also referred to as fuzzing [12]. Fuzzing focuses on finding errors and unexpected behavior in a program at runtime. It does so by constructing random inputs and then pass those to the target program. Thereafter the output is observed, both to verify whether unexpected behavior was encountered and to be able to utilize information about the output to guide the fuzzer towards finding new program behavior. Fuzzing is particularly effective for testing edge cases and finding vulnerabilities that might have been missed by static program analysis and manual code inspection, such as penetration testing. Fuzzing is therefore different from most common test method, as these aim to test correct behavior or the functionality of specific features [12, 40].

In previous work, made along with two other students, we conducted a study of the Danish IT infrastructure, to verify the efforts made to ensure the security of it. The results of this study are presented in the chapter "Danish IT Infrastructure" in the student report "Towards Verification of the OIORASP Protocol" [8].

We now present a summary of the chapter in the following section to motivate the importance of testing components of the Danish IT infrastructure.

Summary of the Danish IT Infrastructure

The Danish government works towards *digitalization of the Danish public sector*. In this process they established Digitaliseringstyrelsen (The Digitalization Board) in 2011 to lead the digitalization [6]. The goals of a digitalized public sector is to lower manual work, improve productivity, and connect provided services. These goals are to be realized through six areas: digital infrastructure, data and technology, cyber and information security, digital service, law and digitalization, and IT leadership. Each area has some designated strategies for adhering to the current plan which range from 2022 to 2024 [4].

The plans for cyber and information security are based on threats assessed by the *Center for Cybersikkerhed* (Center for Cyber Security) [5]. One of the threats is the constant pressure from cyber criminals, which relentlessly attacks critical IT infrastructure. Another threat is from actors from foreign governmental bodies trying to get information on foreign and safety political areas. The Danish government has set four goals for increasing the security in a digital public sector. The goals are to increase robustness security of critical systems, increase competences of employees, increase collaboration with private companies, and collaborate internationally on cyber security. They realize these goals through 34 initiatives, and financed the Danish Cyber Defense with 500 million Danish kroner to attain these goals.

When a project uses more than 30 million Danish kroner on IT solutions per year, they need to provide a portfolio, which keeps track of certain areas of the project [7]. The areas are; usability, technical status, documentation and domain knowledge, economy, contracts and outsourcing, and security.

Solutions developed as part of the digital transformation of the Danish public sector are for example MitID, borger.dk and Digital Fuldmagt [3]. MitID is used to verify citizens, borger.dk is a platform for all publicly provided self-services, and Digital Fuldmagt is used for citizens to pass on power of attorney to relatives. In summary the Danish IT infrastructure connects over 2000 services, and the main focus is on the importance of maintenance, further development, possibility of refactoring, and scalability. Although the government has all these goals there is no mention of verification and validation of these systems, which would seem like an important thing. As such further testing of these systems are very relevant.

Based on this study we found it relevant to pursue further testing of these systems and choose to focus on the OIORASP protocol which is an integral part of Nemhandel. Nemhandel supports a big part of the Danish IT infrastructure by facilitating the sending of electronic business documents.

With this report we aim to explore the potential of using fuzzing for automatic testing of OIORASP. For this a guided model-based blackbox fuzzer, OIOFuzz, was implemented and used on the publicly available .Net library for OIORASP.

Report Outline

In Chapter 2 we present the OIORASP protocol, used for exchanging business documents in both business-to-business and business-to-government context, as well as the context it exists within. Additionally we present the OIOUBL e-business document standard that are used for the documents sent with the protocol, along with how the documents are processed in OIORASP. In Chapter 3 we present theory on fuzzing, where the most common approaches to fuzzing and some different fuzzing concepts are described. In Chapter 4 we first delimits the target to the Schematron validation in OIORASP and then describes the implementation of OIOFuzz, a guided model-based blackbox fuzzer targeting OIORASPs Schematron validation. In Chapter 5 we present experiments of fuzzing OIORASP with OIOFuzz and results of these. Related work is presented in Chapter 6. In Chapter 7 different aspects of the fuzzing harness are discussed. Our conclusion of the project is given in Chapter 8, while future work is considered in Chapter 9.

Project Code

The code for the OIOFuzz is publicly available at:

`https://github.com/fulei345/P10-Code`

The code for the .NET release of the OIORASP protocol is, as of writing (June 16, 2023), publicly available at [10]:

`https://rep.erst.dk/git/openebusiness/library/dotnet`

Chapter 2

OIORASP

The following chapter is partially a rewrite of the chapter "E-Commerce Standardization Efforts" from the student report "Towards Verification of the OIORASP Protocol" that we co-wrote with two other students [8]. Up to and including section Section 2.2 are a rewrite, section Section 2.3 is partly a rewrite, but has been expanded with more information. Section Section 2.4 is completely new.

In this chapter we introduce the OIORASP (Offentlig Information Online Reliable Asynchronous Secure Profile) protocol and the context it exist within. The OIORASP protocol is used for reliable and secure transport of business documents over the internet [9]. The protocol is used for exchanging business documents in both business-to-business (B2B) and business-to-government (B2G) context. It plays an important part in the Danish IT infrastructure as it is used for Danish e-commerce. The protocol is part of efforts to standardize e-commerce in EU through security standards, schemas, and internet protocols. It is mandatory by law to use the OIORASP protocol in B2G context.

Security for the Danish e-commerce infrastructure is of high importance as deficiencies would have consequences impacting all Danish citizen. Testing the OIORASP protocol is therefore important.

As far as we know, no extensive documentation of the OIORASP protocol is available publicly, which has resulted in parts of this chapter being written without explicit sources. These parts is instead written from available guides, inspection of the source code from the OIORASP repository [10] (specifically the .Net version of the library), and analysis of network traffic intercepted with the tool WireShark [10].

Chapter Outline

In Section 2.1 the international and the Danish infrastructure for e-business is presented, to provide the context OIORASP exist within. The OIORASP protocol is

presented in Section 2.2 with a general overview given in Section 2.2.1 and details on the protocol in Section 2.2.2. OIOUBL, which is the document standard used in the OIORASP protocol, is presented in Section 2.3. In Section 2.4 details on the document processing in components of the OIORASP repository is presented.

2.1 E-business Standards

Standards for exchange of e-business documents between government, companies, and customer exist. These standards is meant to ensure that the process is easy and secure. Using common documents and transportation processes facilitates the interoperability between different entities in a business process. Different companies maintain various standards for e-business with the aim of facilitating the interoperability both nationally and internationally.

One of such is the Universal Business Language (UBL) standard, which is maintained by the Organization for the Advancement of Structured Information Standards (OASIS) [11]. The OASIS UBL standard is an XML library of common business documents, such as invoices, catalogues, and purchase orders. UBL is intended to ensure interoperability in general business processes internationally.

2.1.1 OpenPeppol

OpenPeppol is a European association in charge of efforts for European e-procurement standardization, aiming to facilitate interoperability for business processes between European countries. It maintains the Peppol Business Interoperability Specifications (BIS) standard, which are formal requirements ensuring pan-European interoperability of procurement documents. Peppol BIS is devised by the European Committee for Standardization (CEN) during their Business Interoperability workshops (CEN BII). It utilize the UBL standard for the document standardization. OpenPeppol provides technical guidelines for embedding the document standard in existing solution for business document transportation [26, 24].

Additionally it also provides the Peppol eDelivery network, which is a central e-business document transport solution. The Peppol eDelivery network is used to exchange Peppol BIS business documents between local Peppol authorities in European countries. Peppol maintains a centralized Service Metadata Locator (SML) that the eDelivery network depends on, as it defines which Service Metadata Publisher (SMP) to use when retrieving delivery details. The Peppol authorities in each European country must ensure that their Access Points and SMP services conforms to technical service specifications [25].

2.1.2 NemHandel

Erhvervsstyrelsen (ERST) is the Danish Peppol authority. NemHandel is a branch of ERST that provides the Danish Peppol SMP, NemHandelsRegistret (NHR). NHR is the access that is connected to, to retrieve the required delivery details, when sending Peppol BIS documents to Danish companies [9].

NemHandel is meant to facilitate easy and secure exchange of business documents both internationally and within Denmark. It aims to support both senders and receivers in B2G and B2B processes. NHR, besides acting as the Danish Peppol SMP, also acts as the Danish national SMP.

The Danish document standard for e-business is Offentlig Information Online UBL (OIOUBL), which is a subset of document profiles from the UBL 2.0 standard. Which profiles are relevant according to the requirements of Danish national business processes is determined by ERST [16].

2.2 OIORASP

Offentlig Information Online Reliable Asynchronous Secure Profile (OIORASP) is the transport protocol used for exchanging business documents with companies registered in NHR. It is the national adaption of the international family of web service (WS) standards , and it is meant to facilitate easy and secure transport of e-business documents. OIORASP can be considered as the Danish counterpart to the Peppol eDelivery standard. The purpose of the protocol is to provide secure and reliable exchange of business document in asynchronous environments [9].

According to our knowledge extensive documentation of the OIORASP is not publicly available. Service providers registered in NemHandel has to contact NemHandel directly for assistance when implementing the protocol.

In Section 2.2.1 a brief overview of the protocol is presented, while more details is presented in section 2.2.2.

2.2.1 Protocol Overview

An overview of the infrastructure of the OIORASP protocol is shown in Figure 2.1 [9]. An e-business document to be send is produced in the senders own internal business system. From there it is forwarded to an Access Point (AP), also referred to as endpoints, which can be hosted externally. The AP handles the transportation of the e-business document, in a secure and reliable manner, meaning the sender does not have to implement this functionality in their business system. From the AP the sender retrieves the address of the receivers AP from NHR and sends the business document to the receivers AP using the OIORASP protocol. Services such as NHR, where the addresses of registered APs can be

found, is provided by NemHandel to facilitate the functionality of the OIORASP protocol.

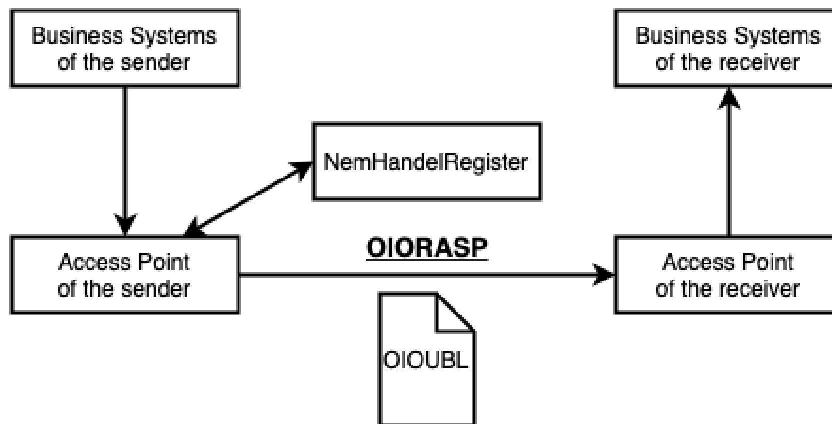

Figure 2.1: From [8]. The OIORASP protocol is used to send business documents between two APs in asynchronous environments in a reliable and secure manner [9]. The protocol is often implemented by AP providers, s.t. businesses themselves do not have to.

The sending of document with the OIORASP protocol consist of four central steps:

1. Look up in a UDDI-server for the receivers endpoint address and UUID.
2. Download receiver certificate from an LDAP-server.
3. Check validity of the receivers certificate with a lookup to an OCSP-server, supplemented by a validity check against a CRL.
4. Send the business document to the receiver.

An overview of the connections made and actions performed during the four steps are presented in Table 2.1. A more detailed description of the steps is presented in Section 2.2.2. During the third step two separate connections is made. The first connection made is to the OCSP-server and the second connection is to an address where the CRL can be downloaded from. Further details on the requests made and their corresponding responses are described in the following section.

2.2.2 Protocol Details

In this section details on the four central steps of the OIORASP protocol, introduced in Section 2.2.1, are presented. The presented details on the protocol are based on information gathered from inspection of the code from the .Net version of the OIORASP repository [10] and network traffic intercepted with WireShark [38]. The intercepted network traffic is from communication between the `ClientExample` from the OIORASP repository, run locally, and a NemHandel demo endpoint.

Step	Access Point	Action Summary
1. Endpoint lookup	discoverypublic .nemhandel.com	Request user UUID using a GLN number. Then receiver endpoint information is requested and received. Lastly, receiver model details are retrieved.
2. Certificate download	crtdir.certifikat.dk	LDAP search request, followed by downloading receiver certificate.
3. Certificate validation	ocsp.ica04.trust2408.com	Certificate validity check by lookup using OCSP.
	crl.oces.trust2408.com	Certificate validity check against CRL.
4. Send document	demo.nemhandel.dk	Send document to receiver endpoint.

Table 2.1: Summary of the four central steps of the OIORASP protocol. Each step is shown, along with the connections made, and a short summary of the performed actions during the step. The fourth connects to the user endpoint. In this case it is the demo endpoint provided by NemHandel [8].

Look Up Receiver Endpoint Address

The first of the four central steps to sending business documents with OIORASP is to look up the address of the receiving endpoint. The receivers GLN (Global Location Number), which is specified in the business document, is used to find the receiving endpoint address. The GLN is sent to the NemHandel SML, which then responds with the address for the receiving partys remote endpoint.

The NemHandel SML is an UDDI-server (Universal Description, Discovery and Integration), containing information about registered SMPs. Information about client endpoints is provided by the SMPs. Multiple SMPs are hosted by NemHandel, with the NHR being on of them, while others exist for integration testing.

The lookup on the UDDI-server consist of three SOAP (Simple Object Access Protocol) requests, which is modelled in Figure 2.2. An exchange of UUIDs (Universally Unique Identifier) is the core essence of the requests. UUIDs is unique identifiers, which is in this context used for schemas and services in the NemHandel UDDI-server. The receivers GLN is sent to the UDDI-server in the first SOAP request and the receivers UUID is returned in the response to the sender. The UUID is then used in the second SOAP request, which requests further information on the receiver. Information about endpoint address and a string certificate of the receiver is returned in the response to this request. The last SOAP request sent

is meant to gather details on the model used by the receiver. The response to this request specifies APs containing different information on the service provided by the receiver, part of which is the types of documents the receiver accepts.

