Technische Universität München
Department of Informatics

Bachelor's Thesis in Informatics

A Workbench to Analyze X.509 in Applications

Adrian Reuter

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A Workbench to Analyze X.509 in Applications

Automatisierte Analyse der Verarbeitung von X.509 Zertifikaten in Applikationen

Author	Adrian Reuter
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Matthias Wachs
Date	March 15, 2017

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, March 15, 2017

Signature

Abstract

Global networking and secure digital communication have become basic needs for society, economy and individuals of our time. Security in digital communication significantly depends on digital certificates, which allow to prove authenticity and establish trust between communication partners. These certificates can only provide any security properties, if relying applications correctly validate them. The correct validation of certificates poses a complex task to overcome by implementing applications.

For this purpose, this thesis analyzes certificate validation of X.509, the most prominent certificate standard, in applications. Outcome of this thesis is an automated workbench, that allows for systematic and reproducible testing of certificate validation in applications. Therefore, the standardized validation process is analyzed and its complexity is elaborated. Based on this analysis, test cases are derived which describe the correct validation behaviour that must be provided by applications. This thesis develops a concept for systematically testing certificate validation in applications and designs a flexible and extensible system approach for the automated workbench. This workbench and the identified test cases are implemented and evaluated.

The certificate validation behaviour of a selected set of applications and libraries is analyzed by placing them into the automated workbench. The evaluation of the test results show that the applications in question predominantly conducted correct certificate validation: However, one of the applications examined accepts certificates that were issued for a different usage, while another library in default configuration accepts certificates that were issued for a different certificate subject. While the first weakness breaks the concept of certificate usage restrictions, the second weakness potentially misleads developers into severe Man-in-the-Middle vulnerabilities, which endanger all protection goals.

Zusammenfassung

Globale Vernetzung und sichere digitale Kommunikation haben sich zu gesellschaftlich, wirtschaftlich und auch persönlich zentralen Bedürfnissen unserer Zeit entwickelt. Die Sicherheit in der digitalen Kommunikation hängt dabei in erheblichem Maße von digitalen Zertifikaten hab, welche Authentizität und Vertrauenswürdigkeit gewährleisten. Diese Zertifikate besitzen jedoch nur dann Aussagekraft, wenn sie von Anwendungen umfassend validiert werden. Die korrekte Validierung von Zertifikaten stellt dabei einen komplexen und fehleranfälligen Prozess dar.

Diese Arbeit untersucht deshalb die Zertifikatsvalidierung des verbreiteten Zertifikatsstandards X.509 in Applikationen. Ziel der Arbeit ist es, eine Testumgebung zu entwerfen und zu implementieren, die das automatisierte, systematische und reproduzierbare Testen der Zertifikatsvalidierung in Anwendungen ermöglicht. Hierfür wird der standardisierte Validierungsprozess analysiert und dessen Komplexität herausgearbeitet. Basierend auf dieser Analyse werden Testfälle abgeleitet, die das Validierungsverhalten in Anwendungen überprüfbar machen. Daran angeschlossen entwirft diese Arbeit ein Konzept zum systematischen Testen von Anwendungen und entwickelt ein flexibles und erweiterbares Systemdesign für die Umsetzung einer automatisierten Testumgebung. Die Testumgebung und die identifizierten Testfälle werden anschließend implementiert und evaluiert.

Die Analyse einer exemplarischen Auswahl an Kommandozeilenprogrammen und Bibliotheken unter Verwendungen der entwickelten Testumgebung zeigt, das Zertifikatsvalidierung mehrheitlich korrekt durchgeführt wird. Eine der getesteten Anwendungen offenbart jedoch Schwächen bei der Prüfung des Verwendungszweck der ihr vorgelegten Zertifikate, während eine andere Bibliothek in ihrer Standardkonfiguration Zertifikate akzeptiert, welche zwar korrekt sind, aber nicht das eigentlich zu authentifizierende Subjekt ausweisen. Während die fehlerhafte Prüfung des Verwendungszwecks das Prinzip von Verwendungszweckbeschränkungen verletzt, kann die zweite Validierungsschwäche Entwickler irreführen und deren Produkte für schwerwiegende Man-in-the-Middle Attacken angreifbar machen, welche sämtliche Schutzziele gefährden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

With the global increase of digital communication and world-wide interconnection of networking infrastructures over the past decades, the need for security in digital data exchanges increased likewise. Today, secure digital communication is a foundation for progress and economic growth. Industry, research and economy depend on security in data exchange, as it prevents loss of know-how and informational advance, but equally simplifies accountable and resilient collaboration around the globe. Besides industry, security in digital communication becomes a fundamental necessity for individuals and involves more and more aspects of our daily lives. Security in digital data exchange helps individuals to preserve confidentiality, privacy and anonymity in their daily communication. Cryptography is the key methodology to secure digital communication. Modern cryptosystems often rely on digital certificates, which mean to prove authenticity and establish trust between endpoints. Consequently, certificates constitude a key technology for achieving many other protection goals, such as confidentiality, integrity, accountability or non-repudiation. X.509 is the most common standard for digital certificates and finds wide adoption in various infrastructures, most prominently for securing internet traffic flows on transport layer, but also for example in secure mailing.
However, certification infrastructures only make sense if all participants comprehensively verify certification documents. Hence, X.509 certificates can only provide any security properties, if they are validated correctly by X.509 implementations. A large amount of libraries and middlewares have been developed to face the challenging task of validating not only single X.509 certificates but entire certification paths. Such libraries constitute complex software projects with many contributors and are adapted to multiple platforms and operating systems. The amount of vulnerability exposures [6] [7], the *GnuTLS Goto Fail* [8] as well as scientific research [9] [10] shows that libraries and middlewares partially deploy broken or incomprehensive certificate validation. In

addition, some libraries overload developers who deploy these libraries with obscure configuration parameters [9] [10]. Moreover, as faulty middlewares are potential weak points for a lot of relying applications, they constitute an attractive target for attackers to put alot of effort and concentration in finding weaknesses and exploiting them.
The aim of this bachelor's thesis is to improve security in network communication by analyzing whether applications enforce correct certificate validation. The objective of this thesis is to design, engineer and implement a testing environment - henceforth called *workbench* - that provides functionality to systematically analyze X.509 certificate validation in applications. As such, the resulting workbench shall assist in investigating the security properies of applications.

1.2 Goals of This Thesis

This thesis develops and implements a workbench to analyze certificate validation in applications. Prior to the realization of the workbench, this thesis analyzes the formal X.509 certificate validation process that has been standardized by the Internet Engineering Task Force (IETF). This analysis identifies:

- which validation steps need to be executed by applications in order to correctly validate X.509 certificates,
- why certificate validation is a complex and thus error-prone process,
- why a workbench is necessary to test certificate validation in a systematical and scalable manner.

Subsequent to this analysis, a system approach for the workbench is designed that allows for automated, systematic and reproducible testing of certificate validation routines inside of applications. Therefore the design elaborates:

- how the generic testing sequence for testing an application looks like,
- which individual components are needed to construct the entire workbench and how do they interact,
- how instrumentation of applications can be realized, while preserving a flexible, extensible and scalable system design.

Thereafter, the workbench is implemented and utilized for exemplarily examining the certificate validation of a set of applications.
The outcome of this thesis is an insight into the certificate validation process as well as a software application to conduct automated tests on applications.

1.3 Document Structure

Chapter 2 of this bachelor's thesis familiarizes with background knowledge that is inherently needed for analyzing certificate validation in applications. This familiarization includes an overview of asymmetric cryptography, the X.509 certification infrastructure, and the *Transport Layer Security (TLS)* protocol, which is presumably the most prominent use case of X.509 certificates for authenticating communication peers.
Chapter 3 presents related work that has put effort into analyzing X.509 certificate validation in applications and developing automated testing environments for measuring certificate acceptance and rejection behaviour.
Chapter 4 conducts an analysis of the certificate validation process and elaborates the complexity of the validation process. As result of this elaboration, supposably error-prone parameters within the validation process are identified and certificate acceptance and rejection scenarios are identified. Following, corresponding test cases are extracted that cover these acceptance and rejection scenarios.
Chapter 5 presents the overall testing concept as well as the system approach for a workbench that provides functionality for automated testing of certificate validation in applications. Subsequently, this chapter conducts requirements engineering and identifies central workbench components and defines their interaction.
Chapter 6 gives an insight into implementation details on how X.509 certificates are generated, how the internal testing logic of the workbench is organized, and how test cases are inputted and test results are outputted by the workbench.
Chapter 7 evaluates the work done in this thesis by comparing the outcomes with the designated goals of this thesis and moreover places a selected set of applications into the workbench testing environment and investigates their certificate validation.
Chapter 8 discusses the impact of the weaknesses discovered in the evaluation results. It explains which vulnerabilities can result from the validation weaknesses and sketches how an attacker might exploit these weaknesses.
Chapter 9 finally summarizes the contribution of this thesis and emphasizes major findings, design decisions and evaluation results. It furthermore presents visions for future work that might extend the workbench's capabilities and integrate it into contexts of larger projects.

Chapter 2

Background

This chapter provides an insight into methods and protocols that constitude crucial basic knowledge for understanding the proceeding presented in this thesis. In particular, this chapter explains the main principles of asymmetric cryptography, which is a key technology for security in modern and scalable network infrastructures. This chapter further introduces the *X.509 Public Key Infrastructure (PKI)*, since it is a widely used infrastructure for distributing public keys for asymmetric cryptography. Furthermore *Transport Layer Security (TLS)* is introduced, as it is one of the most commonly used protocols for securing network traffic, making use of the X.509 PKI for authentication.

2.1 Asymmetric Cryptography

A fundamental concept for secure communication is asymmetric cryptography. Asymmetric cryptography is based on number theory problems that are impractical to solve with raw computational power, such as factorization or discrete logarithm of huge numbers. The computation of such functions is only feasible knowing a secret function parameter. This circumstance is taken advantage of in asymmetric cryptography, using functions that are feasible to calculate but impractical to invert. Such a function is used for encrypting messages and is feasibly computable for everyone. The needed function parameters are called *public key* and are made available for everyone. In constrast, for decrypting messages, the inverse function must be computed, which is only manageable using the secret function parameter, called *private key* [11, pp. 352 – 368]. Hence as shown in Figure 2.1, a plaintext message can be encrypted using the public key, and decrypted with the help of the correspondent private key. Consequently and in constrast to symmetric cryptography, asymmetric cryptography is not based on a common secret key that is used for both encryption and decryption and is secretly shared among the network entities that wish to communicate securely [11, p. 311]. However, asymmetric cryptography is often used to encrypt and securely communicate

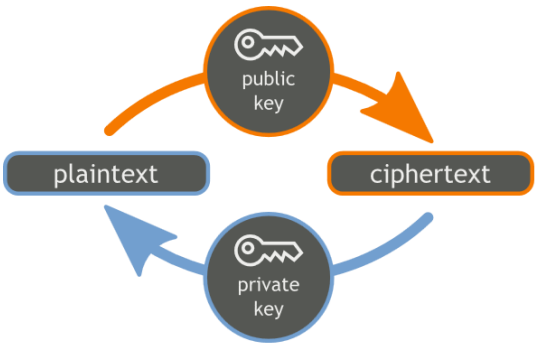

Figure 2.1: Relation between Private Key and Public Key (from [1])

a secret key for symmetric mechanisms, as symmetric algorithms typically benefit of a better performance compared to asymmetric ones and thus are preferably used for bulk data encryption [12].