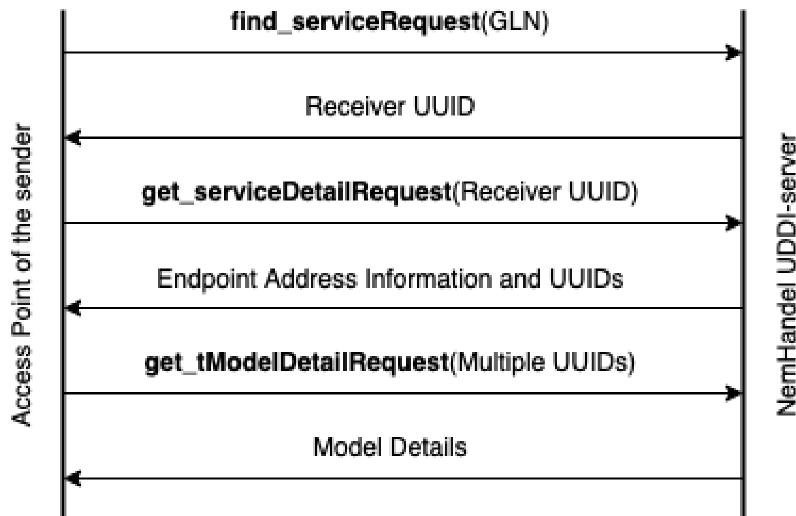

Figure 2.2: An overview of communication when the AP of the sender requests details of the receiver from NemHandel UDDI-server [8].

Download Receiver Certificate

The string certificate obtained from the UDDI-server response is to download the receiver certificate object. The country code and serial number are extracted from the string certificate. These are used to make an LDAP (Lightweight Directory Access Protocol) search request message. This message is used for identifying and downloading the certificate from the certificate directory. The serial number is used for the lookup in the directory and if it is found, the response to the request will contain a certificate object, which has accessible fields for the different parts of the certificate.

Certificate Validation

After the receivers certificate has been downloaded, it needs to be validated to ensure that it is a trusted entity. The certificates used with the OIORASP protocol are X.509 public key certificates and are part of Nets DanID public key infrastructure (PKI) [23]. An OCSP-server (Online Certificate Status Protocol), provided by Nets, is used to validate certificates issued by NemHandel. This validation is supplemented with validation of the certificate against a CRL (Certificate Revocation List) provided by NemHandel. In the CRL it is specified which certificates that

have been revoked and are thereby not valid any longer. The receivers certificate is first validated with the OCSP-server and then against the CRL.

Sending The Business Document

The last of the steps is the sending of the business document. The document is sent through a HTTP POST request to the receivers endpoint address. WS-ReliableMessaging, a web service protocol standard used for reliable delivery of SOAP messages, is used to establish a connection for the transport of the business document. A WS-ReliableMessaging CreateSequence element is then used to establish a sequence of messages. This element contain the ID and ID type for both sender and receiver and the receivers endpoint address.

A request object corresponding to the document being sent is formed, with the endpoint address and identifiers for sender and receiver. Statens It (The Agency for Governmental IT Services) provides the specific namespaces for each SOAP action request. The business document is sent after being encrypted along with signature information.

The encryption of the document is done using the AES-256-CBC block encryption algorithm and the document is signed with a RSA-SHA1 signature. The public key of the receivers endpoint certificate is used as the encryption key for the AES-256-CBC algorithm, s.t. only the receiver can decrypt the document.

If, e.g, the document being sent is an invoice, then a SubmitInvoiceRequest SOAP action will be sent. If the receiver accepts the senders certificate, which is sent alongside the SOAP message, as valid, the receiver will respond with a SubmitInvoiceResponse. This response tells that the document was received successfully and the sequence of messages is ended with terminating SOAP messages.

2.3 OIOUBL - Universal Business Language

OIOUBL is the document standard used by the OIORASP protocol. It is a Danish adaption of the UBL document standard and is a subset of UBL 2.0. The focus for the UBL 2.0 standard is automating online transactions, to save on resources and costs. OIOUBL consists of standards for the format of 15 different types of business documents, such as orders and invoices. Some document types from the UBL 2.0 standard are excluded from OIOUBL. OIOUBL is an extension of OIOXML electronic invoice, an older document standard based on an older version of UBL. Guidelines exist for each of the different document types, which describes the document types overall class structure and contains description of classes and elements in the document type [16]. The structure of an OIOUBL invoice document is presented in Figure 2.3, as an example to provide an overview of the structure of OIOUBL documents.

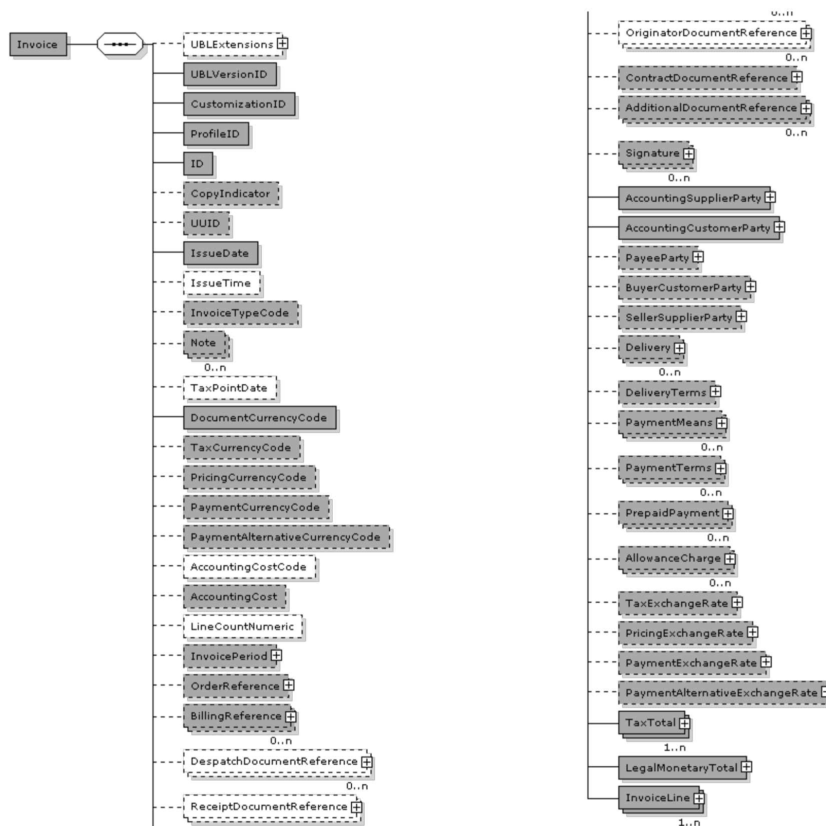

Figure 2.3: Structure of an invoice document. This shows the classes and fields in the document as well as the order they should be in. Classes are marked with a plus at the side. Mandatory classes and fields have full lines, while optional classes and fields have dashed lines. The grey fields and classes can always be used, while the white fields and classes only can be used with bilateral agreement. This structure is from the guideline for the invoice document type [17].

2.3.1 OIOUBL Properties

The Common Class Library is described as the backbone of OIOUBL. The library contains descriptions of all elements that exist in the different types of OIOUBL documents. It is intended to produce the highest possible reusability for the elements.

Two elements that are particularly important in order to understand OIOUBL documents are Party and EndpointID. All types of OIOUBL have two parties that are mandatory, the party for the sender of the document and the party for the receiver of the document. Additionally some documents may also contain other relevant parties. The parties can have different titles between the different types of documents, s.t. the title express the role that the party has in the business process. A unique identifier must be assigned to a party.

The EndpointID element is a unique identifier for the endpoint at the docu-

ments target destination. For all parties that are included in the document an EndpointID must be specified. An EndpointID must be recognizable for the established address register.

Code lists are another significant part of OIOUBL. They are used to specify allowed values for certain elements, e.g. specific currencies or country codes that can be used. A set of code lists exist for the Danish OIOUBL customization. The code lists is meant to assist in achieving fully automated processing of data exchanged for OIOUBL.

Both senders and receivers can specify which types of OIOUBL documents they support, through the use of profiles. A ProfileID that identifies the profile a document relates to must be specified in an OIOUBL document. A profile can describe one or more interconnected business processes, where the processes can be part of one or more OIOUBL documents. The profiles is used to determine which types of documents a party must be able to send and receive, a partys role in the business process, and which processes the party must support.

OIOUBL documents utilize several namespaces. These namespaces are expressed with alias variables in the top of the documents. Throughout the rest of the document the alias variables is used to reference the namespaces [16].

OIOUBL Document Attributes

OIOUBL documents consists of fields and classes. The fields and classes both have the following attributes:

- **UBL-Name**: The name of the field or class in UBL 2.0.
- **DataType**: What data type that field has e.g. Identifier, Date, Text, Code or Numeric. For the classes the datatype is its classtype.
- **Usage**: Whether the field or class always can be used or if it needs bilateral agreement.
- **Cardinality**: Defines how many fields or classes with this name there can be. This can be 1, 0..1, 0..n, and 1..n.

The classes contains their own fields and subclasses, that have the same attributes [17].

2.3.2 Validation of OIOUBL Documents

XSD (XML Schema Definitions) is used to validate the structure of the OIOUBL documents. XSD schemas define and describes the structure of an XML document [33]. OIOUBL specifications are directly based on UBL 2.0 schemas, but with

a few elements excluded due to business related reasons, such as not being of relevance in a Danish context [16]. For OIOUBL documents, the schemas is used to validate that the mandatory fields and classes are present in the document e.g. ID, UBLVersionID and IssueDate. Furthermore it is also used to validate the data types of the fields e.g. identifier, date, and numeric.

Schematron is used to validate business rules for an OIOUBL document. Schematron is an XML schema language and is used to validate XML documents. Rules and checks, which are defined XPath query patterns, are used to validate document instances. A rule could e.g. be that an identifier or code is one of the valid code list values or that a numeric field has some exact value. Schematron files is written as XSLT (Extensible Stylesheet Language Transformations) stylesheets [2].

XSLT is a language used for transforming an XML document into a document consisting of XSL (Extensible Stylesheet Language) formatting object or into other formats similar to XML, such as HTML, by adding styling information to the document. The XSLT transformations expresses rules for transforming source XML trees into result XML trees. A XSLT stylesheet contains a set of template rules. These rules consists of three parts; a pattern used to match with nodes, a set of parameters which can be empty, and a sequence constructor which is used to produce sequences of items through evaluation. The template rules associates a pattern, that matches nodes in the source document, with a sequence constructor and evaluates the sequence constructor, which will often result in construction of new nodes for the result XML tree [35].

XSL is a language used to express stylesheets. It describes how different XML documents should be displayed and is used in XSLT. Additionally it contains advanced styling features that are expressed by an XML document type and defines a set of elements called formatting objects, which expresses the resulting format of the object after a transformation [34, 31].

2.4 Document Processing in the OIORASP Library

When an OIOUBL document is sent with the OIORASP protocol, it is processed in different phases of the protocol. These phases are visualized in Figure 2.4. The first three phases happens at client side. Here the document is first parsed with an XML parser. Then the document is validated, first against the schemas for the specific type of document and then against the Schematron for the document type. Thereafter the document is attached to a SOAP message, where it is encrypted.

The document is then sent to the server, where it goes through two more phases. When the server receives the SOAP message from the client, it extracts the OIOUBL document, where it is decrypted and parsed. Thereafter the server also validates the document with the schemas and Schematron for the document type.

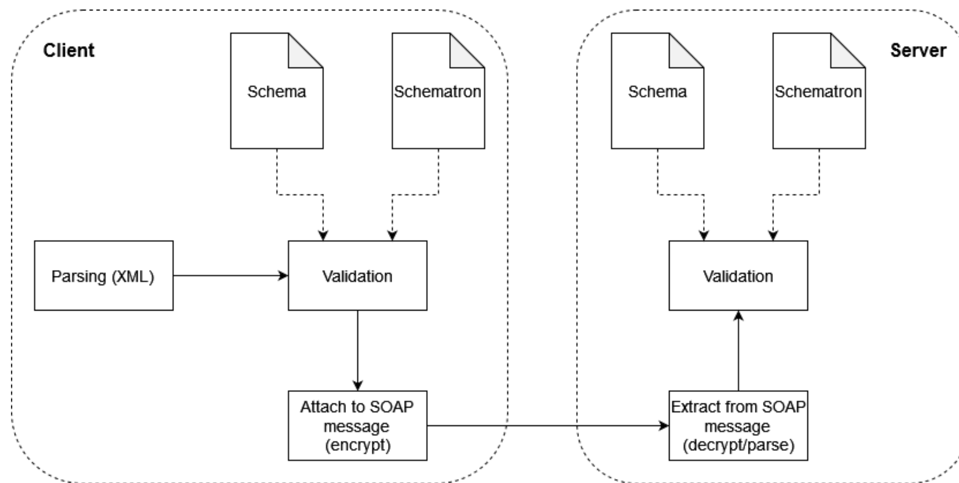

Figure 2.4: The processing of the OIOUBL document throughout client server communication in the OIORASP protocol.

Chapter 3

Fuzzing Theory

In this chapter we will present theory on fuzzing, which will be used for testing OIORASP. Fuzzing is an automatic test method where unexpected inputs are automatically created for a target program. These inputs is then provided to the target program in an attempt to discover bugs in the program [40]. A fuzzer cannot verify correctness of a program in respect to a specification but is suitable to be used to find runtime errors. After the fuzzer has run the user will have a set of inputs that exercise different program behaviors. If some of these program behaviors leads to a crash, the corresponding input can be used the recreate the crash and can therefore be used to identify the bug.

The term *fuzzing* was used for the first time by Professor Barton Miller at the University of Wisconsin in 1988. Miller experienced noise in a dial-up connection, causing bad inputs to the UNIX commands at the ends of the connection. The bad inputs led the UNIX utility programs to crash. This later inspired Miller to give his students a programming exercise that had them creating the first *fuzzers*.
The Fuzzing Book [40], which has been used as the primary source for this chapter, is an online, interactive book that covers the basics of fuzzing and state of the art fuzzing techniques that are being researched.

Chapter Outline

The three common approaches to fuzzing, black-, white-, and greybox fuzzing, is presented in Section 3.1. Different concepts used in fuzzing are presented in Section 3.2. In Section 3.3 the state-of-the-art greybox fuzzer American Fuzzy Lop is introduced, along with a look into some of the techniques it uses. It is introduced as it is the state-of-the-art fuzzer and is used as a baseline for many other fuzzers.

3.1 Fuzzing Approaches

In this section we presents the three common approaches to fuzzing; black-, white-, and greybox fuzzing. The difference in the approaches is found in the amount of program information that the fuzzer utilize. Blackbox fuzzing, an approach where the fuzzer utilize only external program information, is presented in Section 3.1.1. In Section 3.1.2 whitebox fuzzing, an approach where static analysis techniques are used on the target program to gain program information that the fuzzer leverages, is presented. In Section 3.1.3 greybox fuzzing, an approach where instrumentation of the source code is used to gain program information that the fuzzer can leverage, is presented.

A small example program is presented in Listing 3.1. The program is used throughout this section as an example, to showcase the differences in how the different approaches works with a concrete example. The small program takes a string and first checks if the length of string is 24 and then checks if the character at index eleven is an *a*. If both are true it raises an exception.

```
1 def func1(s: str) -> None:
2     if len(s) == 24:
3         if s[11] == 'a':
4             raise Exception()
```

Listing 3.1: Example program using python syntax, which takes a string as an input

3.1.1 Blackbox Fuzzing

Blackbox fuzzing is a fuzzing strategy where the target program is seen as a blackbox. This means that a blackbox fuzzer is only able to observe and use external program information for the fuzzing process, e.g. the execution time of the program or whether the program has crashed. Blackbox fuzzers constructs new inputs, passes them to the target program, and observes the program output.

Blackbox fuzzers advantage is the ease of implementation, as no prior knowledge of the targeted program is required. The disadvantage of blackbox fuzzers is that they are rarely able to pass certain conditional statements due to their random construction of inputs. Not being able to pass the conditional statements makes the testing of the target program shallow and bugs that exist deeper in the program behavior will not be found [40].

> **Example 3.1 (Blackbox Fuzzing)**
> Consider a blackbox fuzzer being used on the small program presented in Listing 3.1. The fuzzer would generate random strings to pass as input for the

> `func1()` function. The probability of one of these string passing the first constraint on line 2 is already low, as it will have to be exactly 24 characters long. The probability of the string then also passing the second constraint on line 3 is even lower, as it would then also need to have an *a* at index eleven. Assuming that the fuzzer chooses a random string length between 1 and 1000 before constructing the characters of the string, would give it a probability of 0.1% to generate a string with a length of 24. Additionally assuming that the constructed strings is composed only of letter from the Danish alphabet, both capital and non capital, the probability of character at index eleven being an *a* is $1/56 \approx 1.8\%$, giving a total probability of $0.1\% \times 1.8\% \approx 0.0018\%$ of the second constraints being passed. As such, the fuzzer, will most likely have to run for a long time to explore all lines of this small program with two nested constraints. With a bigger target program with more constraints, the blackbox fuzzers potential for covering the full code base will be extremely low, almost non-existing. Particularly code blocks nested within multiple constraints is likely to not be explored and potential bugs in those code blocks would not be found.

3.1.2 Whitebox Fuzzing

Whitebox fuzzing is a fuzzing strategy where information about the target programs source code is used by the fuzzer. The information can be used to guide the fuzzer s.t. the amount of the code covered during the fuzzing process is increased or to target specific program locations. Different techniques for gaining the information, such as symbolically executing the target program, can be used [13].

An example of a whitebox fuzzer is SAGE, which is developed by Microsoft [14]. SAGE dynamically conducts a symbolic execution of a program to gather constraints on the conditional branches that are encountered throughout the execution path. It then negates the constraints and attempts to solve them with a constraint solver, which is used to make new inputs that can explore different program paths, as they can be made to explore the different conditional branches.

> **Example 3.2 (Whitebox Fuzzer)**
> Consider a whitebox fuzzer being used on the small program presented in Listing 3.1. Like the blackbox fuzzer it will generates random strings to pass as input to the `func1()` function. However, contrary to the blackbox it will beforehand have gathered information about the two constraints and found out how to solve them. Therefore it will be able to guide the fuzzing process to explore all lines of the function, by making strings that will pass one, none, or both of the constraints. The whitebox fuzzer thereby has the ability to find potential bugs

> nested deeper in the program behavior, that the blackbox fuzzer will be unlikely to find.

Using static analysis techniques provides whitebox fuzzer with information about the target program that can be used to guide the fuzzing process, but it comes with the cost of often being difficult and costly to implement. Moreover full access to the source code is also required to implement whitebox techniques. The techniques, when implemented, also must be adjusted to the programming language of the target program, meaning that before whitebox fuzzing is applicable, a large programming task has to be overcome.

3.1.3 Greybox Fuzzing

Greybox fuzzing provides a middle ground between blackbox and whitebox fuzzing, which can often be desired since blackbox fuzzing lacks information that can be used to guide the fuzzing process, while whitebox fuzzing can be computationally heavy and at times difficult to implement. Greybox fuzzing is a fuzzing strategy where light weight information on the target program, e.g. code coverage, is used to guide the fuzzing process. Techniques that can be used for greybox fuzzing are often built into programming languages, e.g. the Python function `settrace()` from the `sys` core module, which allows for passing a function to be called each time a line of code is executed and can thereby be used to track which lines has been executed [40].

> **Example 3.3 (Greybox Fuzzer)**
> Consider a greybox fuzzer, using code coverage information, being used on the small program presented in Listing 3.1, where it will generate random strings for the input. Like the blackbox fuzzer, the greybox fuzzer has no prior knowledge of how to pass the two constraints in the program. However, when it manage to generate a string that passes the first constraint on line 2, the greybox fuzzer will gain the knowledge that the string achieved new code coverage. This guides the fuzzer to make more string with a length of 24, as it will use the string that passed the constraint to construct new strings. This gives it a better opportunity of eventually also passing the second constraint.

3.2 Fuzzing Concepts

In this section we will describe some different concepts for fuzzing. First, in Section 3.2.1, mutation- and generation-based fuzzing is presented, which use two

different methods for constructing inputs. Secondly, in Section 3.2.2 the concept of smart and dumb fuzzers is introduced. Then the use of code coverage for fuzzers is introduced in Section 3.2.3, and the concept of a power scheduler is introduced in Section 3.2.4.

3.2.1 Mutation- and Generation-based Fuzzing

As fuzzing entails automatically creating unexpected inputs for a target program to try to discover bugs, how the input is constructed is an important consideration. In this section we present two different methods for constructing the inputs: mutation- and generation-based.

Mutation-based fuzzers uses a corpus of initial input seeds for the construction of inputs. The initial corpus contains well formed inputs for the target program. In order to optimize the process of finding new program behavior during fuzzing, it is preferable that the corpus contains inputs causing diverse behaviors.

The fuzzing process for mutation-based fuzzer starts by selecting an input seed from the corpus, which is then mutated before being passed to the target program. The mutated input can thereafter be added to the input corpus, if it has led to new program behavior. Mutations to the input is often made at the byte level of the input data, s.t only singular bytes is affected by the mutation. Such mutations do not require knowledge of the inputs structure and restrictions, and are as such often easy to implement. Constructing input through mutations comes with the downside that the constructed inputs are often unable to reach deep in the program behavior, since random mutations are unlikely to find the correct values for path constraints [40].

Generation-based fuzzers constructs input for the target program by generating the input from scratch based on a specification of valid input. A specification of valid inputs could e.g. be grammars. A grammar is particularly useful for expressing the syntactical structure of an input and can therefore be used to ensure that generated input adhere to the structure. As an example if the input that are generated is a date, the first 4 digits needs to represent a year, and the next 4 digits needs to represent a month and a day respectively. A grammar can then ensure that the generated input adhere to these structural rules of the date format [40].

Using structures such as grammars allows for the possibility of deriving derivation trees of the generated input. Derivation trees allows for easy expansion and replication of inputs, and analysis of the derivation of interesting inputs.

A downside for generation-based fuzzers is the requirement of knowing the input specification before being able to implement an input generator and potentially defining a grammar or similar to use for the generation.

3.2.2 Smart and Dumb Fuzzer

Another way of categorizing fuzzers are as smart and dumb fuzzers [20].

A dumb fuzzer is dumb because it does not take the context of the target or the input into account. The only thing it knows is the resulting output of a specific input. The advantages of a dumb fuzzer is that a simple version is quick and easy to setup. On the other hand it is unlikely to make valid instances of highly structured inputs such as XML. As such it will also mostly, or even completely, be stuck at the parser if the input is a file, rather than actually fuzzing the program. A very simple blackbox fuzzer is a dumb fuzzer.

A smart fuzzer is smart since it utilizes information about the target and its input. The information can e.g. be the code coverage archived by an input, or the structure of the input. With this information it is more likely to find bugs since it can reach more of the target program. The disadvantage is that it requires more analysis of the target, and more work to setup. A lot of greybox and whitebox fuzzer are smart fuzzers.

3.2.3 Code Coverage

An common concept in fuzzing is code coverage. Code coverage is commonly used in testing, as it is a good measurement of how much of the program functionality has been tested. The two most common metrics for code coverage in fuzzing is statement coverage and branch coverage. Statement coverage is a measurement of how many of the statements in the code that has been executed. Branch coverage focus on whether each branch in conditional statements has been taken. Fuzzers, particularly greybox fuzzers, uses code coverage to guide the fuzzing process s.t. they can explore non-executed statements. They do this by choosing input seeds that exercise new program behavior more often [40].

3.2.4 Power Scheduler

Another concept is the use of a power scheduler, which is introduced by The Fuzzing Book [40]. A power scheduler is used to choose the next input to be fuzzed and assigns energy to each input seed. The higher energy a seed has, the higher is the probability of it being chosen. The power scheduler assign more energy to interesting seeds, based on some criteria. This criteria could be seeds with more code coverage or seeds that discovers new coverage.

3.3 American Fuzzy Lop

American Fuzzy Lop (AFL) is a state-of-the-art coverage-based greybox fuzzer [39]. AFL uses light weight instrumentation of C programs, which is injected into the

target program when it is compiled with an `afl-gcc` option. AFL is made to be an all general purpose tool, that implements a range of techniques that have been found to be effective for fuzzing.

One technique concerns coverage measurements, with branch coverage being captured by the instrumentation injected in the target program. AFL count the amount of times one branch of the code has been reached directly from another branch. With `A`, `B`, and `C` being code branches and a execution trace `A` $\rightarrow$ `B` $\rightarrow$ `C`, AFL will store the tuples `(A, B)` and `(B, C)`. A global map containing the branch coverage tuples is kept. This way of storing the branch coverage allows AFL to differentiate between different execution traces quickly. Storing only the tuples helps to avoid comparing large traces and thereby risking problems with path explosions. AFL stores coarse tuple hit counts in buckets in the ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, +128.

A second technique used concerns avoiding computational overhead by storing as little information as possible. A trace is only stored if it is considered interesting. Traces that are interesting is e.g. ones where a new tuple is encountered, or where the hit count of a tuple moves to a new bucket. As an example consider the two traces `A` $\rightarrow$ `B` $\rightarrow$ `C` $\rightarrow$ `D` $\rightarrow$ `E` and `A` $\rightarrow$ `B` $\rightarrow$ `C` $\rightarrow$ `A` $\rightarrow$ `E`. The second trace would be considered a interesting trace and therefore stored, as it contains two new tuples, `(C, A)` and `(A, E)`, compared to the first trace that has been explored beforehand. An execution trace might cover a lot of branches without being considered interesting if it does not contain any new tuples.

When an interesting trace is encountered, both the branch coverage tuples and the mutated input producing the trace are stored. These inputs can then be used as input seeds for further testing.

Chapter 4

Implementing OIOFuzz

In this chapter we present the implementation and general design of our fuzzing tool OIOFuzz used to fuzz test the OIORASP protocol. We developed our own fuzzing tool, rather than deploying an existing tool. This was chosen to get a fuzzing tool that is highly specialised towards the OIORASP protocol. This also means that the fuzzing tool is not meant as a general purpose tool, although it implements well known fuzzing concepts, that are used for general purpose or for other specialised tools. OIOFuzz is a guided model-based blackbox. It primarily utilize mutation-based fuzzing but contains some functionality for generation-based fuzzing as well.

The implementation is a proof of concept, aiming to demonstrate a fuzzing tools potential for automatically testing the OIORASP protocol. The implementation has been made with a focus on one of the 15 business document types included in the OIOUBL document standard, specifically the invoice document type. A model expressing the structure and types of the elements has been made for the invoice document type, but not for the other business document types. We have instrumented the `ClientExample`, which is the OIORASP implementation of the client side, which reside in the sample folder of their .NET repository [10]. The `ClientExample` and the `httpEndpointExample`, which is their implementation of the server endpoint, is their own C# project inside the library. The fuzzer is guided by the output from the `ClientExample`.

Chapter Outline

First in Section 4.1, the target for fuzzing the OIORASP protocol is delimited to focus primarily on the Schematron validation in the OIORASP protocol. The general approach taken to fuzzing the OIORASP protocol is sketched in Section 4.2 using a flowchart. In Section 4.4 details on the OIOUBL documents, that are important for the implementation of OIOFuzz, is presented. In Section 4.3 the setup

of OIORASP server and client endpoint, as well as changes made to the library to facilitate fuzzing it, are described. An overview of the structure of OIOFuzz is given in Section 4.5, while details on different components of OIOFuzz is provided in Section 4.6. Lastly we briefly assess the classification of OIOFuzz in Section 4.7.

4.1 Delimiting the OIORASP Fuzzing Target

In this section we will narrow in the target for the fuzzing of the OIORASP protocol. In Chapter 2 we described the different concepts for the OIORASP protocol such as the process of sending the document and the validation of the documents. Among those the validation of the documents, particularly the Schematron validation, contain the most OIORASP specific components and are deemed more susceptible to human error. Therefore this is chosen as the primary target to fuzz.