2.1.1 Example of Asymmetric Cryptography

Putting asymmetric cryptography into practice, every communication peer maintains a key pair, consisting of a private key and a public key. The private key is kept secret and is never published. Contrary, the public key is not confidential at all and is published to every other peer that wishes to communicate with the peer owning it. The public key is often exchanged during connection establishment phase and is used by the other peers to encrypt messages destined for the owner of the correspondant private key.
To give an example, please consider the following situation as described in Figure 2.2: Peer Alice maintains the key pair $priv_{Alice}$ and pub_{Alice}, while peer Bob maintains the key pair $priv_{Bob}$ and pub_{Bob}. If Alice wants to send a confidential message to Bob, she is using Bob's public key pub_{Bob} to encrypt the message, and only Bob will be able to decrypt this message using his private key $priv_{Bob}$. Vice versa, Bob can send a confidential message to Alice by using pub_{Alice} for encryption, and only Alice will be able to decrypt that message with the help of her private key $priv_{Alice}$.

Besides encryption, some asymmetric cryptosystems can also be used for authentication, i.e. proving which peer originated a message. If Alice sends a message and signs it with her private key $priv_{Alice}$, any receiver can verify the signature with the help of Alice's public key pub_{Alice}. That means Alice can attach a signature to her messages that proves that only herself could have originated them [13].

As result of both encrypting and signing a message, an adversary peer Eve is neither capable of reading the confidential communication between Alice and Bob, nor is Eve capable of successfully altering or injecting messages without Alice and Bob noticing the manipulation attempt.

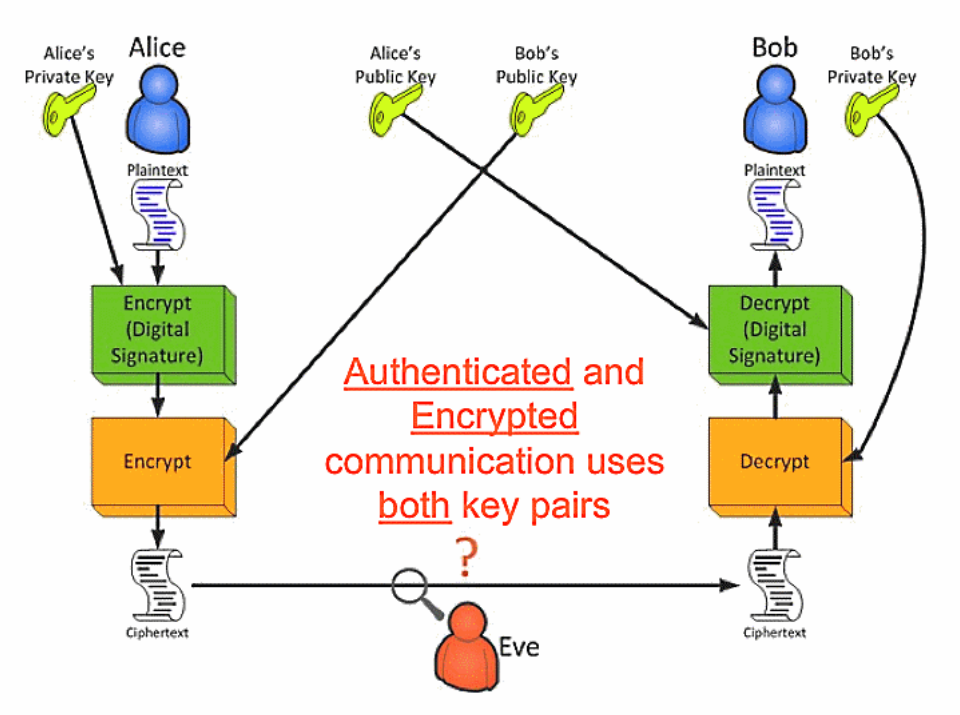

Figure 2.2: Alice and Bob Communicating Securely (from [2])

2.2 X.509 Public Key Infrastructure

The X.509 PKI is a widely used infrastructure to establish a trustworthy correlation between public keys and their corresponding network entities. That means the X.509 PKI aims to guarantee that a specific public key pub_X belongs to a specific network entity X, disabling other entities to impersonate as X and thereby break into a supposably confidential communication.

2.2.1 X.509 Certificates

Within the X.509 PKI, a network entity is distinctively identified by a *Distinguished Name (DN)*. The DN is a collection of information about a subject, such as the organization name, organizational unit, country, state, province, city and common name [14]. A X.509 certificate proves the ownership of a public key. Therefore a certificate includes the following obligatory information [14]:

- version of X.509 certificate format (mostly v3)
- certificate serial number
- subject public key (the public key to be certified)

- subject DN (the DN of the owner of the public key)
- issuer DN (the DN of the authority issuing the certificate)
- validity period
- signature algorithm information
- signature value
- [optional extensions]

Besides these basic information fields, a certificate may carry various additional extensions, such as subject alternative names, key usage restrictions, name constraints or policy constraints. A detailed explanation of all extensions can be found in RFC 5280 [14].
Individuals or organizations that wish to obtain a certificate for one of their public keys can apply to a commonly trusted third party for issuing the desired certificate. To do so, they build a *Certificate Signing Request (CSR)*, which mainly includes the respective public key as well as their Distinguished Name [15]. To actually prove the ownership of the public key, the CSR is signed with the correspondent private key [16]. The CSR is then sent to a *Registration Authority (RA)*, which is in charge to check whether the applicant is legitimate to ask for the desired certificate, and will validate the signature. If this process was successful, the Registration Authority instructs a subordinated *Certification Authority (CA)* to issue the requested certificate. The newly created certificate is signed with private key of the Certification Authority to make it unforgeable and trustworthy for other network entities [16].

2.2.2 Certificate Status and Revocation

There are situations where previously issued certificates have to be declared invalid, e.g. in case of private key disclosure, private key loss or changes in owner information. To face these situations where certificates need to be revoked before their regular validity period expires, two mechanisms gained wide adoption, namely *Certificate Revocation List (CRL)* and the *Online Certificate Status Protocol (OCSP)*. Both mechanisms are usually maintained by Certification Authorities and each authority is responsible for publishing certificate status information for the certificates it issues. Therefore certificates optionally contain extension fields pointing to the web location of the relevant CRL or the OCSP service [14]. CRLs are lists of revoked certificates and they are signed by the CA. Of course they are only up-to-date for a limited period of time and after their validity period expired, certificate validation routines are requested to retrieve an updated version. In case that a certificate's serial number appears on that list, this certificate has been revoked and should be consequently rejected [14]. The Online Certificate Status

Protocol serves the same purpose, but offers an almost real-time status query service instead of temporarily static and quickly outdated lists. Certificate validation routines can send *OCSP requests* to the appropriate OCSP responder that is specified in the certificate extension [17]. The OCSP responder will answer with a signed *OCSP response*, indicating the requested certificate status as either "good", "revoked" or "unknown" [17].

2.2.3 Trust Establishment and Certificate Chaining

If a network entity intends to prove its ownership of a certain public key to another entity, it delivers its X.509 certificate to that entity. With its signature, the issuing Certification Authority acts as warrantor, confirming that the public key presented in the certificate belongs to the owner also specified in the certificate [18]. Hence it is inherently required that the issuing CA is seen as a trustworthy third party for all network entities that shall trust this certificate. To achieve this, CAs themselves present their own certificates, either signed by another authority or selfsigned (signed with their own private key and thus involving no further authority). As result, a so-called *chain of trust* evolves, with a ultimately trusted certificate at its end, called *root certificate* [18]. These root certificates constitude a minimal set of certificates that are trusted by default and distributed via an out-of-band mechanism [19], such as being shipped with an application, an operating system or any other kind of X.509 implementation.

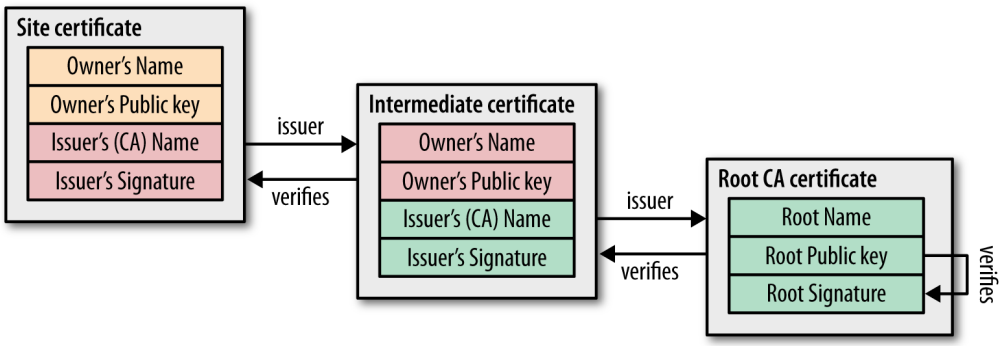

Figure 2.3: Chain of Trust (from [3])

Figure 2.3 demonstrates an exemplary chain of trust, consisting of a certificate for an arbitrary website ("site certificate"), a certificate of an intermediate certificate authority ("intermediate certificate") and a self-signed certificate of another certificate authority ("root CA certificate"): If a network entity tries to verify the authenticity of the site certificate, it needs to check its signature. This is done using the public key of the issuer, which is the intermediate CA. That public key is presented in the intermediate certificate. However, the authenticity of the intermediate certificate needs to verified as well. Its signature can be validated using the public key of the issuer, which is the root

CA. The appropriate public key is contained in the root CA certificate, which is trusted by default and thence requires no more verification. Besides the signatures along the chain of trust, each and every certificate needs to be validated with regard to domain name, validity period, key usage, revocation state according to CRLs or OCSP, as well as many other attributes and extensions. A precise analysis of all checks that need to be done within the certificate validation process will follow in Chapter 4.1.

2.3 Transport Layer Security

Transport Layer Security (TLS) is a protocol that operates on top of *Internet Protocol (IP)* and *Transmission Control Protocol (TCP)*, providing an encrypted, authenticated and integrity protected session for superior protocols. TLS has become an internationally widely adopted standard protocol for secure network communication, intensively used for securing HTTP, IMAP, SMTP, SIP, FTP and many other omnipresent protocols.