The Schematron validation is used to ensure that the document complies with certain business rules applicable for the type of document. The Schematron validation is executed with `Saxon` API for .Net. `Saxon` is a package containing tools for processing XML documents with XSLT, XQuery, XPath, and XML Schema [29]. The `Saxon` .Net API is run with the document and the XSL file for the OIOUBL Schematron for the document type.

The Schematron validation of the documents was chosen as the target, as it contains the most OIORASP specific components. Most of the functionality of the OIORASP library is achieved using .Net namespaces, most often in a straightforward manner. While there is certainly potential for finding errors in the use of .Net namespaces and integration of these, we believe there is a higher chance for bug finding in the Schematron validation. The Schematron XSL files are custom made for the OIOUBL documents and the files are rather big, e.g. the Schematron file for the invoice documents is 36470 lines long, making them more susceptible to human error. Moreover the Schematron rules are not described formally elsewhere, but only indirectly through the guidelines for the documents. Pairing that with the use of a third party tool for the validation of the Schematron files, makes the Schematron validation a good target for potentially finding errors.

Targeting the Schematron validation depends on the fuzzing being directed towards creating more schema valid documents.

The Schematron validation of OIORASP is an unusual fuzzing target, since it is a document that gets compiled. This is unlike common fuzzer targets, which is usually a program or a network protocols that can crash.

4.2 Fuzzing Approach

A flowchart showing our process for fuzzing OIORASP is presented in Figure 4.1. An OIOUBL document is chosen from the corpus, mutated, and then passed to the client. In the client the document is first parsed with an XML parser and thereafter validated with schemas and Schematron. These part are encapsulated inside the dashed lines, to show that they are part of the client. If an error happens at these parts, the document will be logged. While the target of the fuzzing is the Schematron validation, the client will still go through the parsing and schema validation and it is still possible that errors could be found there, in which case they will be logged too, as they are important to track as well. Lastly the corpus is updated, where the document that was used can be added, s.t. it can be used for later fuzzing iterations.

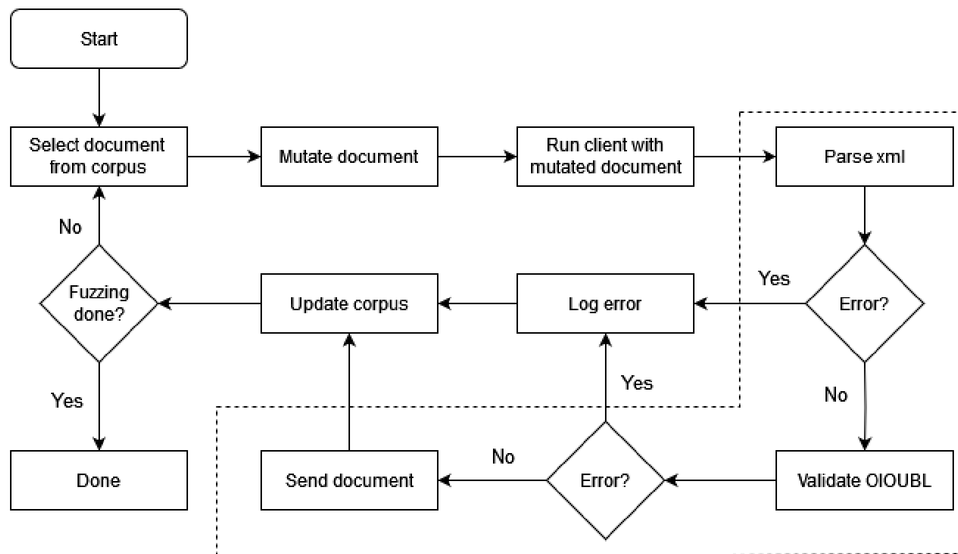

Figure 4.1: A flowchart showing an overview of the general approach taken to fuzz the OIORASP protocol. The part encapsulated by the dashed lines is part of the client.

4.3 OIORASP Library Setup

Certain alterations is made to the library, mostly to the demo client and server endpoints provided as part of the library, to support the fuzzing harness.

We use a test certificate for authentication when running the OIORASP server endpoint, as production certificates for OIORASP is only available for NemHandel service providers. The test certificate is provided in the library. However, the demo endpoints in the library are configured to use production certificates. Therefore the server endpoints configuration needs to be modified to use the publicly available

test certificates. This is done by adding the root certificate of the used test certificate to the list of valid root certificates, that is contained in the configuration file.

The demo server endpoint is hosted as a IIS (Internet Information Service) application. In order to use the test certificate with the IIS application the application pool used for the application has to be given permission to access the test certificates private key.

The demo client endpoint is changed to simply return the URL of the locally hosted server instead of making the UDDI lookup to get the address of the endpoint. Moreover, the LDAP and OCSP steps of the protocol are removed from the demo client, as these steps deals with downloading and verification of the server certificate. These steps are irrelevant for the local server with the test certificate. This means that the client endpoint now only executes the fourth of the steps presented in Section 2.2.1.

The library were also altered with some light instrumentation to support tracking of code coverage.

4.4 OIOUBL documents

The OIOUBL document standard specifies highly structured documents, which present certain challenges that needs to be considered during the fuzzing of the documents.

One set of structural rules originates from the OIOUBL documents being of the XML document format. Elements in XML documents consists of start-tag and a matching end-tag. A start-tag must have a matching end-tag and elements must be properly nested [30]. A small part of an example OIOUBL invoice document is presented in Listing 4.1. This listing contains an OrderReference element with three child elements. The complete document can be found in Appendix A.

```
1 <cac:OrderReference>
2   <cbc:ID>5002701</cbc:ID>
3   <cbc:UUID>9756b468-8815-1029-857a-e388fe63f399</cbc:UUID>
4   <cbc:IssueDate>2005-11-01</cbc:IssueDate>
5 </cac:OrderReference>
```

Listing 4.1: An OrderReference class and its child element, found in an example invoice document.

If the fuzzer makes changes to any of the start-tags or end-tags, e.g. `<cbc:ID>` in the listing, or changes the order the tags comes in, the document will no longer be a valid XML document. As such, when sending such a document with the `ClientExample` it will not get past the XML parsing.

Additionally, the OIOUBL document standard comes with its own structural rules, which is expressed in the schema rules that the document has to adhere to and in the guideline for the document type. These rules specifies elements that must be present as well as elements that can be present. The rules exist for both the

root class and for the specific subclasses that can exist in the document. Moreover it also specifies the order that the elements must appear in. The schema rules for the OrderReference class from the invoice document type, is visualized in Figure 4.2. The ID field in the OrderReference class is mandatory, which can be seen in the figure as it has full lines.

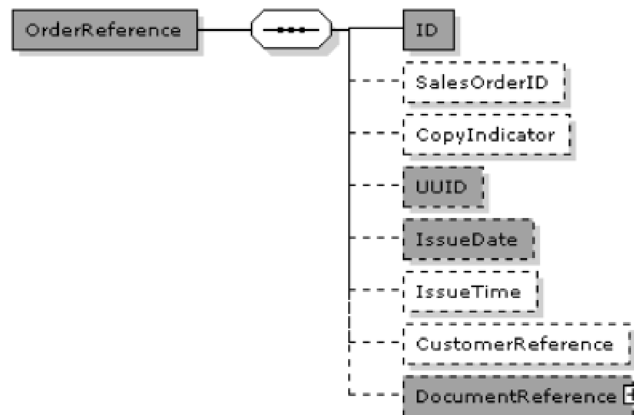

Figure 4.2: Structure of the OrderReference subclass in an OIOUBL invoice document. This shows the fields and subclasses of the class as well as the order they should be in. Classes are marked with a plus at the side. Mandatory classes and fields have full lines, while optional classes and fields have dashed lines [17].

If the fuzzer deletes the mandatory ID field, adds a new non-valid field, or changes the order of the fields in the class, it will make the document invalid according to the schema rules.

The OIOUBL documents uses namespaces to express which schema file should be used to check a specific element. XML namespaces is identified by a URI reference which can be declared with a prefix that can be used to refer to the namespace in the rest of the document [32]. In Listing 4.1 the namespace prefixes *cac* and *cbc* is used in the elements tags. All elements in OIOUBL documents has a namespace prefix. The declarations of these namespaces is presented in Listing 4.2.

```
1 <Invoice xmlns="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2" xmlns:cac="
    urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2" xmlns:
    cbc="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2"
```

Listing 4.2: Part of the namespace declaration in an example invoice document

Making changes to the namespaces or namespace prefixes, anywhere in the document, would make the document unable to be validated as the correct schema cannot be found for the fields using that namespace. We made a small test to verify that the XML parser caught the error, when one of the namespaces of a field was manually changed. This resulted in the expected parser error of the start tag

not having a matching end tag. Therefore we made OIOFuzz with a focus on not changing the namespaces.

4.5 Fuzzer Structure

In Figure 4.3 it is shown how our fuzzing harness, OIOFuzz, interacts with OIORASP. On the left side it shows the OIOUBL document being parsed to the fuzzing harness, where it is then mutated and written to the client. The client then parses and validates the document before sending it to the server over WCF (Windows Communication Foundation). The server receives the document and also parses and validates it. If that happens successfully a message stating that the response was received is returned, otherwise an error is returned. Feedback is returned from OIORASP to the fuzzing harness runner, whether that is successfully received response or an error from either the client or the server. Finally the runner sends the feedback to OIOFuzz s.t. it can be used for choosing future seeds for the fuzzing.

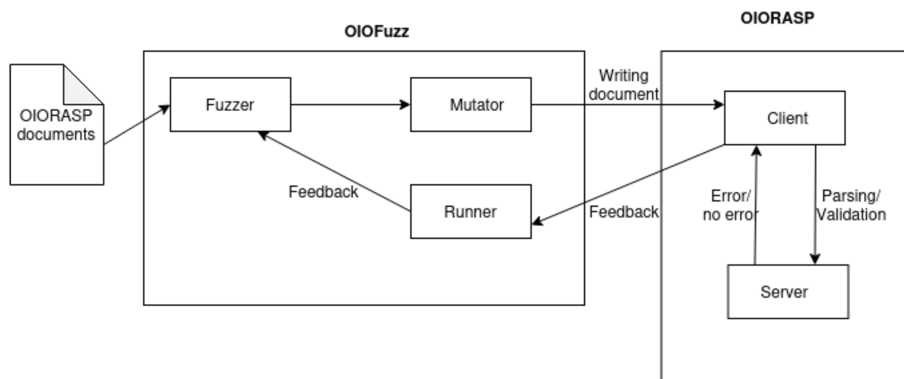

Figure 4.3: Overview of how the fuzzing harness, OIOFuzz, interacts with OIORASP.

In Figure 4.4 an overview of the structure of OIOFuzz is shown. The fuzzing harness consists of 7 components. The components are a parser, fuzzer, runner, power scheduler and a mutator consisting of a field mutator and a structure mutator. The arrows is used to show that the documents are only passed in that direction and a line indicates that the documents are passed both directions. The dotted lines indicates that the mutator uses the model when mutating the documents. The parser takes OIOUBL documents from a given input corpus and parses them to the internal format XMLtree s.t. they can be modified. Thereafter the fuzzer uses the power scheduler to choose a document from the corpus to mutate. One of the available mutators is then chosen and used to mutate the document, whereafter OIOFuzz passes the document to the runner, where the document is written to a file. At last the runner runs the ClientExample with the mutated OIOUBL

document and receives the feedback from it. When mutating a OIOUBL document the field or structure mutator can be chosen. The field mutator mutates the fields of the OIOUBL document, where it changes the values in the fields. The structure mutator makes structural mutations on the OIOUBL document, such as moving elements or making new elements with the use of the invoice model. The invoice model has the specification of all the classes and fields that can be present in an OIOUBL invoice document. The components of the fuzzing harness are described more in-depth in Section 4.6.

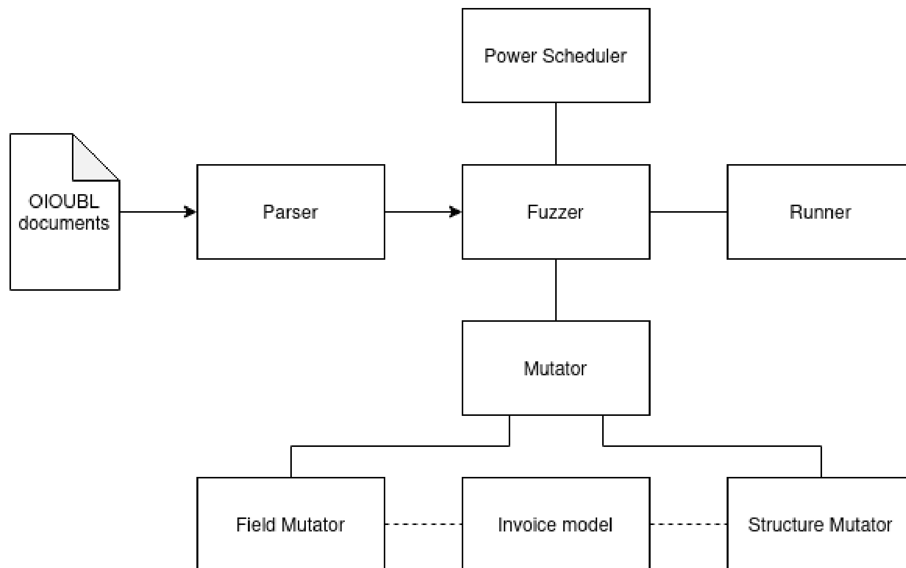

Figure 4.4: Overview of the structure of the fuzzing harness. Arrows show that the documents are only passed in that direction and a line indicates that the documents are passed both directions. The dotted lines indicates that the mutator uses the model when mutating the documents.

4.6 Components

Details on the different components of OIOFuzz are presented in this section.

4.6.1 Invoice Model

To support the fuzzing of OIOUBL documents, a model for invoice documents were constructed, based on the guideline for the invoice [17] and schemas from the OIOUBL Common Class Library [10]. This model defines structural properties and field types for the document. This could be done for other document types but given the time that was used on making the invoice as baseline this was not deemed feasible.

The dataclasses Python module is used for the model. A dataclass contains fields, consisting of a name and a type [28]. The model consists of dataclasses for the root class and all classes that can exist in that type of document, with all their fields defined in the order from the documents guideline. For subclasses the field type is that of the dataclass for the specific class. In Listing 4.3 a part of the dataclass for the invoice root element, is shown with its first four fields. The first field, UBLExtensions, is a class with the dataclass of the class as its type, while the next three are fields with the OIOUBL type Identifier, which is defined with the type str.

```
@dataclass
class Invoice():
    UBLExtensions: Optional[UBLExtensions]
    UBLVersionID: str
    CustomizationID: str
    ProfileID: str
```

Listing 4.3: Part of the Invoice dataclass

As mentioned in Section 2.3.1 fields and subclasses in an OIOUBL class can have different cardinalities; 0..1, 1, 0..n, 1..n. Fields and subclasses with a cardinality of 1 or 1..n is mandatory, as there need to exist exactly one or at least one such field in the class. Meanwhile fields and subclasses with a cardinality of 0..1 or 0..n is optional, as such fields can exist in the class, but does not have to. The Python typing Optional type is used for the optional fields, which means that it is either the type or None. The UBLExtensions class field in Listing 4.3 is an example of an optional subclass and as such has its type defined with Optional [UBLExtensions]. Fields with a cardinality of 0..n or 1..n, are defined with List for their types. Concretely this means that fields with cardinality 1..n are defined with List[type], and 0..n are defined with Optional[List[type]].

Some classes in the invoice models contains subclasses with recursive class types. The type of these subclasses is defined with a forward reference, which the mutator then has to convert to the real type at runtime. The existence of recursive defined classes, presents certain challenges and possibilities for the fuzzing harness. One challenge is to avoid spending too much time on making recursive classes for one document. Another challenge arises when writing the mutated document to a file for the ClientExample since it is very big. The possibilities consist of the capabilities of making interesting documents that might be able to trigger unexpected behavior.

The model provides the possibility of creating valid fields that are not already present in the example documents used for the initial corpus. It is also used for checking the type of a field in the Field Mutator, s.t. the mutator can choose to use operators targeting the specific type. Additionally the model can also be used

for generating new OIOUBL documents from scratch.

The challenge of avoiding that OIOFuzz spends too much time on making one document because of the recursive classes is particularly worth considering as they can be encountered many times when making new classes. When the probability of making optional classes has been set considerably high while running OIOFuzz, we have experienced it using hours on the same document because of how many classes it has constructed.

4.6.2 Parser

The parser is used to parse a OIOUBL document into an object that is modifiable in the fuzzer. The python module `xml.etree.ElementTree` is used for this. The parser has one method, `load_corpus`, which takes a path to the folder with the documents. This method is called with a folder containing selected valid files from the OIORASP repository. The documents in the folder is then parsed into `ElementTree`s and the method returns a list of these. The `ElementTree` parser is different from the XML parser in the `ClientExample`. Since XML parser comes in many different forms this could lead to a mismatch in XML parsing, which should be taken into consideration.

4.6.3 Fuzzer

The fuzzer runs a fuzzing campaign, i.e. the whole fuzzing process, and keeps track of the important things, e.g. the mutated documents, the code coverage. The important parts of the fuzzer is:

- The `multiple_runs` method runs the fuzzing campaign, with the supplied number of runs.
- The `run` method is called for each run. It starts by calling the `fuzz` method which returns a document to be sent. Then the document is sent to the `ClientExample` and it gets the code coverage and the outcome. The outcome is e.g. whether an exception occurred, a crash happened or it succeeded sending the document. The code coverage is which code blocks has been run doing the fuzzing. After getting the result the fuzzer checks if the amount of seeds in population with that outcome is less than the maximum amount of the outcome defined in the settings. If it is less, the new seed is added to the population. The population is a list of seeds consisting of the documents available to be chosen by the power scheduler.
- The `fuzz` method first provide seeds to the population by returning existing seeds. After the existing seeds has been loaded it runs the `create_candidate` to make new candidates.

- The `create_candidate` method uses a power scheduler to choose a candidate from the population. Thereafter it makes a number of mutations on the document with the chosen mutator. The number of mutations depends on the mutation count defined in the settings. It also checks if the seed has been chosen more times than the amount defined for a replace count in the settings, in which case it is removed from the population.
- The Power Scheduler assigns energy to each seed based on how interesting they are based on some criteria. The concept of a power scheduler is described in Section 3.2.4.

4.6.4 Mutator

The mutator consists of two separate mutators, where one is chosen at random when mutating the OIOUBL documents. The `Field Mutator` makes mutations to the text of the documents fields, while the `Structure Mutator` makes structural mutations to the document. A XML tree structure of part of an OIOUBL document is presented in Figure 4.5. This XML tree will be used as a example for some of the mutation operators.

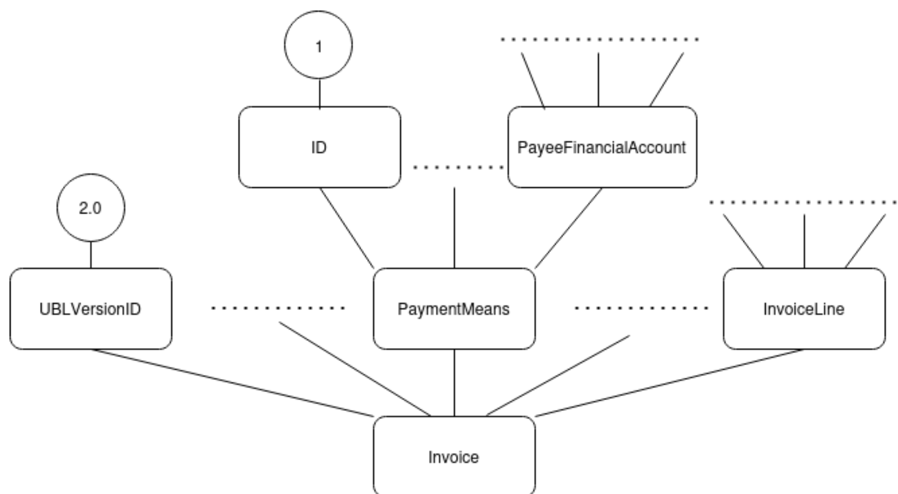

Figure 4.5: OIOUBL document as a XML tree. The boxes are classes and fields and a line shows that they are element of that class or field. The circles is the text in a field.

Field Mutator

The `Field Mutator` mutates the text in a field, e.g. if the field is the UBLVersionID shown in Figure 4.5 it will mutate the string `2.0`. It chooses a random field in the document to make the mutations on.

The mutation operators is categorized as string operators and interesting float operators. The reason for the categorization of float operators are that the datatype used for all fields with numeric values are floats, based on exception returned from the client when treating all fields as strings. One of the string mutators is `replace_string_mutator` which generates a whole new string to replace the old text. Additionally the mutator has three operators for substrings and three operators for singular characters. These operators respectively replace, delete, or add a substring or character at a randomly chosen index of the original text string.

The field mutator has a probability of taking the type of the Field into consideration, which is set in the settings. When it does the the values for the types are generated from scratch. If the mutators does not take it into consideration the string operators are used since all fields text can be considered as strings.

Structure Mutator

The structure mutator consist of operators to make structural changes to the OIOUBL documents. These operators are used to duplicate, delete, move, and add new fields. The mutation starts in its `mutate` method, where one of the operators is randomly chosen. If the chosen operator is not the `add_field` method that creates a new field, an element in the document is chosen at random and the chosen operator is called with the element and its parent element as input parameters. The parent element cannot be accessed from the child `Element`. Therefore a mapping between all elements and their parent elements is created and it is used when calling the mutation operators.

The structure mutator has four mutations operators:

- The `duplicate_field` method duplicates the given element at the same position in the document with a probability specified in settings. The rest of the time it inserts the duplicate element at a random position in the whole document.
- The `delete_field` method removes the given element from the parent element.
- The `move_field` method first removes the element from the document. Thereafter it inserts the element back in the document at a new position.
- The `add_field` method creates a new valid element from the type of document and inserts it in the document. To create a valid element it uses the model for the OIOUBL document type. A random field for the document type is chosen and made with the mutators `make_field` method. If the field is not a subclass, `make_field` creates a random element of the correct type for the field. If it is a subclass, it makes an element for the class and

subelements for all its fields, which is found using the dataclass for the class. The element is then inserted either at the correct position, as specified in the model, or at a random position in the document, based on a probability defined in the settings.

An example of a mutation done with the structure mutator is visualized in Figures 4.6 and 4.7 with an OIOUBL document as an XML tree. In Figure 4.6 the `PayeeFinancialAccount` class is highlighted as it is chosen to be mutated with the `move_field` method. In Figure 4.7 the class has been deleted from the `PaymentMeans` class and inserted into the `InvoiceLine` class.

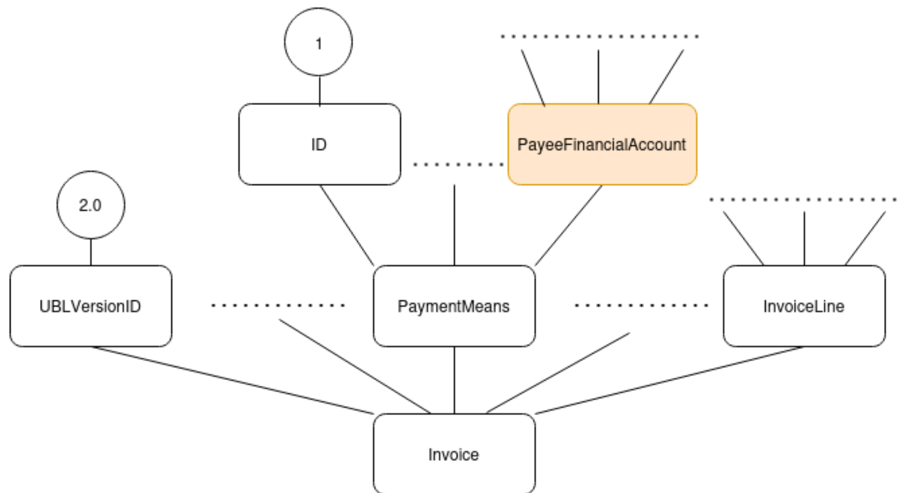

Figure 4.6: Part of an OIOUBL document as an XML tree, with the Invoice class as the root. The boxes is classes and fields and the circles is the field text. The box in orange is the chosen class to be mutated

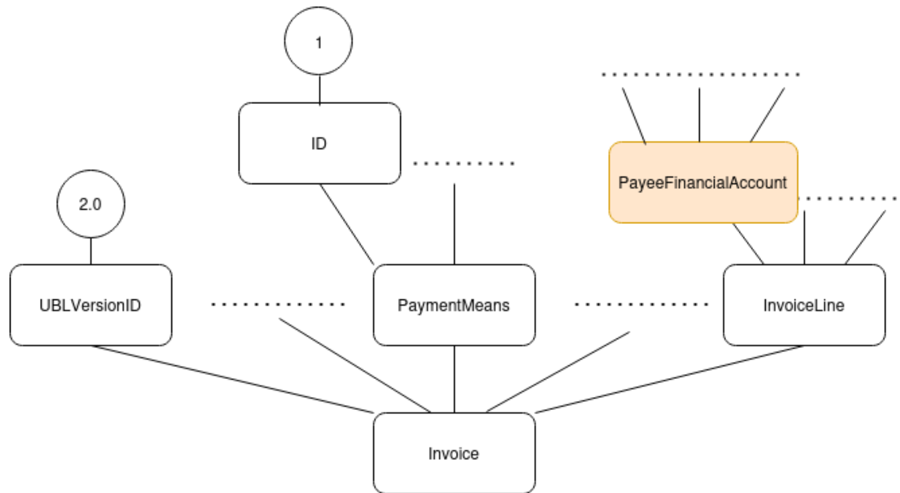

Figure 4.7: Part of an OIOUBL document as an XML tree, with the Invoice class as the root. The box in orange is the class after being moved.

4.6.5 Runner

The runner facilitates the communication between the fuzzer and the `ClientExample`. It runs the `ClientExample` with the fuzzed documents, and handles the output from the `ClientExample` as well as potential crashes. It has a `run` method which gets the fuzzed document as an `ElementTree` and writes it to a file whose path is passed to the `ClientExample` as an argument. Thereafter it runs its `start_process` method and returns the result to the fuzzer. `start_process` runs the `ClientExample` as a subprocess with a 30 second timeout. It then calls the method `handle_feedback` with the standard output of the `ClientExample`. `handle_feedback` finds all the manually instrumented code blocks in the standard output, and saves the code coverage for the fuzzer. It also finds any exceptions that has occurred in the `ClientExample` from the standard output.

4.7 Classification of OIOFuzz

We classify OIOFuzz as a guided model-based blackbox. It is guided since it uses the output of the `ClientExample` to guide the fuzzing process. Despite it having code coverage information it is not used for anything, as it did not provide anything more than looking at the output, thus making it a blackbox fuzzer. It is model-based since it uses a model to generate part of OIOUBL documents and to guide the fuzzing process. Additionally it is a mutation-based fuzzer, but contains functionality for generation-based fuzzing.

Chapter 5

Fuzzing OIORASP

In this chapter we will present experiments conducted to test OIOFuzz as well as results of them. Experiments was done in iterations and not only on the fully developed fuzzing harness.

An explanation of some exploration conducted early in the fuzzing process and the outcome of it is given in Section 5.1. Then an overview of running OIOFuzz with a basis setup and with some different variations where the setup is changed is provided in Section 5.2. Lastly in Section 5.3 an error found in the Schematron validation is analyzed and described.

5.1 Initial Exploration

An initial version of the fuzzing harness was made, to conduct early exploration of the result of fuzzing OIORASP. The only mutation operators contained in the initial fuzzing harness were simple string operators in the `Field Mutator`. Instead of choosing a random field to mutate, the `Field Mutator` looped through all fields in the document and chose whether to mutate the field, with a probability of 15% of doing so. It was set to 15% to mutate a few fields, since mutating too many would cause the validator to catch the same error every time. The exploration consisted of running a few experiments with the example invoice document in Appendix A as initial corpus, where the output was observed after sending the document with the `ClientExample`. Observation of the output were accomplished by first logging exceptions that occurred in the `ClientExample` along with the document that triggered it. Thereafter it was manually inspected what triggered the exception.