2.3.1 Protocol Description

TLS is derived from a protocol called *Secure Sockets Layer (SSL)*, a protocol originally developed by Netscape for securing HTTP-traffic between web servers and Netscape's webbrowser Netscape Navigator. Based on SSL, the *Internet Engineering Task Force (IETF)* has developed TLS as vendor-independent industrial standard protocol and already released version 1.2, currently working on version 1.3. Working on top of IP and TCP, which together offer reliable and stateful connections, TLS is located on the session layer (layer 5) of the ISO/OSI model. [11, pp. 796 – 798]

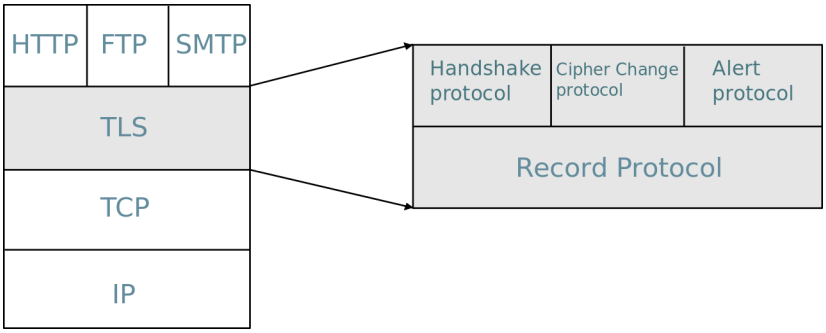

Figure 2.4: TLS Protocol Stack (from [4])

TLS distinguishes between connections and sessions. A TLS session defines security parameters that are used for all subordinate connections of that session, such as the cipher suite, the master secret and the certificate of the communication peer. That allows multiple TLS connections to be assigned to a session and thereby using the same

security parameters of that session, thus reducing negotiation traffic. A cipher suite primarily defines the usage of a certain hash algorithm and encryption algorithm, while the master secret is used to derive further key material. However, every connection maintains its own keys for encryption and integrity protection, derived from the master secret shared within the session. [11, pp. 805 – 806]

2.3.2 Subprotocol Types

As portrayed in Figure 2.4, the TLS protocol consists of the *Record Protocol* and three subprotocols, namely *Handshake Protocol*, *Change Cipher Spec Protocol* and *Alert Protocol*. Within the TLS protocol stack, the Record Protocol is one layer below the other three TLS protocols of equal height. It is responsible for fragmenting higher protocol data into data blocks, applying compression, calculating a *Message Authentication Code (MAC)* for each block and finally encrypting the concatination of data block and MAC [11, p. 799]. The Handshake Protocol is necessary for negotiating the cipher suite and cryptographic parameters along with authenticating the communication peers. The Alert Protocol is used to indicate warnings, such as on certificate rejection or a discrepancy of MACs [20]. Finally, the Change Cipher Spec Protocol is used to indicate that all future TLS records will be sent encrypted and integrity protected by the cipher suite that has been negotiated during the handshake [20].

2.3.3 Handshake Sequence and Connection Establishment

The handshake sequence always starts with a *Client Hello* message sent from the client to the server, mainly containing a list of client-side supported cipher suites, a 28 bytes random number and optionally the session ID of a previously established session. The server responds to the Client Hello with a *Server Hello* message, including either the proposed session ID or a new one, the chosen cipher suite and also a 28 bytes random number. In case the server sent the proposed session ID, it is thereby confirming the existance of that session and the handshake sequence can be abbreviated; if not, the server sends a new session ID and thus initiating a new session [11, p. 802]. The Server Hello is followed by a *Certificate* message, presenting the server certificate. It is optionally followed by a *Server Key Exchange* message in case a cipher suite was chosen that requires further security parameters. A *Certificate Request* message is likewisely optional, demanding the client to authenticate with a certificate as well. The server finishes its Server Hello with a *Server Hello Done* message [20].
After the Server Hello Done was received, the client optionally sends - if requested by a previous Certificate Request message - its certificate encapsulated in a *Certificate* message. In any case the client sends a *Client Key Exchange* message, either including a 48 byte random key called *premaster secret* encrypted with the server's public key, or including its Diffie-Hellman public value [11, p. 803]. If a client certificate has been

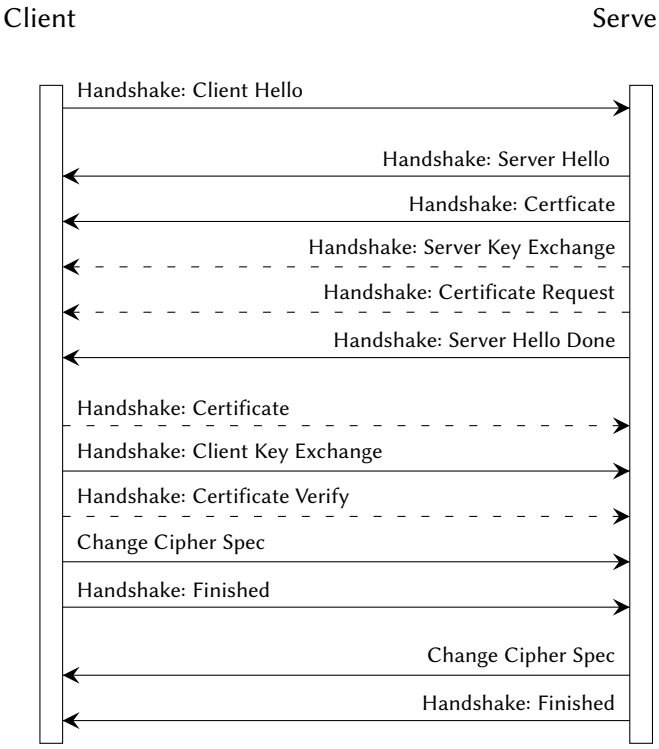

Figure 2.5: TLS Handshake Sequence

exchanged, the Client Key Exchange message is followed by a *Certificate Verify* message, proving the ownership of the respective public key by sending a signed hash of all handshake messages that have been exchanged so far. In any case the client sends a *Change Cipher Spec* message, indicating that all further messages will be encrypted and authenticated with the negotiated cipher suite. The keys for encryption and authentication are derived from the master secret, which is calculated separately on both client and server using the exchanged random values and the premaster secret [20].
The client concludes its part of the handshake with a *Finished* message, containing a hash of all handshake messages that have been exchanged so far, signed with the authentication key derived from the master secret. The server also sends a *Change Cipher Spec* message and finishes the handshake sequence with a *Finished* message by its side, likewise containing a signed hash of all exchanged handshake messages. This message is particularly important, as the server also proves the ownership of the public key contained in its certificate: the server is only capable of calculating the authentication key needed for the signed hash, if it was able to decrypt the premaster secret the client sent in its Client Key Exchange message [20].

Chapter 3

Related Work

A lot of security flaws arise when deploying X.509 libraries or X.509 middlewares in superior software components. The APIs of such libraries offer great flexibility to developers due to various configuration parameters and settings that can be supplied. However, this flexibility also moves greater responsibility to developers to correctly deploy the libraries and enforce extensive certification path validation. Some relevant validation steps such as hostname validation are even intentionally disabled during development phase in order to ease software testing [10]. If not re-enabled before rolling out the official software release, such configurations break the security concept of X.509. With the article *"SSL sicher implementieren"* [10], Dr. Yun Ding illustrates common mistakes when deploying X.509 libraries and brings well-known vulnerabilities to the readers attention. Ding gives advices to avoid well-known security breaches and sharpens the senses of developers to apply strict certificate validation. The proceeding *"The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software"* [9] by Georgiev et al. provides an analysis of broken certificate validation routines in several commonly used X.509 libraries and middlewares that lead to Man-in-the-Middle vulnerability in applications that deploy these software components. Georgiev et al. especially warns developers of weaknesses in online shopping middlewares. The author further highlights the fact that libraries often provide confusing and misunderstandable setting options, which frequently lead to unintentional insecure deployment of these libraries [9]. However, both publications remain eye-opening depictions of common mistakes and weaknesses of X.509 implementations and give appropriate advices to face them, but do not implement a workbench to actually test applications for faulty certificate validation routines.
In his master thesis proposal, titled *"Do Web Browsers Obey Best Practices When Validating Digital Certificates ?"* [21], Krati Kiyawat presents an evaluation of three different web browsers regarding their build-in certificate validation routines. Kiyawat discovers varying certificate acceptance and rejection behaviour across different web browsers as well as varying behaviour within different platform versions of the same browser. Further-

more Kiyawat points out that the desktop browser versions often apply more stringent certificate validation than their mobile versions for Android or iOS. Yet, Kiyawat's work concentrates on evaluating web browsers only and is not considering TLS-enhanced applications in general. Moreover Kiyawat's thesis identifies a rather small set of test cases, mainly consisting of expired, invalid, self-signed and revoked test certificates. This thesis though aims for more extensive and detailed testing, including test cases that cover entire certification paths as well as certificate extensions. Additionally and in contrast to Kiyawat's testbed, the workbench which is subject to this thesis aims to automate the testing process to some practicable extend, in order to facilitate a frequent testing of various applications.

With the paper *"SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps"* [22], Sounthiraraj et al. put great effort into a testbed which automatically detects whether developers deploy custom certificate validation routines in Android apps and thus potentially introduce Man-in-the-Middle (MitM) vulnerabilities. If SMV-Hunter detects a costum validation routine via static code analysis, an additional dynamic analysis at runtime executes a MitM attack to confirm the vulnerability. While SMV-Hunter shows a high degree of automation and constitutes an efficient framework for testing Android apps at large scale, it is focused on Android apps only and does not consider arbitrary TLS-enhanced applications. Furthermore the focus of SMV-Hunter is set to detecting certificate validation routines that spuriously accept self-signed certificates or simply accept all certificates, but does not concentrate on extensively testing validation routines whether they are checking all critical fields within certificates and certification paths.

This thesis is thematically related to Brubaker's et al. proceeding *"Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations"* [23], demonstrating an automated differential testing method for SSL/TLS libraries. The test certificates used by Brubaker et al. for testing TLS libraries are generated by fragmenting real certificates from the internet and randomly stitching fragments together; thus deriving new syntactically correct certificates that even cover unorthodox combinations of field values and extensions [23]. Differential testing in this context means that potentially faulty validation routines are detected by comparing the validation results of multiple SSL/TLS implementations. Whenever at least two implementations disagree about the validity of a certain test certificate, this potentially indicates a partly incorrect validation logic in one of the implementations [23]. Yuting Chen and Zhendong Su published a similar proceeding in 2015 titled *"Guided differential testing of certificate validation in SSL/TLS implementations"* [24], which presents a considerably more efficient differential testbed for SSL implementations. However, both projects are focused on a bounded list of wellknown SSL/TLS libraries such as OpenSSL [1] and GnuTLS [2], but are not designed to test arbitrary TLS-enhanced applications. Moreover, security flaws

[1] OpenSSL Cryptography and SSL/TLS Toolkit, `https://www.openssl.org`; last accessed on 2017/02/27

[2] GnuTLS Transport Layer Security Library, `https://www.gnutls.org`; last accessed on 2017/02/27

that arise from insecure deployment of TLS libraries or custom validation routines are consequently not focused by these two projects.