The most common exception encountered in the early exploration was a schema exception stating that the value of a field is not a valid value for its type, e.g. decimal, boolean, or date. The commonness of this exception was not completely unexpected, as an OIOUBL document field has a specific datatype, while the mutator

treated all field texts as strings. However, it does showcase that blindly mutating the field text without taking regard of the datatype limits the exploration.

Another common exception encountered was an exception specifying that the type of the document could not be found from the XML document. Inspection of the source code showed that it occurred in the method `FindUniqueDocumentType`, which is called earlier than the validation of the document with schemas and Schematron. Inspecting the documents that was logged when the exception occurred, showed that it happened when the invoice field CustomizationID had been mutated. Another small experiment for further exploration of this exception was also conducted. In this experiment only the CustomizationID field was mutated. During this experiment an exception stating that the Schematron validation of the document failed would sometimes occur instead. The Schematron exception message specified that the CustomizationID field text must be OIOUBL-2.01, OIOUBL-2.02 or OIOUBL-2.1. Further inspection showed that the Schematron exception occurred when text was appended at the end of the string, e.g. OIOUBL-2.0192. This indicates that `FindUniqueDocumentType` checks for the values with prefix checking rather than checking the exact values. This inaccuracy does not present any real bug, but is rather just a whimsical implementation.

5.2 Running OIOFuzz

In this section we present details about running OIOFuzz and observations made during the process. First some observations made during early iterations of running OIOFuzz is presented in Section 5.2.1. Thereafter the general setup for running OIOFuzz is presented in Section 5.2.2, and different variations of the setup explored are described Section 5.2.3.

5.2.1 Observations From Early Iterations

In our early iterations of running OIOFuzz with the mutators mentioned in Section 4.6.4 we only found a few different errors. This is because it only added one of each kind of error to the population making the population very stale. To update the population while running we chose to delete a seed after it has been sent ten times. We also chose to keep track of how many documents causing the different exceptions were in population. This was done as some of the exceptions occur more frequently that others, like XML parser errors, or schema errors, and we wanted to avoid that the population was filled with these. Therefore a maximum was set for all the different outcomes to limit these in the population.

5.2.2 Experiment Setup

In this section we will describe the setup of the experiments conducted with the OIOFuzz.

The initial corpus contains 14 invoice example documents from the OIORASP library, providing OIOFuzz with an initial corpus consisting of a small number of valid documents to fuzz. The differences between the documents is mainly that they have different profiles and are identified in different ways. The different identification and respective profiles is e.g. CVR number, D-U-N-S number, and EAN.

The experiments uses settings for different variables dictating the fuzzing process. The variables each have a base value. The different settings, along with their base values and a short explanation of their purpose, is presented in Table 5.1.

Name	Base value	Description
MUTATION_COUNT	5	Maximum number of mutations made before sending.
PLACEMENT_PROB	95%	Probability that a field is inserted at a specific place. Else it is inserted some random place in the document.
OPT_PROB	50%	Probability of an optional field being made
MAX_RECUR_DEPTH	30	Maximal recursion depth when creating invoice classes
TYPE_PROB	80%	Probability of mutating a field with regards to its correct type
REPLACE_COUNT	10	The number of times a seed is sent before it is getting replaced

Table 5.1: Variables in the settings for OIOFuzz, with a base value and a short description.

The MUTATION_COUNT determines the maximal amount of mutations made to the chosen document before it is sent to the client. Its base value is set to 5 to keep the amount of mutation made low, since a high amount of mutations will cause drastic changes to the document causing OIOFuzz to potentially missing out on exploring parts of the target. PLACEMENT_PROB is used in the `duplicate_field` and `add_field` operators in the structure mutator. It is used to determine whether a field should be inserted at a specific place or some random place in the document. For the `duplicate_field` operator the specific place is at the same position in the document as the field being duplicated, and for the `add_field` operator the place is the correct place in the document, as specified by the schema rules. Its base value is set to 95% to guide it towards making more

schema correct documents, while still providing some opportunity to explore mutations that will most likely make non schema correct documents. Despite mainly focusing on Schematron validation, we see a value in still letting OIOFuzz explore making such mutations. OPT_PROB determine the probability of a field that is optional for a class being made in the `add_field` operator. MAX_RECUR_DEPTH is the maximal recursion depth of nested classes being made with the `add_field` operator. Their base values is set to 50% and 30 respectively, chosen to ensure that OIOFuzz does not use too much time on constructing those documents. REPLACE_COUNT defines the amount of times a document is chosen as the fuzzing seed before being replaced. Its base value is set to 10. Values for maximum amount of seeds leading to a specific outcome is presented in Table 5.2.

Schema	Schematron	Unknown	Fail	XML	Pass
5	15	50	50	5	5

Table 5.2: Maximum amount of documents giving specific outcomes allowed in the population. The outcome describes what category of exceptions the output belongs to. Fail refers to ClientExample crashes, XML refers to xml parsing exceptions, and Pass refers to the document being successfully sent.

The outcome for schema error, XML parser error and Pass was set to 5 as those is not the target of the fuzzing and we therefore want to avoid that OIOFuzz spend too much time exploring these. The outcome Schematron error is set to 15 as those are the target, while the outcome with unknown errors or where the ClientExample failed is set to 50 since they are important to explore further.

5.2.3 Different Variations

Various variations of the setup was used for experiments with OIOFuzz, where some of the settings values is changed. This was generally done by increasing or decreasing a few or all of the values for a variation. However increasing the values of OPT_PROB and MAX_RECUR_DEPTH results in more classes being created with the `add_field` operator, causing creation of big documents and a high runtime for singular iterations. As such these values were only decreased in the standard variations and and special variation aimed to test the protocol ability to handle big document was set up. The different variations did not result in any new interesting behavior.

Constructing Big Documents

An experiment was setup where OIOFuzz was guided towards constructing bigger documents, to test whether the `ClientExample` is able to properly handle those. This was done by increasing the value of OPT_PROB and MAX_RECUR_DEPTH,

to 70% and 50 respectively. With these settings OIOFuzz managed to construct documents that was 707.000 KB and where OIOFuzz had generated 2057813 new classes in the document with the `add_field` operator. In comparison the example documents are normally 9 KB. However, other than taking more time, this did not cause any new unexpected behavior.

5.3 Schematron Error

An error in the OIORASP Schematron was found with OIOFuzz, which cause the Schematron validation of the document to fail. The validation fails because of a type error, stating that a sequence of more than one item is not allowed as the argument of `fn:string-length()`. The error occurred when OIOFuzz duplicated the field PaymentNote in the class PayeeFinancialAccount. According to the documentation, multiple instances of the PaymentNote field is allowed in the PayeeFinancialAccount class. This can be seen in Table 5.3 which contains the most commonly used fields and classes in the PaymentMeans class, which has the PayeeFinancialAccount class as its subclass. The row for the PaymentNote field in the PayeeFinancialAccount class has been marked with bold text in the table. However, the implementation of the Schematron does not take this into account, causing the validator to try checking the length of a string on a list of strings.

UK-name	DK-name/DK-Alternativ term	Use
ID	BetalingsMådeNummer	0..1
PaymentMeansCode	BetalingsMådeKode	1
PaymentDueDate	BetalingsDato	0..1
PaymentChannelCode	BetalingsKanal	0..1
InstructionID	BetalingsID	0..1
InstructionNote	LangAdvisering	0..1
PaymentID	KortArtsKode	0..1
PayerFinancialAccount / ID	Kontonummer	0..1
PayerFinancialAccount / PaymentNote	KortAdvisering	0..n
PayerFinancialAccount / FiBranch / ID	Registreringsnummer	0..1
PayerFinancialAccount / FiBranch / FinancialInstitution /ID	BankID	1
PayeeFinancialAccount / ID	Kontonummer	1
PayeeFinancialAccount / PaymentNote	**BetalingsNote**	**0..n**
PayeeFinancialAccount / FiBranch / ID	Registreringsnummer	0..1
PayeeFinancialAccount / FiBranch / Name	BankFilialNavn	0..1
PayeeFinancialAccount / FiBranch / FinancialInstitution /ID	BankID	1
PayeeFinancialAccount / FiBranch / Address / *	BankFilialAdresse	0..1
CreditAccount / AccountID	KreditorNummer	1

Table 5.3: The PaymentMeans OIOUBL class with its most commonly used fields and subclasses. The two first column show the name of the fields in English and Danish respectively and the last column shows the allowed cardinality of the field [18]. The PaymentNote field in the PayeeFinancialAccount subclass has been marked with bold text.

The Schematron causing the validation to fail is shown in Listing 5.1, specifically the $string - length(cac : PayeeFinancialAccount/cbc : PaymentNote)\>20$ check. The rule also checks if the PaymentMeansCode field in the PaymentMeans class has a value of 42, which it has in the used example document. The PayeeFinancialAccount with a duplicated PaymentNote field is shown in Listing 5.2.

```
1 <xsl:if test="(cbc:PaymentMeansCode = '42') and string-length(cac:
    PayeeFinancialAccount/cbc:PaymentNote)&gt; 20">
2    <Error>
3       <xsl:attribute name="context">
4          <xsl:value-of select="concat(name(parent::*),'/',name())"/>
5       </xsl:attribute>
6       <Pattern>(cbc:PaymentMeansCode = '42') and string-length(cac:
            PayeeFinancialAccount/cbc:PaymentNote)&gt; 20</Pattern>
7       <Description>[F-LIB133] PaymentMeansCode = 42, PaymentNote must be no more
```

```
            than 20 characters</Description>
8       <Xpath>
9         <xsl:for-each select="ancestor-or-self::*">/<xsl:value-of select="name()"/>
              [<xsl:value-of select="count(preceding-sibling::*[name(.)=name(current
              ())])+1"/>]</xsl:for-each>
10      </Xpath>
11    </Error>
12 </xsl:if>
```

Listing 5.1: Schematron rule for the length of the string in PaymentNote

```
1 <ns3:PayeeFinancialAccount>
2   <ns2:ID>1234567890</ns2:ID>
3   <ns2:PaymentNote>A00095678</ns2:PaymentNote>
4   <ns2:PaymentNote>A00095678</ns2:PaymentNote>
5   <ns3:FinancialInstitutionBranch>
6     <ns2:ID>1234</ns2:ID>
7   </ns3:FinancialInstitutionBranch>
8 </ns3:PayeeFinancialAccount>
```

Listing 5.2: PayeeFinancialAccount class with a duplicated PaymentNote field

This error occurs as an assert error when the `ClientExample` is compiled in debug version and a popup windows comes up. It is an unhandled assertion, indicating that it is an unexpected error. The reason that it is unexpected is that the document has already been validated with schema and should therefore be guaranteed to be of the correct format to be validated with Schematron. This is a strange implementation since assert should only be used validate internal stuff and not be used on the input of a user, such as the OIOUBL document we try to send with the `ClientExample` [1]. When `ClientExample` is compiled in release version the assert error does not occur and instead only the exception stating that the document could not be Schematron validated occurs.

Additionally it was observed that three other similar Schematron rule exists. One of them is where the only difference in the rule is that the PaymentMeansCode fields value is checked if it is 31 instead of 42. The other two are for the PayerFinancialAccount, which is also a subclass of the PaymentMeans class, instead of the PayeeFinancialAccount and has the same check for values of the PaymentNote field. It can be seen in Table 5.3 that multiple occurrences of the PaymentNote field in the PayerFinancialAccount class is also allowed. To verify that the same error occur with these rules a PayerFinancialAccount with a PaymentNote field was added to the example document, as the class is not present in the document. Then a document for each combination of values and duplication of the PaymentNote field was sent with the `ClientExample`. As expected all these lead to the same error appearing. However OIOFuzz never managed to find the other errors when we ran it. The reason for this is that it would need to make some very specific mutations to trigger them, which is extremely unlikely without some way to guide it

towards those mutations. The PaymentMeansCode field already has a value of 42 in the example documents and a PayeeFinancialAccount exist with a PaymentNote field, meaning that the document were only one mutation away from finding the error it found, i.e. duplicating the PaymentNote field. Meanwhile to trigger the other errors it would have to either change the value of the PaymentMeansCode field to exactly 31, add a PayerFinancialAccount class and thereafter duplicate its PaymentNote field or do both of those things.

Chapter 6

Related Work

In this chapter related work to fuzzing XML and Schematron is presented.

Peach fuzzer

Peach fuzzer is a smart fuzzer that can use both generation- and mutation-based fuzzing [21]. It uses its own custom files, called Peach Pit files, to define the structure and type information of the files that is to be fuzzed. Peach provides a fuzzing engine with robust monitoring capabilities, where the user can specify their own fuzzing strategy and mutators. It is commonly used to fuzz file formats, network protocols, and APIs. Peach has been used in the paper "Model-based whitebox fuzzing for program binaries" [27], where it is used to model the target file format, to generate new chunks and to integrity check the fuzzed documents.

Skyfire

Skyfire is a data-driven seed generation fuzzer [36]. It targets highly structured files such as XML and Schematron files. The authors of the paper collects a high amount of samples and their corresponding grammar to extract their semantics rules and the frequency of production rules. Skyfire then use these to learn a probabilistic context grammar (PCSG) for the model which is used to generate well-distributed seeds. These seeds are then used to fuzz several open-source XSLT and XML engines. They tested generating seeds with Skyfire + AFL against AFL on crawled seeds. The result was that they increased the line code coverage with 20% and function coverage with 15%. They found 19 new memory corruption bugs and 32 denial-of-service bugs.

Superion

Superion is a grammar-aware greybox fuzzer [37]. It builds upon AFL and targets more structured input like XML. Using a grammar for the file formats it makes an abstract syntax tree (AST) of the parsed inputs. They introduce two grammar-aware mutation strategies, with one being a tree-based mutation. Tree-based mutations replaces subtrees in test input AST with another subtree from another test input AST. They tested the program on one XML engine and three JavaScript engines and compared their results against AFL and jsfunfuzz [19] which is a grammar-aware fuzzer. Compared to those they improved 16,7% on line code coverage and 8,8% on function coverage. They also found 21 new vulnerabilities.

Chapter 7

Discussion

In this chapter we discuss different aspects of our fuzzer OIOFuzz. While it managed to find an error in the Schematron validation, we believe it has certain deficiencies. We will discuss the deficiencies and how they could be improved.

First in Section 7.1 we discuss our instrumentation of the ClientExample and how this could be improved. Then in Section 7.2 we present the idea of making our fuzzer in C# instead and the benefits that could archive. Thirdly we discuss how we did not take the Schematron file more into consideration and how a Schematron guided fuzzer would look like in Section 7.3. At last in Section 7.4 we discuss making OIOFuzz fully generation-based instead of being mutation-based with elements of generation-based.

7.1 Instrumentation of the ClientExample

We instrumented the `ClientExample` as part of our early exploration, as this seemed beneficial. In the end it did not help very much, since the `ClientExample` covered the same few branches of code, when given the documents constructed by OIOFuzz. It therefore cannot be used to guide the OIOFuzz anymore than the output of the `ClientExample`, since those branches are just different exceptions thrown or a successful sending. Instrumenting the `ClientExample` was also mostly done on the called functions, calls to other libraries and its exception handling.

In order to gain more information from the instrumentation, the compiled dll files from the OIORASP library and other libraries they use, such as Saxon, could also have been instrumented. This would give us more information about what happens when those libraries is called. We could potentially have used Scharpfuzz or WinSharpFuzz [22] which is a fork of Scharpfuzz, as instrumenting dll is part of their fuzzing process. Since we are focusing on Schematron, this could give us more information of what happens in Saxon when it process the Schematron file and validates the document. However it might not give enough information to be

really beneficial, since Saxon is an engine that compiles Schematron to validate it and it would not give information on which Schematron rules are checked.

7.2 C# Fuzzer

Instead of writing OIOFuzz in Python it could have been written in C#. This could have provided the benefit of not having to write the document to a file before sending it to the `ClientExample`, which slows the fuzzing process down. One way of ensuring this could be to incorporate the `ClientExample` in our fuzzer, where the fuzzer would parse the document, potentially using the XML parser from the `ClientExample`, fuzz it and then pass it to the `ClientExample` as a object or string. As the `ClientExample` is setup to parse a document at a specific file path, how it is processed would have to be altered s.t. it receives the document directly instead.

7.3 Schematron Guided Mutations

OIOFuzz was built around the idea of wanting to take advantage of the example documents provided with the library. Therefore OIOFuzz was made as a mutation-based fuzzer, where the mutator was designed with a focus on the different components of the example documents i.e. classes, fields and text in fields. The target was narrowed in to the Schematron validation, the focus on this target was achieved by guiding OIOFuzz towards making more documents that are valid according to the schema rules. However making mutation based on the Schematron for the document types could have been more beneficial, since the mutation would then be more focused on different Schematron rule and be capable of exploring a higher number of them.

A potential way of doing so is making a mechanism for guiding OIOFuzz to choose mutations based on Schematron rules. One way that such mechanism can be envisioned to function is where a Schematron rule is first randomly chosen. Thereafter it would identify which fields are contained in the rule and mutate those. If a field that are contained in the rule does not exist in the document, the field would have to be made along with all non existing ancestor classes. Furthermore which mutations is made could be based on the constraints specified in the rule. As an example consider the Schematron rule in Listing 7.1. The rule is defined inside a template associated with the `PaymentMeans` class in the `Invoice` document, meaning that it involves classes and fields from the `PaymentMeans` class. On line 1 in the listing it can be seen that the contained fields are `PaymentMeansCode` and `ID` in the `PayeeFinancialAccount` subclass. As such OIOFuzz would then identify these fields as targets for the mutations. Furthermore if it were to take the constraints in the rule into account, it could e.g.

determine that the `PaymentMeansCode` fields needs to have a value of 31 before this rule can be broken and then set the value of the field to that.

```
1 <xsl:if test="(cbc:PaymentMeansCode = '31') and not(cac:PayeeFinancialAccount/cbc:ID
    )">
2   <Error>
3     <xsl:attribute name="context">
4       <xsl:value-of select="concat(name(parent::*),'/',name())"/>
5     </xsl:attribute>
6     <Pattern>(cbc:PaymentMeansCode = '31') and not(cac:PayeeFinancialAccount/cbc:ID
        )</Pattern>
7     <Description>[F-LIB107] PaymentMeansCode = 31, ID element is mandatory</
        Description>
8     <Xpath>
9       <xsl:for-each select="ancestor-or-self::*">/<xsl:value-of select="name()"/>[
            <xsl:value-of select="count(preceding-sibling::*[name(.)=name(current())
            ])+1"/>]</xsl:for-each>
10    </Xpath>
11  </Error>
12 </xsl:if>
```

Listing 7.1: Schematron rule for the length of the string in PaymentNote

A proper version of such a mechanism would require more analysis of the Schematron rules and how to more beneficially make mutations based on them. Furthermore it might need the functionality to solve the constraints in the rules to give enough of a benefit to OIOFuzz, which would give a higher computational overhead.

7.4 Generation-based Fuzzer

OIOFuzz was made as a mutation-based fuzzer with elements of generation, through the `add_field` operator, which generates new fields and subclasses. It could alternatively have been made as a fully generation-based fuzzer, generating the OIOUBL document from scratch. Using the invoice model it is already possible to generate whole invoice document. However it is almost impossible for those documents to be valid or close to valid, as the model lacks consideration of field attributes and exact values of field specified by code listings. While the aim is not to generate valid document it is preferred that they can be close to valid, otherwise they would always be caught early by the schema validation. To generate documents that are closer to being valid the model needs to handle the field attributes and consider allowed values as specified in the code listings. Handling those elements would require a more in depth analysis of them.

Having a fully generation-based fuzzer would potentially enable a broader exploration of the Schematron rules, as more fields and classes would be generated with the their correct attributes and values from code listings. However the

mutation-based fuzzing harness was a suitable approach since we had the provided example documents.

Chapter 8

Conclusion

We aimed to explore the potential of fuzz testing the OIORASP protocol, as this is a key part of the Danish IT infrastructure. For the exploration we first studied the OIORASP protocol and described the overall structure of the protocol and the OIOUBL documents that are sent with the protocol. Then theory on fuzzing was studied, and different fuzzing concepts were presented. Thereafter we delimited the target into the Schematron validation of the OIORASP protocol. A fuzzing harness OIOFuzz was set up and the structure and implementation of it were described. Different experiments of using OIOFuzz on the OIORASP protocol were set up and run.

We made a proof of concept implementation of a guided model-based black-box fuzzer targeting OIORASPs Schematron validation, and managed to find an error in the Schematron validation with it. However, as we discussed in Chapter 7 it lacked some features to be better capable of finding more errors than it did. In particular more features for guiding OIOFuzz to broader exploration of the Schematron rules are desired, as it would give the most benefits.

Based on all this we conclude that this project was a first step in fuzzing the OIORASP protocol and showcased its potential of finding bugs. With further development there is potential for finding more bugs, as the developed fuzzing tool has areas where possible improvements can be made.

Chapter 9

Future Work

While OIOFuzz were able to fuzz OIORASPs Schematron validation and managed to find an error in it, it is also clear that it has room for improvement. In this chapter we will present the future work deemed as the most beneficial for improving OIOFuzz.

Expanding the Generation of Elements

An area of OIOFuzz that should be expanded upon is the generation of new elements using the document model. The implemented invoice model lacks handling of field attributes and consideration of the allowed values specified by the code listings. Expanding the model to handle those would lead to a broader exploration of the Schematron rules. The broader exploration is a result of new elements being generated with some of these values and attributes, as a lot of the Schematron rules check for these values and attributes. With such expansion OIOFuzzs potential for finding errors in the Schematron validation phase would increase.

Models for More Document Types

As described earlier the only document type we made a model for was the invoice document type. This is only one of the 15 different types of OIOUBL business document. For a full test of OIORASPs Schematron validation models should also be made for the rest of the document types. As described in Section 4.6.1 it took a considerable amount of time to manually make the first model by looking through OIOUBL guidelines for invoice documents [17]. Therefore the future work of making models for all the other document types would preferably be made with an automated process. This could be done by using the schema files OIOUBL uses to validate the documents. The important schema files for this task are UBL-CommonAggregateComponents-2.1.xsd and the schema file for the document type. The UBL-CommonAggregateComponents-2.1.xsd contains the defini-

tion of all classes in all document types. The schema files can found in the common folder in the OIORASP library. Making these models would get us more access to more example documents which OIOFuzz could handle with the partial generation. We could also possible find more Schematron errors since each document type has its own Schematron file to validate it.

Bibliography

[1] Casey Casalnuovo et al. "Assert Use in GitHub Projects". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 755–766. DOI: 10.1109/ICSE.2015.88.

[2] Chimezie Ogbuji. *Validating XML with Schematron*. https://www.xml.com/pub/a/2000/11/22/schematron.html. Accessed: June 16, 2023. 2000.

[3] Digitaliseringsstyrelsen. *It-løsninger - Danmarks digitale infrastruktur*. https://digst.dk/it-loesninger/. Accessed: June 16, 2023. 2021.

[4] Digitaliseringsstyrelsen. *Mål- og resultatplan*. https://digst.dk/om-os/om-digitaliseringsstyrelsen/. Accessed: June 16, 2023. 2021.

[5] Digitaliseringsstyrelsen. *National strategi for cyber- og informationssikkerhed*. https://fm.dk/media/25359/national-strategi-for-cyber-og-informationssikkerhed_web-a.pdf. Accessed: June 16, 2023. 2021.

[6] Digitaliseringsstyrelsen. *Om Digitaliseringsstyrelsen*. https://digst.dk/om-os/om-digitaliseringsstyrelsen/. Accessed: June 16, 2023.

[7] Digitaliseringsstyrelsen. *Vejledning til model for porteføljestyring af statslige it-systemer*. https://digst.dk/media/28086/vejledning-til-model-for-portefoeljestyring-af-statslige-it-systemer-april-2022.pdf. Accessed: June 16, 2023. 2021.

[8] Emil F.L Aagreen, Frederik A Jensen, Mikkel T Jensen, Tobias B.S Hansen. *Towards Verification of the OIORASP Protocol*. https://projekter.aau.dk/projekter/files/512993360/cs_22_ds_9_05.pdf. Accessed: June 16, 2023. 2022.

[9] Erhvervsstyrelsen. *En teknisk introduktion til NemHandel*. https://nemhandel.dk/vejledning-en-teknisk-introduktion-til-nemhandel. Accessed: June 16, 2023.

[10] Erhvervsstyrelsen. *OIORASP Repository*. https://rep.erst.dk/git/openebusiness/library/dotnet. Accessed: June 16, 2023.

[11] G. Ken Holman. *Universal Business Language Version 2.2*. http://docs.oasis-open.org/ubl/os-UBL-2.2/UBL-2.2.html. Accessed: June 16, 2023.

[12] Patrice Godefroid. “Fuzzing: Hack, Art, and Science”. In: *Commun. ACM* (2020), 70–76. DOI: 10.1145/3363824. URL: https://doi.org/10.1145/3363824.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. “Grammar-Based Whitebox Fuzzing”. In: Association for Computing Machinery, 2008. ISBN: 9781595938602.

[14] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[15] Hans-Henrik Busk Stie, Karen Wiis Weiss. *Dansk firma lagt ned af russiske hackere - er gået i højeste kriseberedskab*. https://nyheder.tv2.dk/samfund/2022-03-02-dansk-firma-lagt-ned-af-russiske-hackere-er-gaaet-i-hoejeste-kriseberedskab. Accessed: June 16, 2023. 2022.

[16] IT- og Telestyrelsen. *OIOUBL Intro*. https://www.oioubl.info/documents/en/en/Intro/OIOUBL_INTRO.pdf. Accessed: June 16, 2023. 2007.

[17] IT- og Telestyrelsen. *Online OIOUBL Dokumentation - Invoice*. https://www.oioubl.info/documents/en/en/Dokument/oioubl_guide_faktura.pdf. Accessed: June 16, 2023. 2007.

[18] IT- og Telestyrelsen. *Online OIOUBL Dokumentation - Payment means and Payment terms*. https://www.oioubl.info/documents/en/en/Guidelines/OIOUBL_GUIDE_PAYMENT.pdf. Accessed: June 16, 2023. 2015.

[19] Jesse Ruderman. *Introducing jsfunfuss*. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. Accessed: June 16, 2023.

[20] Matt. *A guide to fuzz testing*. https://testfully.io/blog/fuzz-testing/. Accessed: June 16, 2023.

[21] Michael Eddington. *Peach Fuzzer*. https://peachtech.gitlab.io/peach-fuzzer-community/. Accessed: June 16, 2023.

[22] Nathaniel Bennett. *WinSharpFuzz: Coverage-based Fuzzing for Windows .NET*. https://github.com/nathaniel-bennett/winsharpfuzz. Accessed: June 16, 2023.

[23] Nets DanID A/S. *CPS - Certification Practice Statement*. https://www.nets.eu/dk-da/kundeservice/NemID-Til-Private/Documents/CPS_3.1.pdf. Accessed: June 16, 2023.

[24] Peppol. *Peppol BIS Specifications – An Overview*. https://peppol.eu/what-is-peppol/peppol-profiles-specifications/. Accessed: June 16, 2023.

[25] Peppol. *Peppol eDelivery Network – An Overview*. https://peppol.eu/what-is-peppol/peppol-transport-infrastructure/. Accessed: June 16, 2023.

[26] Peppol. *What is Peppol?* https://peppol.eu/what-is-peppol/. Accessed: June 16, 2023.

[27] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "Model-Based Whitebox Fuzzing for Program Binaries". In: ASE '16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553. ISBN: 9781450338455. DOI: 10.1145/2970276.2970316. URL: https://doi.org/10.1145/2970276.2970316.

[28] Python. *dataclasses — Data Classes*. https://docs.python.org/3/library/dataclasses.html. Accessed: June 16, 2023.