Chapter 4

Problem Analysis

X.509 certificates bear high responsibility for secure communication, as they are a mean to prove authenticity and establish trust between endpoints. Consequently, certificates constitude a key technology for achieving many other protection goals, such as confidentiality, integrity, accountability or non-repudiation. This chapter analyzes the motivation for the workbench that is subject to this thesis. The first section elaborates the certificate validation as specified in RFC 5280 [14]. The second section explains why certificate validation poses a complex task to be solved by X.509 implementations. Subsequently, the benefit of a workbench that is systematically testing certificate validation routines is illustrated by an investigation of possible use cases. Resulting from the analysis of the validation algorithm presented in RFC 5280 [14], certificate acceptance and rejection scenarios are extracted and test cases are derived that cover the corresponding scenarios.

4.1 Certificate Validation Process

In X.509, an entity that wishes to prove its identity is only mandated to present its end-entity certificate, and the process of constructing a complete certification path (including all intermediate certificates in correct order) and validating it remains the task of the relying peer. In contrast, TLS, which makes use of the X.509 PKI, obligates the entity that wishes to prove its identity to present the whole certification path to the relying party. RFC 5246 [20] specifies that TLS applications are entitled to expect a valid and complete certification path in form of a gapless certificate chain during the handshake; and are entitled to reject any certificate that comes without a valid chain that could be validated by the TLS implementation. As this thesis is focused on X.509 in TLS applications, it does not analyse the process of constructing certification paths, because the complete certification path is sent during handshake sequence.
RFC 5280 [14] specifies the certificate fields and values that are comprised in the X.509v3

Internet Profile, as well as an exemplary certification path validation algorithm. The validation logic of this algorithm defines the minimum functionality that needs to be provided by conforming implementations. The certificate validation process can be devided into two major parts: validating the basic certificate fields of each certificate within the chain, and validating the chain dependencies of a certification path. It is important to note that throughout this whole thesis we consider a certification path to be validated in direction from the trust anchor through all intermediate certificates down to the leaf certificate.The certification path itself that is to be validated, as well as trust anchor information, the current date and time, initial name constraints and initial policy constraints are considered to be the minimal inputs to any validation routine [14]. Based on these inputs, a conforming X.509 implementation validates the following:

Basic certificate processing
Each certificate within the certificate chain needs to be processed and their basic information fields must be validated. The *validity period* [14, Section 4.1.2.5] of a certificate must include the date and time at which the validation is executed, while the border timestamps 'NotBefore' [14, Section 4.1.2.5.1] and 'NotAfter' [14, Section 4.1.2.5.2] of the certificate are part of the valid time span. The certificate furthermore *must not be revoked* at the current time of validation, which needs to be checked by an out-of-band mechanism, such as a Certificate Revocation List or the Online Certificate Status Protocol. Finally, the cryptographic *signature value* [14, Section 4.1.1.3] at the end of the certificate must be verified to ensure the certificate is authentic and has not been forged.

Chain processing
Besides validating each certificate within the chain, all chain dependencies need to be validated as well. A certificate chain can be denoted as sequence *cert 1 .. n*, where *cert 1* is the top-most CA certificate that is immediately signed by the trust anchor, and *cert n* is the leaf certificate of the end-entity. Often but not necessarily trust anchors are encoded as self-signed certificates for practicability reasons and thus can be seen as *cert 0*. However, trust anchors are not considered to be part of the chain itself but rather an independent input parameter. For each certificate *i* within the chain (*cert 1 .. n*) it must be verified that the subject Distinguished Name (DN) of *cert i* equals the issuer DN of *cert i+1*. Subject DNs as well as issuer DNs are part of the basic certificate fields [14, Section 4.1]. Additionally, the subject DN of the first certificate in the chain *cert 1* must equal the DN of the trust anchor. This comparison of Distinguished Names ensures that the certification path is a *gapless chain of trust* that ends with the end-entity certificate in question.
For each version 3 certificate within the chain except the leaf certificate at the very end, a X.509 implementation must verify the basic constraints extension [14, Section 4.2.1.9]. This includes the verification that the *'cA' bit* [14, Section 4.2.1.9] of the basic constraints extension is set to true, in order to ensure that the certificate is a CA

certificate. Moreover, the *maximum path length* must not be exceeded. The maximum path length limits the number of non-self-issued intermediate CA certificates that are allowed to follow a CA certificate. The maximum path length is determined by the 'pathLenConstraint' [14, Section 4.2.1.9], which can be optionally set in the basic constraints extension. Furthermore, for all non-self-issued intermediate certificates of the chain it must be verified that the certificate's 'Subject' name [14, Section 4.1.2.6], which is part of the basic information fields, as well as all Subject Alternative Names (SAN) included in the 'Subject Alternative Name' extension [14, Section 4.2.1.6] - if present - fulfill the *name constraints* imposed by the 'Name Constraints' extension [14, Section 4.2.1.10] of preceding certificates. Therefor the subject names must firstly be within the permitted namespace and secondly must not be part of the explicitly inhibited namespace. Additionally to name constraint awareness, all intermediate certificates must fulfill the *policy constraints* that can be imposed and sequentially narrowed by 'Policy Constraints' extensions [14, Section 4.2.1.11] of preceding certificates within the chain.
If the *'Key Usage'* extension [14, Section 4.2.1.3] is present, each intermediate certficate in the chain must have the 'keyCertSign' bit [14, Section 4.2.1.3] set to true, which indicates that the public key that is being certified by the certificate is authorized to be used for signing further certificates.
Finally, *any other extensions* that were marked as 'critical' must be processed and verified. If an implementation encounters an extension that is marked as critical but is not able to process it, the certificate and thereby the whole certificate chain must be rejected. If any of the previously explained verifications fails, the validation process is aborted and the certificate chain must be rejected [14].

4.2 Complex Certification Paths

Correctly validating not only single X.509 certificates but entire certification paths from the trust anchor through all intermediate certificates down to the leaf certificate is a challenging task. Certification topologies are likely to get much more complex, multi-branched and meshed than simple chain structures. Complicated certification structures typically evolve when Certification Authorities of different Public Key Infrastructures cross-sign their respective CA-certificates in order to establish a larger common trust domain [25]. Figure 4.1 demonstrates a typical bilateral cross-signing situation: CA 1, which is responsible for PKI 1, cross-signes the root certificate of another CA 2, which is responsible for PKI 2 and vice versa. This procedure allows users of PKI 1 to trust certificates issued by CA 2 of PKI 2, yet only having installed the root certificate of CA 1 in their own root certificate store. For example, the signature of user certificate "cert. 2.2" of PKI 2 can be verified by users of PKI 1 with the help of certificate "cert. 2.1", whose signature in turn can be verified by certificate "cert.1". The latter is trusted by default as it is installed in the root store.

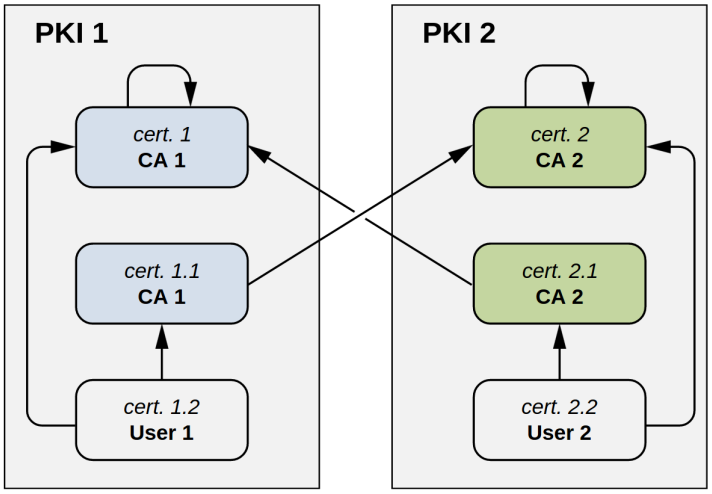

Figure 4.1: Example of Cross-Certification (from [5])

The complexity of certification paths is also increased by key renewals. The security of cryptographic keys can be enhanced by regularly renewing it and thereby preventing attackers from computing active keys in reasonable time. In case a Certification Authority wishes to change its own key pair and use a newer one, the CA issues a certificate for its new public key using the old key for signature and vice versa [25]. Without cross-signing the keys with the respective other key, the CA can not be trusted by its users during key transition phase, as the self-signed CA certificate for the new key first needs to be established in the trust store of all clients.
The X.509v3 PKI additionally introduces the concepts of name constraints and policy constraints as a mean to further restrict certificate validity [14]. Both constraint types can be imposed to a certificate by optional certificate extensions. Name constraints allow a CA to limit the competence of subordinate CAs to be only entitled to issue certificates for subjects within a narrowed namespace. Consequently, name constraints gradually restrict the namespace in which subject name as well as all Subject Alternative Names (SAN) must fall [14]. A certificate policy restricts the context in which a certificate should be used and in which context it is aiming to be trustworthy. For example, by using a specific policy identifier, a CA might issue a certificate for authentication in domains with lower security requirements (e.g. a end-entity certificates for blogging websites), which has not an adequate level of assurance for high security domains (e.g. authentication in aviation control systems) [26]. Both, name constraints as well as policy constraints, allow to narrow the competence and hence the validity of subsequent certifications gradually with each certification step. As the validation of these constraints requires implementations to maintain continuously changing internal data structures, these constraints introduce additional complexity and error-proneness to the certificate validation process.

In a nutshell, correctly validating certification paths is a challenging and error-prone

process. The algorithms and recommendations for constructing such certification paths are not analyzed in this thesis and are covered by RFC 4158 [27]. The workbench that is subject to this thesis is not supposed to construct certification paths itself, but rather takes certification paths as input and processes them by testing TLS-enhanced applications whether they properly or spuriously accept or reject these certification paths.

4.3 Automated Certificate Validation Testing

The previous depiction of the complexity of the certificate validation process motivates the need for a workbench that can test the certificate validation routines within applications. Moreover the complexity already indicates that testing validation routines comprises a multitude of test cases. Manually executing each test case for one application is already laborious and time consuming, but testing multiple applications or even testing multiple applications several times becomes unfeasible. Out of it results the need for automation of the testing process, which is exactly the objective of the workbench of this thesis. The workbench aims to automate and accelerate the testing process and thus introduces manageability and scalability for testing certificate validation routines. This section illustrates what a X.509 workbench which analyzes certificate validation in applications can be used for, and which benefits result from using such an automated and universal testing environment. Three major use cases are depicted in the following, but further use cases potentially exist and might show up in the context of other projects, or might be inspired when deploying the workbench.

4.3.1 Application Blackbox Testing

The workbench which is subject to this thesis is primarily designed for testing certificate validation routines within TLS applications. The workbench offers automation of the testing process from a black box perspective with regard to the application that is to be tested. Basically, the needed information about the application to be tested is how to start it and how to make it access a remote location while using TLS for securing the connection. In this use case the central question that can be answered by deploying the workbench is "Does an application validate certificates correctly ?", which is an essential quality criteria when examining the security of an application.