[29] Saxonica. *What is Saxon?* https://www.saxonica.com/html/documentation10/about/whatis.html. Accessed: June 16, 2023.

[30] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. https://www.w3.org/TR/2008/REC-xml-20081126/. Accessed: June 16, 2023. 2008.

[31] W3C. *Extensible Stylesheet Language (XSL) Version 1.1*. https://www.w3.org/TR/xsl/. Accessed: June 16, 2023.

[32] W3C. *Namespaces in XML 1.0 (Third Edition)*. https://www.w3.org/TR/xml-names/. Accessed: June 16, 2023. 2009.

[33] W3C. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. https://www.w3.org/TR/xmlschema11-1/. Accessed: June 16, 2023. 2012.

[34] W3C. *What is XSL?* https://www.w3.org/Style/XSL/WhatIsXSL.html. Accessed: June 16, 2023. 2021.

[35] W3C. *XSL Transformations (XSLT) Version 2.0 (Second Edition)*. https://www.w3.org/TR/2021/REC-xslt20-20210330/. Accessed: June 16, 2023. 2021.

[36] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 579–594.

[37] Junjie Wang et al. "Superion: Grammar-Aware Greybox Fuzzing". In: *CoRR* abs/1812.01197 (2018).

[38] WireShark. *WireShark*. https://www.wireshark.org/. Accessed: June 16, 2023.

[39] Zalewski, Michał. *Technical whitepaper for afl-fuzz*. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: June 16, 2023. 2014.

[40] Andreas Zeller et al. *The Fuzzing Book*. Retrieved 2021-10-26 15:30:20+02:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/.

Appendix A

OIOUBL Invoice Document

OIOUBL Invoice example document, used as one of the document in the initial seed corpus

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Invoice xmlns="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2" xmlns:cac="
    urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2" xmlns:
    cbc="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2" xmlns
    :ccts="urn:oasis:names:specification:ubl:schema:xsd:CoreComponentParameters-2"
    xmlns:sdt="urn:oasis:names:specification:ubl:schema:xsd:SpecializedDatatypes-2"
    xmlns:udt="urn:un:unece:uncefact:data:specification:
    UnqualifiedDataTypesSchemaModule:2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:schemaLocation="urn:oasis:names:specification:ubl:schema:xsd:
    Invoice-2 UBL-Invoice-2.0.xsd">
3   <cbc:UBLVersionID>2.0</cbc:UBLVersionID>
4   <cbc:CustomizationID>OIOUBL-2.02</cbc:CustomizationID>
5   <cbc:ProfileID schemeAgencyID="320" schemeID="urn:oioubl:id:profileid-1.2">
        Procurement-OrdAdv-BilSim-1.0</cbc:ProfileID>
6   <cbc:ID>A00095678</cbc:ID>
7   <cbc:CopyIndicator>false</cbc:CopyIndicator>
8   <cbc:UUID>9756b4d0-8815-1029-857a-e388fe63f399</cbc:UUID>
9   <cbc:IssueDate>2005-11-20</cbc:IssueDate>
10  <cbc:InvoiceTypeCode listAgencyID="320" listID="urn:oioubl:codelist:
        invoicetypecode-1.1">380</cbc:InvoiceTypeCode>
11  <cbc:DocumentCurrencyCode>DKK</cbc:DocumentCurrencyCode>
12  <cbc:AccountingCost>5250124502</cbc:AccountingCost>
13  <cac:OrderReference>
14    <cbc:ID>5002701</cbc:ID>
15    <cbc:UUID>9756b468-8815-1029-857a-e388fe63f399</cbc:UUID>
16    <cbc:IssueDate>2005-11-01</cbc:IssueDate>
17  </cac:OrderReference>
18  <cac:AccountingSupplierParty>
19    <cac:Party>
20      <cbc:EndpointID schemeID="DK:CVR">DK16356706</cbc:EndpointID>
21      <cac:PartyIdentification>
22        <cbc:ID schemeID="DK:CVR">DK16356706</cbc:ID>
```

```
23        </cac:PartyIdentification>
24        <cac:PartyName>
25          <cbc:Name>Tavleverandøren</cbc:Name>
26        </cac:PartyName>
27        <cac:PostalAddress>
28          <cbc:AddressFormatCode listAgencyID="320" listID="urn:oioubl:codelist:
              addressformatcode-1.1">StructuredDK</cbc:AddressFormatCode>
29          <cbc:StreetName>Leverandørvej</cbc:StreetName>
30          <cbc:BuildingNumber>11</cbc:BuildingNumber>
31          <cbc:CityName>Dyssegård</cbc:CityName>
32          <cbc:PostalZone>2870</cbc:PostalZone>
33          <cac:Country>
34            <cbc:IdentificationCode>DK</cbc:IdentificationCode>
35          </cac:Country>
36        </cac:PostalAddress>
37        <cac:PartyTaxScheme>
38          <cbc:CompanyID schemeID="DK:SE">DK16356706</cbc:CompanyID>
39          <cac:TaxScheme>
40            <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxschemeid-1.1">63</
                cbc:ID>
41            <cbc:Name>Moms</cbc:Name>
42          </cac:TaxScheme>
43        </cac:PartyTaxScheme>
44        <cac:PartyLegalEntity>
45          <cbc:RegistrationName>Tavleleverandøren</cbc:RegistrationName>
46          <cbc:CompanyID schemeID="DK:CVR">DK16356706</cbc:CompanyID>
47        </cac:PartyLegalEntity>
48        <cac:Contact>
49          <cbc:ID>23456</cbc:ID>
50          <cbc:Name>Hugo Jensen</cbc:Name>
51          <cbc:Telephone>15812337</cbc:Telephone>
52          <cbc:ElectronicMail>Hugo@tavl.dk</cbc:ElectronicMail>
53        </cac:Contact>
54      </cac:Party>
55    </cac:AccountingSupplierParty>
56    <cac:AccountingCustomerParty>
57      <cac:Party>
58        <cbc:EndpointID schemeAgencyID="9" schemeID="GLN">5798009811578</cbc:
              EndpointID>
59        <cac:PartyIdentification>
60          <cbc:ID schemeAgencyID="9" schemeID="GLN">5798009811578</cbc:ID>
61        </cac:PartyIdentification>
62        <cac:PartyName>
63          <cbc:Name>Den Lille Skole</cbc:Name>
64        </cac:PartyName>
65        <cac:PostalAddress>
66          <cbc:AddressFormatCode listAgencyID="320" listID="urn:oioubl:codelist:
              addressformatcode-1.1">StructuredDK</cbc:AddressFormatCode>
67          <cbc:StreetName>Fredericiavej</cbc:StreetName>
68          <cbc:BuildingNumber>10</cbc:BuildingNumber>
69          <cbc:CityName>Helsingør</cbc:CityName>
```

```
70          <cbc:PostalZone>3000</cbc:PostalZone>
71          <cac:Country>
72            <cbc:IdentificationCode>DK</cbc:IdentificationCode>
73          </cac:Country>
74        </cac:PostalAddress>
75        <cac:Contact>
76          <cbc:ID>7778</cbc:ID>
77          <cbc:Name>Hans Hansen</cbc:Name>
78          <cbc:Telephone>26532147</cbc:Telephone>
79          <cbc:ElectronicMail>Hans@dls.dk</cbc:ElectronicMail>
80        </cac:Contact>
81      </cac:Party>
82    </cac:AccountingCustomerParty>
83    <cac:Delivery>
84      <cbc:ActualDeliveryDate>2005-11-15</cbc:ActualDeliveryDate>
85    </cac:Delivery>
86    <cac:PaymentMeans>
87      <cbc:ID>1</cbc:ID>
88      <cbc:PaymentMeansCode>42</cbc:PaymentMeansCode>
89      <cbc:PaymentDueDate>2005-11-25</cbc:PaymentDueDate>
90      <cbc:PaymentChannelCode listAgencyID="320" listID="urn:oioubl:codelist:
          paymentchannelcode-1.1">DK:BANK</cbc:PaymentChannelCode>
91      <cac:PayeeFinancialAccount>
92        <cbc:ID>1234567890</cbc:ID>
93        <cbc:PaymentNote>A00095678</cbc:PaymentNote>
94        <cac:FinancialInstitutionBranch>
95          <cbc:ID>1234</cbc:ID>
96        </cac:FinancialInstitutionBranch>
97      </cac:PayeeFinancialAccount>
98    </cac:PaymentMeans>
99    <cac:PaymentTerms>
100     <cbc:ID>1</cbc:ID>
101     <cbc:PaymentMeansID>1</cbc:PaymentMeansID>
102     <cbc:Amount currencyID="DKK">6312.50</cbc:Amount>
103   </cac:PaymentTerms>
104   <cac:TaxTotal>
105     <cbc:TaxAmount currencyID="DKK">1262.50</cbc:TaxAmount>
106     <cac:TaxSubtotal>
107       <cbc:TaxableAmount currencyID="DKK">5050.00</cbc:TaxableAmount>
108       <cbc:TaxAmount currencyID ="DKK">1262.50</cbc:TaxAmount>
109       <cac:TaxCategory>
110         <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxcategoryid-1.1">
              StandardRated</cbc:ID>
111         <cbc:Percent>25</cbc:Percent>
112         <cac:TaxScheme>
113           <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxschemeid-1.1">63</
                cbc:ID>
114           <cbc:Name>Moms</cbc:Name>
115         </cac:TaxScheme>
116       </cac:TaxCategory>
117     </cac:TaxSubtotal>
```

```
118     </cac:TaxTotal>
119     <cac:LegalMonetaryTotal>
120       <cbc:LineExtensionAmount currencyID ="DKK">5050.00</cbc:LineExtensionAmount>
121       <cbc:TaxExclusiveAmount currencyID ="DKK">1262.50</cbc:TaxExclusiveAmount>
122       <cbc:TaxInclusiveAmount currencyID ="DKK">6312.50</cbc:TaxInclusiveAmount>
123       <cbc:PayableAmount currencyID ="DKK">6312.50</cbc:PayableAmount>
124     </cac:LegalMonetaryTotal>
125     <cac:InvoiceLine>
126       <cbc:ID>1</cbc:ID>
127       <cbc:InvoicedQuantity unitCode="EA">1.00</cbc:InvoicedQuantity>
128       <cbc:LineExtensionAmount currencyID="DKK">5000.00</cbc:LineExtensionAmount>
129       <cac:OrderLineReference>
130         <cbc:LineID>1</cbc:LineID>
131       </cac:OrderLineReference>
132       <cac:TaxTotal>
133         <cbc:TaxAmount currencyID="DKK">1250.00</cbc:TaxAmount>
134         <cac:TaxSubtotal>
135           <cbc:TaxableAmount currencyID="DKK">5000.00</cbc:TaxableAmount>
136           <cbc:TaxAmount currencyID="DKK">1250.00</cbc:TaxAmount>
137           <cac:TaxCategory>
138             <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxcategoryid-1.1">
                    StandardRated</cbc:ID>
139             <cbc:Percent>25</cbc:Percent>
140             <cac:TaxScheme>
141               <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxschemeid-1.1">63
                      </cbc:ID>
142               <cbc:Name>Moms</cbc:Name>
143             </cac:TaxScheme>
144           </cac:TaxCategory>
145         </cac:TaxSubtotal>
146       </cac:TaxTotal>
147       <cac:Item>
148         <cbc:Description>Hejsetavle</cbc:Description>
149         <cbc:Name>Hejsetavle</cbc:Name>
150         <cac:SellersItemIdentification>
151           <cbc:ID schemeAgencyID="9" schemeID="GTIN">5712345780121</cbc:ID>
152         </cac:SellersItemIdentification>
153       </cac:Item>
154       <cac:Price>
155         <cbc:PriceAmount currencyID="DKK">5000.00</cbc:PriceAmount>
156         <cbc:BaseQuantity unitCode="EA">1</cbc:BaseQuantity>
157         <cbc:OrderableUnitFactorRate>1</cbc:OrderableUnitFactorRate>
158       </cac:Price>
159     </cac:InvoiceLine>
160     <cac:InvoiceLine>
161       <cbc:ID>2</cbc:ID>
162       <cbc:InvoicedQuantity unitCode="EA">2.00</cbc:InvoicedQuantity>
163       <cbc:LineExtensionAmount currencyID="DKK">50.00</cbc:LineExtensionAmount>
164       <cac:OrderLineReference>
165         <cbc:LineID>2</cbc:LineID>
166       </cac:OrderLineReference>
```

```
167       <cac:TaxTotal>
168         <cbc:TaxAmount currencyID="DKK">12.50</cbc:TaxAmount>
169         <cac:TaxSubtotal>
170           <cbc:TaxableAmount currencyID="DKK">50.00</cbc:TaxableAmount>
171           <cbc:TaxAmount currencyID="DKK">12.50</cbc:TaxAmount>
172           <cac:TaxCategory>
173             <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxcategoryid-1.1">
                    StandardRated</cbc:ID>
174             <cbc:Percent>25</cbc:Percent>
175             <cac:TaxScheme>
176               <cbc:ID schemeAgencyID="320" schemeID="urn:oioubl:id:taxschemeid-1.1">63
                      </cbc:ID>
177               <cbc:Name>Moms</cbc:Name>
178             </cac:TaxScheme>
179           </cac:TaxCategory>
180         </cac:TaxSubtotal>
181       </cac:TaxTotal>
182       <cac:Item>
183         <cbc:Description>Beslag</cbc:Description>
184         <cbc:Name>Beslag</cbc:Name>
185         <cac:SellersItemIdentification>
186           <cbc:ID schemeAgencyID="9" schemeID="GTIN">5712345780111</cbc:ID>
187         </cac:SellersItemIdentification>
188       </cac:Item>
189       <cac:Price>
190         <cbc:PriceAmount currencyID="DKK">25.00</cbc:PriceAmount>
191         <cbc:BaseQuantity unitCode="EA">1</cbc:BaseQuantity>
192         <cbc:OrderableUnitFactorRate>1</cbc:OrderableUnitFactorRate>
193       </cac:Price>
194     </cac:InvoiceLine>
195 </Invoice>
```