4.3.2 Regression Testing of Applications

The workbench also introduces comparability with regard to the security of applications. Using the workbench to test several different applications - while deploying the same

test suite as input parameter - creates a neutral and impartial testing environment, that generates reproducible test results and allows to compare applications with respect to their certificate validation routines. The test results can constitude a metric for ranking applications and libraries. Besides comparing different applications, the workbench can likewise serve to compare different versions / releases of the same application, and thereby be a tool for regression testing that verifies whether and how certificate acceptance behaviour changed over time or from one version to another. In a nutshell the workbench can be used to compare the certificate validation routines in:

- application A **vs.** application B
- version X **vs.** version Y
- application + version A.X **vs.** application + version B.Y

4.3.3 Certificate Validity Testing

Furthermore a X.509 workbench can be used for testing collections of real-world certificates rather than test certificates solely build for testing purposes. In this use case the testing principle of the workbench is inverted and an application that has previously proven to deploy correct certificate validation would be configured as client component. Then the workbench can be fed with collections of officially issued certificates and test whether they are valid or not. This can be a useful functionality, e.g. for large companies that have to maintain large amounts of certificates for their IT infrastructure, in order to check if certificates did expire, were revoked or are invalid for some other reason, and which certificates needs to be renewed to be valid and effective.

4.4 Identification of Test Cases

This section identifies test cases for a default test suite that is based on the preceding analysis of RFC 5280. The test cases cover most of the relevant tests that need to be executed on an arbitrary TLS-enhanced application to ensure that the validation logic of the application in question conforms to the minimum validation algorithm specified in RFC 5280 [14]. Certainly there are environments in which certificate validation routines need to be more restrictive and implement more stringent security policies to increase the level of trust. Consequently, further test cases would be needed to test such a custom certificate acceptance behaviour. However, security requirements vastly vary and this thesis focuses on the minimum requirements specified in RFC 5280 [14].
In the following, a certification path is assumed as sequence of certificates 0…n, where certificate 0 is the trust anchor and certificate n is the end-entity leaf certificate, with a varying number of intermediate CA certificates in between. In the following tables each row represents one test case. The column 'Information' specifies the attribute

within the certificates that is being modified, and 'Modification' shows the corresponding modification of the attribute value. The modification is added to a generically valid certification path and turns the path into either a valid or invalid path. 'Scope' indicates to which certificate within the certification path the modification is applied, and 'Expect' specifies whether the certification path is expected to be accepted or rejected.

Signature and Validity Period

The signature at the end of a certificate must fit the certificate content. If applications do not check the signature value, they are vulnerable to attackers who present forged certificates. The validity period must include the current time of validation, otherwise applications spuriously accept certificates that expired or a not valid yet. Table 4.1 lists corresponding test cases to test both signature and validity period values.

ID	Scope	Information	Modification	Expect
1	1,..,n	signature value	valid signature value	accept
2			invalid signature value	reject
3		validity period	notBefore ≤ NOW ≤ notAfter	accept
4			notBefore ≥ NOW	reject
5			notAfter ≤ NOW	reject

Table 4.1: Test Cases for Signature and Validity Period

Name Chaining and Trust Anchors

The term *name chaining* refers to the requirement that every certificate within the certification path must have been issued by the subject of the respective preceding certificate [25]. If this requirement is violated, the chain of trust is broken. Likewise, the first intermediate certificate within the certificate chain needs to be issued by the trust anchor. The trust anchor certificate can optionally be prepended to the certificate chain and thereby be denoted as *cert* 0. Table 4.2 lists the test cases to test name chaining in applications.

ID	Scope	Information	Modification	Expect
6	0,..,n-1	subject DN, issuer DN	subject DN of cert i = issuer DN of cert $i+1$	accept
7			subject DN of cert $i \neq$ issuer DN of cert $i+1$	reject
8	0		trust anchor recognized and prepend. to chain	accept
9			trust anchor recognized but not part of chain	accept
10			unrecogn. self-signed cert prepend. to chain	reject
11			no corresponding trust anchor for cert chain	reject

Table 4.2: Test Cases for Name Chaining and Trust Anchors

Version and Basic Constraints

For all version 3 intermediate certificates the basic constraints extension [14, Section 4.2.1.9] must be present and the 'cA' bit must be set to true. Moreover, for all version 1 and version 2 certificates it must be verified via a out-of-band process whether they really are CA certificates, or must be rejected otherwise. The path length constraint - if present - adjusts the maximum length of the certification path, which must not be exeeded. Table 4.3 lists the test cases to test version and basic constraints.

ID	Scope	Information	Modification	Expect
12	1,..,n-1	version, basic constraints ext.	v3, basic constraints present, cA-bit set	accept
13			v3, basic constraints NOT present	reject
14			v3, basic constraints present, cA-bit NOT set	reject
15			unrecognized v1 certificate	reject
16			unrecognized v2 certificate	reject
17	1,..,n-2		path length constraint NOT present	accept
18			path length constraint present in cert x, path length $> n - x$	accept
19			path length constraint present in cert x, path length $< n - x$	reject

Table 4.3: Test Cases for Version and Basic Constraints

Key Usage and Extended Key Usage

If present, the key usage extension must be validated by all conforming implementations, regardless of whether it is marked critical or not. For all CA certificates, the 'keyCertSign'-bit must be set to true. When a leaf certificate is used in combination with RSA key encipherment, the key usage must have the 'keyEncipherment'-bit [14, Section 4.2.1.3] set to true. When the extended key usage extension [14, Section 4.2.1.12] is marked as critical in an end-entity certificate, the 'serverAuth'-bit of this extension must be set to true for server certificates and the 'clientAuth'-bit of this extension must be set for client certificates. If both the key usage extension as well as the extended key usage extensions are present, they must agree in intended certificate usage. Table 4.4 lists test cases to test key usage extension and extended key usage extension.

Additional Validations

Moreover, any unrecognized critical extension must lead to certificate rejection, as well as the value 'anypolicy' must not occur in the policy mappings extension. Additionally to the verifications specified in RFC 5280, hostname validation must be carried out to prevent active Man-in-the-Middle vulnerability. Therefore, the target domain to which an application is about to connect to must be included in either the subject Common Name (CN) or any of the Subject Alternative Names of the leaf certificate. Table 4.5

lists test cases for various additional validation steps. Testing certificate policies, name constraints and certificate revocation states goes beyond the scope of this bachelor's thesis and will not be treated in the progression of this thesis. However it must be noted that the workbench that is being designed and developed within the bounds of this thesis is capable of testing these constraints, as corresponding test cases can be identified by future work and can then be easily inputted into the workbench as extended or alternative test suite.

ID	Scope	Information	Modification	Expect
20	1,..,n-1	key usage ext.	key usage ext. present (critical), keyCertSign-bit set	accept
21			key usage extension present (NOT critical), keyCertSign-bit NOT set	reject
22			key usage extension present (critical), keyCertSign-bit NOT set	reject
23	n		key usage extension present (critical), keyEncipherment-bit set	accept
24			key usage extension present (critical), keyEncipherment-bit NOT set	reject
25		extended key usage ext.	extended key usage ext. present (critical), serverAuth-bit set	accept
26			extended key usage ext. present (critical), serverAuth-bit NOT set	reject
27		key usage ext., extended key usage ext.	key usage = keyEncipherment, extended key usage (critical) = serverAuth	accept
28			key usage = keyEncipherment, extended key usage (critical) ≠ serverAuth	reject
29			key usage ≠ keyEncipherment, extended key usage (critical) = serverAuth	reject

Table 4.4: Test Cases for Key Usage and Extended Key Usage

ID	Scope	Information	Modification	Expect
30	n	subject DN, SAN extension	subject CN = target domain, target domain NOT included in SAN	accept
31			subject CN ≠ target domain, target domain included in SAN	accept
32			subject CN ≠ target domain, target domain NOT included in SAN	reject
33	1,..,n	unrecognized extension	unrecognized critical extension present	reject
34	1,..,n-1	policy mapping extension	'anypolicy' occurs in mapping extension	reject

Table 4.5: Test Cases for Additional Validation Steps

Chapter 5

Workbench Design

This chapter illustrates the design of the workbench. It presents a high-level system approach as well as the resulting functional and technical requirements for the workbench. This chapter also introduces the idea of a strict separation of the test suite from the actual workbench testing logic itself. Subsequently, the major components of the workbench and their interaction are explained. Furthermore, this chapter depicts how applications can be integrated into the workbench and how instrumentation of heterogenous TLS-enhanced softwares can be accomplished. Finally, the necessary input information sets as well as the generated output information set are specified.

5.1 System Approach

TLS distinguishes a server role and a client role. A connection is always initiated by the client and the client is in charge to validate the certificate presented by the server. Optionally, a server can obligate the client to authenticate with a client certificate as well and thereby enforce mutual authentication. Conceptually the workbench can be divided into three major components. The workbench comprises a server component, a client component and a central *control unit*. The control unit implements the automation and the testing logic, and has access to a pool of test certificates and their corresponding private keys. Chaining several test certificates forms a certification path that is either valid - and thus should be accepted by a validation routine - or invalid, and thus should be rejected by a validation routine. Such certification paths represent the concrete instantiation of a test case which is investigating the certificate acceptance and rejection behaviour of the software that is to be tested. The control unit takes such a chain of certificates and the private key for the leaf certificate of the chain from the pool and instruments the server software to use it for any future incoming TLS connections. Then, the control unit instruments the client component - which is the software to be tested with regard to its certificate validation routine - to initiate a TLS-connection to the

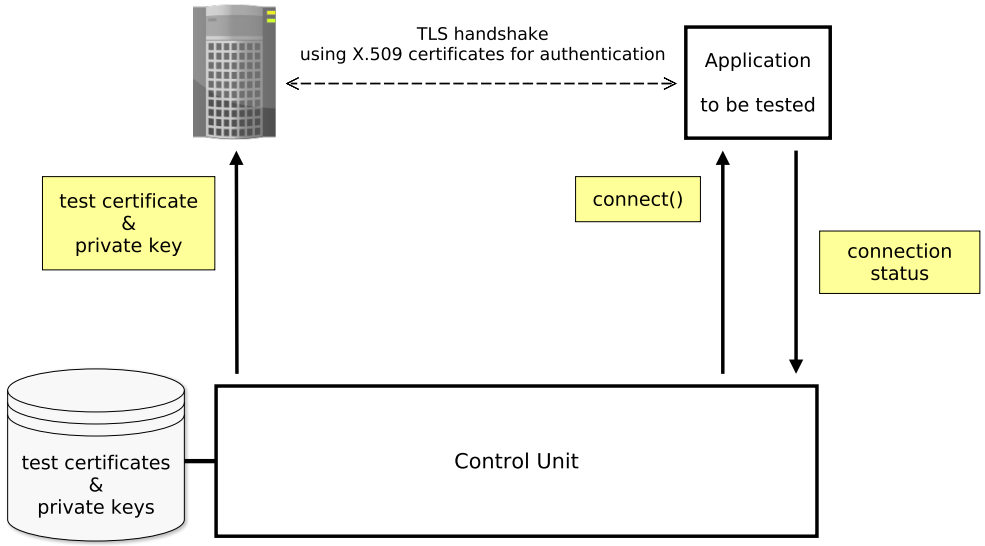

Figure 5.1: High-Level System Approach for the Workbench

server component and thereby makes the client validating the certificate chain presented by the server. After the TLS handshake has finished, the control unit investigates the output and the exit code of the client software:

- If the client decided to accept the certification path (which can be the correct or the wrong behaviour, depending on the test case) the connection was successfully established.

- If the client decided to reject the certification path (which can be also the correct or wrong behaviour, depending on the test case) the TLS connection establishment is aborted during handshake sequence and accessing the server component failed.

If the measured acceptance or rejection matches the expected behaviour for the certificate chain, the test case is considered as "passed", and otherwise as "failed". This process of assigning a new certificate chain (and the appropriate private key) to the server component, then instructing the client component to access the server, and finally measuring the connection result is repeated for every test case.
As result, the workbench is always testing the certificate validation routine inside the software that has the role of the TLS client. In case that the validation routine inside of a server software (e.g. Apache Webserver [1]) is the one to be tested, this server software must be placed into the workbench with the role of the TLS client that is accessing the workbench's regular testing server.

[1] Apache HTTP Server Project, `http://httpd.apache.org`; last accessed on 2017/02/19

5.2 Requirements Analysis

This section identifies functional as well as technical requirements that need to be taken into consideration for the design of the workbench. The functional requirements mainly derive from the use cases of the workbench, whereas the technical requirements primarily depend on the system approach of the workbench presented in Section 5.1.

5.2.1 Functional Requirements

In order to test applications, the previously identified test cases need to be translated into syntactically correct X.509 test certificates. Such test certificates carry manipulated attribute values that correspond to their test case definition. Multiple test cases need to be grouped to test suites. The workbench that is subject to this thesis must be able to read-in test suite specifications and must be able to distribute X.509 certificates to TLS server as well as TLS client applications. Moreover, the workbench must provide functionality to setup, execute, tear-down and clean-up the test environment for each test case of a test suite. To achieve this, the workbench must be able to instrument TLS server and TLS client applications, as well as to recognize the connection state of the application that is to be tested. The test results need to be stored in a structured and machine-readable file. The workbench must be adaptable to be universally applicable to new TLS-enhanced applications.

5.2.2 Technical Requirements

When generating X.509 test certificates, distinct attribute values and certificate extensions need to be specified and altered, even if this produces syntactically correct but semantically corrupt certificates. The workbench must be able to handle large amounts of certificate data. To allow flexible applicability to TLS applications, an adapter component for each application needs to be implemented which is using a standardized interface. The server component of the workbench must be able to configure the leaf certificate, the certificate chain and the private key for each test case. The client component needs to instrument the application that is to be tested, i.e. to install a custom trust anchor, connect to the server component and finally measure the output and the exit code of the application in question. In the context of application instrumentation, the workbench needs to provide functionality to read-in and alter configuration files.

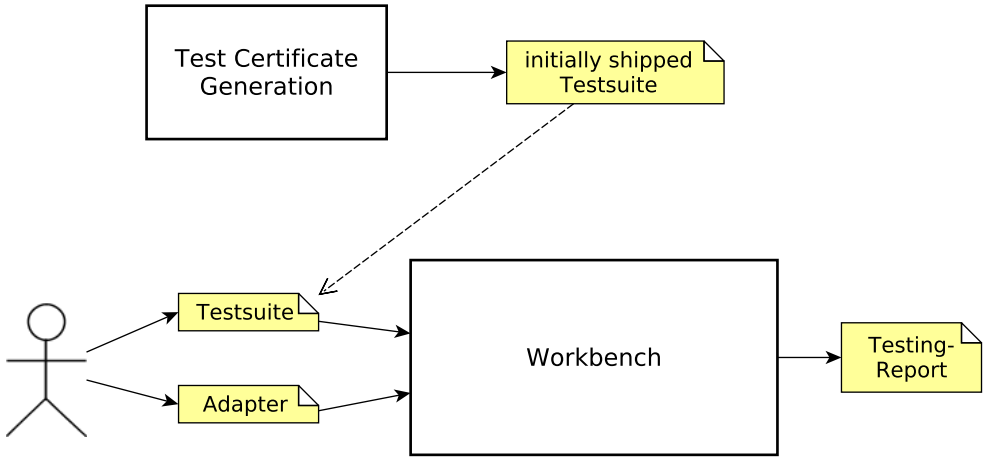

Figure 5.2: Workbench Interaction

5.3 Testing Concept

A major design choice for the workbench is the strict separation of the workbench testing logic and the actual test suites which comprise all the test cases that are executed on the application that is to be tested. A test suite specification can be created and configured independently and outside the bounds of the workbench itself. As such, the test suite can be reused for multiple test runs with various applications. It can comprise a custom composition of valid and invalid X.509 certificate chains to fit the individual security requirements that applications in question have to meet. This introduces great flexibility to define own test suites which are testing certificate validation routines for different levels of trustworthiness, e.g. one test suite that is only testing the minimal validation steps as specified in RFC 5280 [14] and one test suite that tests for more restrictive certificate validation. This concept also allows to use the workbench testing logic for future X.509 version specifications and validation constraints.
As shown in Figure 5.2 the workbench has two main input parameters: the test suite to be executed and an adapter implementation that is adapting the application in question to the automated workbench. After reading-in both parameters, the workbench processes the test suite and executes all test cases contained in it on the application that is instrumented by the given adapter. During the testing process, the workbench logs information about the progress and the success of the test run to the output stream of the workbench. Internal sequences of events and statuses can be optionally logged into log files, which can be inquired for debugging purposes or for obtaining detailed information in case a test case execution failed or an application instrumentation error occured. After the testing process has completed, the test results are saved to a struc-

tured and machine-readable file. The output file can also be converted to an HTML representation on-the-fly, which provides better readability and convenience for users. The latter functionality will be further discussed in Section 6.2.

5.4 Testing Sequence

As illustrated in Figure 5.3 the testing logic of the workbench consists of six different control units that organize the testing process. The *User-Interaction-Controller* is the unit that is initially invoked by the user and handles all interaction with the workbench user. It parses user inputs, configures the workbench to use the settings specified by the user and prints status information to the screen. The main control unit is the *Main-Controller*, which is instrumenting the *Testing-Controller* unit and is initiating a test run using a specific test suite and a specific application adapter. The Testing-Controller is in charge to prepare a test suite execution as well as executing it. Therefore it sequentially prepares, executes, tears down and cleans up each test case of the test suite, making use of the application instrumentation features provided by the *Server-Controller* and the *Application-Controller* units. The lifecycle that is traversed by each test case includes the following:

Setup Test Case:
Before a test case can be executed, all test case specific parameters need to be installed. That is the server component must be equipped with the new leaf certificate as well as the corresponding private key. Additionally, the new certificate chain with all intermediate CA certificates (and optionally including the trust anchor certificate) needs to be installed in the server component. Afterwards, many server applications must be reloaded or restarted in order for configuration changes to take effect. Finally, the new trust anchor certificate, that was used when creating the test certificates, must be implanted into the root certificate store of the application that is to be tested (i.e. the client component).

Execute Test Case:
The execution of a test case basically consists of two steps. At first, the client component is instructed to connect to the URL of the server component and thereby is forced to validate the certificate chain presented by the server component during the connection establishment phase. Meanwhile, the workbench must wait for the client component software to finish the connection establishment and must wait for the connection result. The resulting information - whether the server component was accessed successfully or not and due to which reason a connection was potentially aborted - is finally stored in corresponding fields within the workbench's internal representation of the test case.

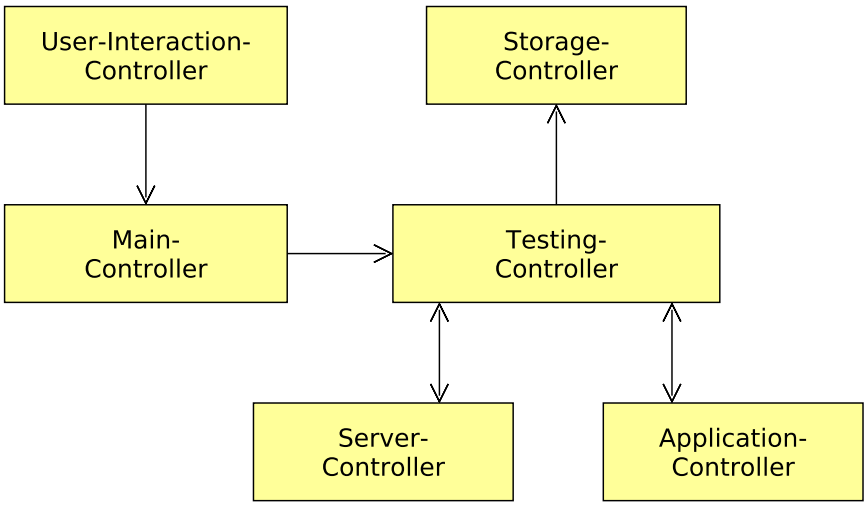

Figure 5.3: Workbench Controllers

Tear-down Test Case:
After the execution of the test case has finished, some client component applications need to be explicitly terminated if they did not already terminate after trying to access the server component.

Clean-up Test Case:
Finally, after the test case was executed and the connection was successfully torn down, the individual trust anchor certificate that was previously installed in the client component needs to be removed. Moreover, the server component configuration for the leaf certificate, the private key and the certificate chain needs to be neutralized in case the server component application is not operating in adhoc mode with non-persistent configurations.

The results for each test case, which have been previously stored for each test case in its individual in-memory representation, are brought together and saved to a common output file with the help of the *Storage-Controller*. A precise definition of required inputs and outputs will follow in Section 5.6, and their implementation will be discussed in Section 6.2.

5.5 Application Integration

The heterogeneity of TLS-enhanced server and client software demands adapter implementations. Every software offers different possibilities and options regarding start-up parameters, configuration files, output information, exit codes and termination. Taking all these differences into account makes an adapter implementation for each software unavoidable. However, the functionalities required by the workbench testing logic for instrumenting an application are conceptually the same for every application and can be phrased into interface definitions, whose methods need to be implemented by all custom adapter implementations. Hence the workbench includes the two interfaces, namely *Server-Controller* and *Application-Controller*. The Server-Controller interface must be used to derive custom server component implementations and primarily enforces methods for assigning new leaf certificates, private keys, certificate chains and for reloading / restarting the server component software. The Application-Controller interface must be used to derive custom client component implementations and primarily enforces methods for triggering the application that is to be tested to connect to the server component as well as terminating the application. The Application-Controller interface also enforces methods for implanting and removing trust anchor certificates from the root certificate store of the application in question.
A custom adapter implementation for instrumenting the server component is much less likely to be necessary than a client component adapter. Generally speaking, a custom server component adapter is only necessary if an application is to be tested that uses a TLS-secured protocol other than HTTPS; and thus a testing server software other than the default must be used. In constrast, a client component adapter is necessary for every new application that is to be tested. Conceptually a client component adapter can instrument either a traditional client software (e.g. GNU Wget [2]) or also a server software (e.g. Apache [3]). Custom adapter implementations can be given as program parameters at start-up of the workbench and are then dynamically loaded into the static workbench core implementation.

5.6 Structured Storage

This section specifies the structured sets of information that constitude a wellformed input test suite for the workbench and the resulting output information that the workbench is generating after the execution of a test suite. The following notation in extended Backus-Naur form (EBNF) describes a test suite in the context of this thesis:

[2] GNU Wget, `https://www.gnu.org/software/wget/`; last accessed on 2017/02/19
[3] Apache HTTP Server Project, `http://httpd.apache.org`; last accessed on 2017/02/19

⟨*Testsuite*⟩ ::= ⟨*TestsuiteName*⟩ ⟨*TestsuiteDescription*⟩ { ⟨*Testcase*⟩ }

⟨*TestsuiteName*⟩ ::= ⟨*TestsuiteDescription*⟩ ::= UTF8-String

⟨*Testcase*⟩ ::= ⟨*TestID*⟩ ⟨*TestDescription*⟩ ⟨*ExpectedBehaviour*⟩ ⟨*LeafCertificate*⟩ ⟨*PrivateKey*⟩ ⟨*TestCertificateChain*⟩ ⟨*TrustanchorCertificates*⟩

⟨*TestID*⟩ ::= UTF8-String

⟨*TestDescription*⟩ ::= UTF8-String

⟨*ExpectedBehaviour*⟩ ::= `accept` | `reject`

⟨*LeafCertificate*⟩ ::= ⟨*PrivateKey*⟩ ::= pathToPemFile

⟨*TestCertificateChain*⟩ ::= ⟨*TrustanchorCertificates*⟩ ::= pathToPemFile

The term *pathToPemFile* represents a path to a PEM encoded binary file including either a certificate, a private key or a sequence of certificates. The output generated by the workbench adheres to the following scheme, which reuses the preceding definitions of <*TestID*> and <*ExpectedBehaviour*>:

⟨*TestReport*⟩ ::= { ⟨*TestResult*⟩ }

⟨*TestResult*⟩ ::= ⟨*TestID*⟩ ⟨*TestStatus*⟩ ⟨*ExpectedBehaviour*⟩ ⟨*MeasuredBehaviour*⟩ [⟨*RejectionMessage*⟩]

⟨*TestStatus*⟩ ::= `success` | `fail` | `error`

⟨*MeasuredBehaviour*⟩ ::= ⟨*ExpectedBehaviour*⟩

⟨*RejectionMessage*⟩ ::= UTF8-String

Chapter 6

Implementation

This chapter provides an insight into implementation details. It explains how the test certificates that correspond to the test cases identified Section 4.4 are generated. Also the data format of input and output data is defined and which libraries are used to create and write data files. Furthermore, this chapter explains how the workbench components are implemented and how the testing sequence is implemented. Finally, this chapter describes how future adapter implementations and alternative test suites can be integrated into the workbench environment.

6.1 Certificate Generation with X.509 Certificate Builder

The test cases identified in Section 4.4 need to be translated into valid and invalid test certificates and certification paths that examine the certificate acceptance and rejection behaviour of an application in question. X.509 certificates can be comfortably created with the help of the *cryptography.io* [1] library for Python programming language. The huge advantage of this library compared other frameworks and libraries such as pure *OpenSSL* [2], *GnuTLS* [3] or *Bouncy Castle* [4] is that it provides a lightweight and easy-to-use 'CertificateBuilder' class that allows for adding, removing and altering almost any attribute field of an X.509 certificate data structure. Using the *cryptography.io* library, generating a certificate consists of the following steps:

1. generate an asymmetric key pair
2. create a new empty certificate structure and fill it with information sets; e.g. subject name, issuer name, public key, validity period and extensions

[1] Cryptography.io Python Library, `https://cryptography.io/en/latest/`; last accessed on 2017/02/27
[2] OpenSSL Cryptography and SSL/TLS Toolkit, `https://www.openssl.org`; last accessed on 2017/02/27
[3] GnuTLS Transport Layer Security Library, `https://www.gnutls.org`; last accessed on 2017/02/27
[4] Legion of the Bouncy Castle, `https://www.bouncycastle.org`; last accessed on 2017/02/27

3. sign the certificate structure

An exemplary Python code snippet that creates a certificate using the *cryptography.io* library might look like this [28]:

```
# Step 1
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)
# Step 2
cert = x509.CertificateBuilder()
cert = cert.subject_name(x509.Name([...]) )
cert = cert.issuer_name(x509.Name([...]) )
cert = cert.public_key(priv_key_root.public_key())
cert = cert.serial_number(x509.random_serial_number())
cert = cert.not_valid_before(datetime.datetime(2017,3,15))
cert = cert.not_valid_after(datetime.datetime(2022,3,15))
cert = cert.add_extension(
    x509.BasicConstraints(ca=True,path_length=None),
    critical=True
)
# Step 3
cert = cert.sign(private_key, hashes.SHA256(), default_backend())
```

As the certificates needed for the default test suite of our workbench are almost identical except for some test case specific modifications, we can avoid creating and filling all new and empty certificate structures for each certificate and instead generate a template certification path. Therefore a valid default certification path of length three is generated, including a root CA certificate, an intermediate CA certificate and an end-entity certificate. For each test case this default certification path is used, and the certificates within the path that need to be changed for this test case are copied into new objects. Afterwards, the information set that is specific for the test case can be adjusted in these new objects, reusing the original keys for signing the modified certificate structure. For each test case identified in Section 4.4, one or two representative indices x were picked from the *scope* and separate certification paths were generated carrying the respective modification in certificate x.

6.2 Data Management and Data Storage

Both the input and output information sets, as specified in Section 5.6, adhere to a strict structure and thereby suggest a structured data format. Thus the Extensible Markup Language (XML) is a well suited data format for storing test suite specifications as well as storing test results. A conforming XML document that specifies an input test suite for the workbench has the following structure:

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite>
    <name>Default test suite</name>
    <description>This test suite covers ...</description>
    <pathsRelativeToThisFile>true</pathsRelativeToThisFile>
    <testcases>
        <testcase>
            <id>1</id>
            <description>Invalid signature cert </description>
            <expectAcceptance>true</expectAcceptance>
            <leafCertificate>/001/leaf.crt</leafCertificate>
            <privateKey>/001/leaf.key</privateKey>
            <certificateChain>/001/chain.pem</certificateChain>
            <trustAnchors>/001/rootCA.pem</trustAnchors>
        </testcase>
        ...
    </testcases>
</testsuite>
```

The document primarily contains a test suite name, a test suite description, and a sequence of test case specifications. Such a test case specification includes an alphanumeric identifier, a description, a definition whether the test case is expected be accepted or not, and the path information where the respective *pem* files can be found. Optionally, the test suite specification can include a boolean value which indicates whether the path information within the test case specifications should be relative to the location of this test suite file. This toggle allows to easily copy and paste the whole test suite to an arbitrary directory without needing to change any paths inside the XML document.
An output file, which is created by the workbench when it finished a test run, has the following structure:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet href="reportstyle.xsl" type="text/xsl"?>
<testreport>
```

```
    <testsuiteName>Default test suite</testsuiteName>
    <clientComponent>cURL 7.47.0</clientComponent>
    <serverComponent>NanoHTTPD 2.3.1</serverComponent>
    <testStart>17-02-26 15:01:05</testStart>
    <testFinish>17-02-26 15:02:20</testFinish>
    <testDuration>00:01:15</testDuration>
    <testresults>
        <testresult>
            <id>001</id>
            <status>success</status>
            <expectedBehaviour>reject</expectedBehaviour>
            <measuredBehaviour>reject</measuredBehaviour>
            <rejectionMessage>Due to ...</rejectionMessage>
        </testresult>
        ...
    </testresults>
</testreport>
```

The document names the test suite, the client software which was tested, the software of the testing server used by the workbench, and timestamps which indicate the test execution start, finish and duration. Additionally, the output document references a XSL file. This file is used by the Extensible Stylesheet Language Transformation technology, which allows web browsers that support this technology to transform XML documents on-the-fly into styleable HTML code. The result is an easily readable and convenient representation of the XML output document for users that open the XML file with a recent web browser.
Inside the Python script for building the test suite, the LXML [5] library was used to automatically create the XML test suite specification file. The workbench main program uses the JDOM [6] library for reading in and parsing XML test suite specifications as well as writting the XML output files.

6.3 Implementation of Testing Logic

The workbench testing logic is implemented in Java programming language and primarily includes the four controller classes *UserInteractionController*, *MainController*, *TestingController* and *StorageController*; each serving their designated purpose as defined in Section 5.4. As shown in Figure 6.1, the testing logic moreover includes two interface classes, namely *ServerController* and *ApplicationController*, which define the

[5] LXML - XML and HTML with Python, `http://lxml.de`; last accessed on 2017/02/27
[6] JDOM - accessing XML data from Java code, `http://www.jdom.org`; last accessed on 2017/02/27

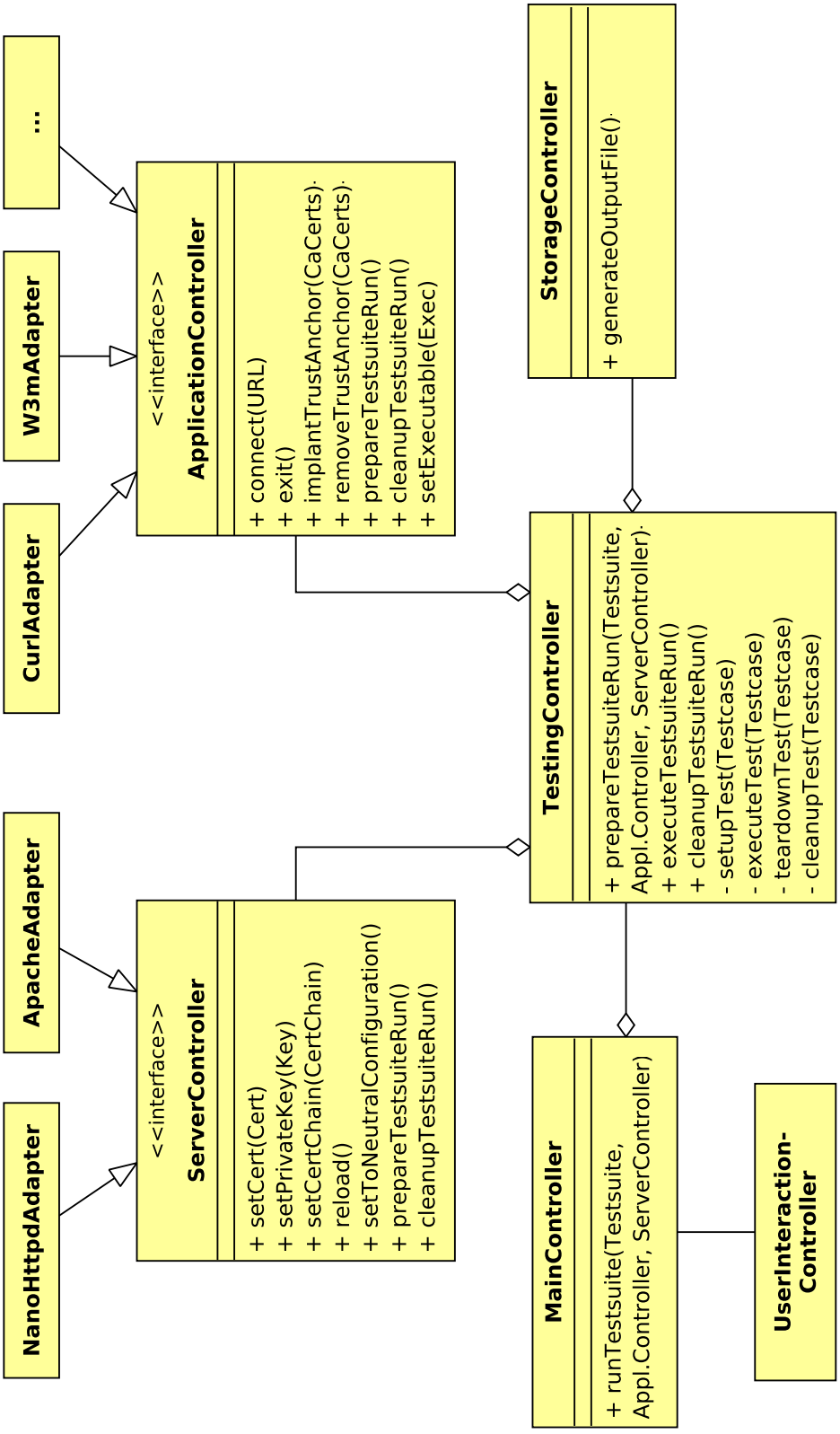

Figure 6.1: Workbench Core Classes

methods that adapter classes need to implement. Inside a derived adapter class, such as the *CurlAdapter* or the *W3mAdapter*, the *ProcessBuilder* class - which is part of the *java.lang* standard package - is used to create a new operating system process and execute the respective application binary. The ProcessBuilder class allows to hand over startup arguments as well as fetching the input-, output- and error-streams and the exit code of the newly created OS process. The latter functionality is used to determine whether an application was successfully started and whether a client component application is correctly or spuriously accepting the certificate chain presented by the workbench testing server. The class name of an adapter implementation to be used for the testing process is given as workbench startup argument, allowing to use

```
Class adapterClass = Class.forName("nameOfAdapterClass");
```

for dynamically determining and loading the corresponding adapter implementation at runtime. The testing sequence after reading in and configuring startup arguments by the *User-Interaction-Controller* conceptually consists of the following steps:

```
mainController.runTestsuite(".../testsuite.xml",
                        applicationAdapter, serverAdapter)
    testingController.prepareTestsuiteRun(".../testsuite.xml",
                        applicationAdapter, serverAdapter)
        testsuite = loadTestsuite(".../testsuite.xml")
        serverAdapter.prepareTestsuiteRun()
        applicationAdapter.prepareTestsuiteRun()
    testingController.executeTestsuiteRun()
        foreach(testcase in testsuite)
            testingController.setupTestcase()
            testingController.executeTestcase()
            testingController.teardownTestase()
            testingController.cleanupTestcase()
        endForeach
        storageController.generateOutputFile()
    testingController.cleanupTestsuiteRun()
        serverAdapter.cleanupTestsuiteRun()
        applicationController.cleanupTestsuiteRun()
```

In the pseudo code snippet above, the indention indicates that following method calls with higher indention are called from the context of the preceding method. While the snippet is not complete and does not correspond to the exact implementation syntax, it fairly represents the core of the testing sequence. The methods for setting up, executing, tearing down and cleaning up a test case implement the steps defined in Section 5.4

Chapter 7

Evaluation

This chapter evaluates the outcome of this thesis. The evaluation is splitted into two parts. The first section compares the achievements with the initial goals of this thesis. The analysis part, design part as well as the implementation part of the thesis are taken into consideration. The second section evaluates the certificate validation behaviour of several software projects by executing them in our workbench.

7.1 Comparison with Goals of This Thesis

Analysis of Certificate Validation Process
The vision of this thesis is to analyze X.509 certificate validation in applications. Therefore an analysis of the formal certificate validation process is required. This thesis accomplishes this requirement by analyzing the validation process specified in RFC 5280 [14], identifying essential validation steps that conforming validation routines need to go through. Based on this analysis, the thesis derives certificate acceptance and rejection scenarios, with each scenario leading to a set of test cases that investigate the validation behaviour of applications. Table 7.1 provides an overview of all test scenarios that were identified and successfully implemented to constitude a default test suite for testing certicate validation in applications.

Automated, Systematic and Reproducible Testing
Within the bounds of this thesis, a workbench was designed and implemented, that allows for analyzing certificate validation in applications in an automated, systematical and reproducible manner. Due to the variety of domains where X.509 certificates are deployed, it is simply unmanageable to design an integrated analysis tool that covers all heterogenous use cases of X.509 certificates in applications. Therefore this thesis early decides to focus on analyzing certificate validation in applications that use TLS as secure transport layer protocol. The benefit of this decision is that the resulting

Test Scenario	Identified and Implemented
Signature	✓
Validity period	✓
Name chaining	✓
Basic constraints	✓
Key uage	✓
Extended Key Usage	✓
Hostname Validation	✓
Revocation State	✗
Name Constraints	✗
Certificate Policies	✗

Table 7.1: Implemented Test Cases

workbench covers a multitude of use cases of X.509 certificates, as TLS constitudes a basic technology on transport layer for securing many superiour protocols and application data exchanges. The workbench that is designed and implemented within the bounds of this thesis is capable of sequentially preparing and executing large amounts of given test cases on TLS-enhanced applications whose validation routines are to be examined. The validation behaviour of the application in question is documented for each test case and manifested in a test report as output document. Test results show to be consistent throughout multiple executions under the same circumstances and thus compose reproducible outputs.

Extensible and Scalable System Design

The workbench is designed with flexibility and extensibility with regard to the applications that can be placed into it, on server side as well as on client side. The applicability of the workbench to arbitrary TLS-enhanced applications is achieved by small adapter implementations, that need to be implemented for each application. The adapter to be used can be specified as workbench startup parameter and is dynamically loaded at runtime. Moreover, the workbench also shows universality and extensibility with regard to the tests that are executed on applications in question. This quality is achieved by a strict separation of the workbench testing logic and the test suites that include the test cases to be executed. The default test suite specification or a custom test suite specification can be given as input parameter at workbench startup in terms of a XML document. The test result output file generated by the workbench is also a XML document, which implies that output data is stored in a highly structured manner and hence offers the possibility to be read in and used in future processing. At the same time the XML nature of the output file allows a convenient on-the-fly transformation into a user-friendly HTML representation, using XSLT technology.

Opportunities for Improvement
The default test suite identified within the bounds of this thesis does not include test cases that cover certificate policies, name constraints and certificate revocation states. Also, using the *cryptography.io* [1] Python library, no X.509 certificates of version 2 could be crafted, as this functionality was not implemented in this library. Likewise, the library does not support crafting unrecognized critical marked extensions, so test cases 16 and 33 were omitted during implementation of our default test suite. However, test cases can be identified and implemented by future work and can be easily inputted into the workbench as alternative test suite, possibly but not necessarily using the *cryptography.io* library for certificate generation.

7.2 Certificate Validation in Applications

The following sections analyze the certificate validation inside of several different TLS-enhanced command-line applications and libraries, by implementing corresponding adapter classes and thus placing the applications inside the workbench testing environment. For retrieving the results of this evaluation, the workbench is operated on a recent Lenovo Thinkpad Yoga 260 notebook, manufactured in beginning of 2016 and running Ubuntu 16.04 LTS. The hardware of this notebook includes an Intel Skylake i7-6500U CPU and 8 GB DDR4-2133 memory. The test suite identified in Section 4.4 is used as common input for all test runs, and comprises exactly 30 test cases. Moreover, NanoHTTPD 2.3.1 [2] is used as workbench server component software, which is a pure and lightweight Java web server framework.The test execution duration for executing the whole test suite on an application was approximately between 8 and 13 seconds. The Java process of the workbench core program consistently consumed 2.5% or less of the system's DDR memory and 1% or less of the system's CPU power.

7.2.1 Exemplary Blackbox-Testing of Command-Line Applications

This section evaluates the certificate validation inside of three different applications, namely W3M [3], cURL [4] and Wget [5]. Therefore the respective application adapter classes *W3mAdapter*, *CurlAdapter* and *WgetAdapter* were implemented, which allow the workbench to instrument the respective application and analyze its certificate acceptance and rejection behaviour in a reproducible and automated manner.

[1] Cryptography.io Python Library, `https://cryptography.io/en/latest/`; last accessed on 2017/02/27
[2] NanoHTTPD web server, `https://github.com/NanoHttpd/nanohttpd`; last accessed on 2017/03/13
[3] w3m - Pager and Text-based Browser, `http://w3m.sourceforge.net`; last accessed on 2017/03/03
[4] cURL - Library for Transferring Data with URLs, `https://curl.haxx.se`; last accessed on 2017/03/03
[5] GNU Wget, `https://www.gnu.org/software/wget/`; last accessed on 2017/02/19

Application	Version	TLS Backend	Backend Version
W3M	0.5.3	OpenSSL	1.0.2g
cURL	7.47.0	GnuTLS	3.4.10
cURL	7.47.0	OpenSSL	1.0.2g
Wget	1.17.1	OpenSSL	1.0.2g

Table 7.2: Tested Command-Line Applications

W3M is a small but powerful web browser application that operates on command-line only and thus requires no Graphical User Interface (GUI) environment. It relies on OpenSSL as backend TLS library. When analyzing W3M version 0.5.3 with our workbench (relying on OpenSSL 1.0.2g), the application showed excellent certificate validation behaviour. As shown in Table 7.3, W3M validated all certificates correctly and hence passed all test cases. Furthermore, when certificates were correctly rejected as expected by the test case, the rejection message thrown by W3M always indicated the correct reason for rejection. The mean test execution duration of three test runs was approximately 8 seconds.

cURL is a free command-line tool for transfering data with various protocols, such as HTTP, FTP, LDAP and many others, supporting both up- and downloading of data. When compiled and installed manually, cURL can be configured to rely on a TLS library of choice, supporting OpenSSL, GnuTLS and many others. In our evaluation, cURL 7.47.0 was once compiled to use GnuTLS version 3.4.10, and another time to use OpenSSL 1.0.2g. As shown in Table 7.3, when compiled with GnuTLS, cURL consistently failed in test cases 24 and 29, thus accepting certificates with wrong key usage specification. Test case 24 verifies that the key usage extension in leaf certificates allows key encipherment, which is necessary for our test certificates. Test case 28 and test case 29 both verify that the key usage extension and the extended key usage extension agree in a common key usage scenario (as mandated by RFC 5280 [14] in Section 4.2.1.12); one test case correctly setting the key usage but misconfiguring the extended key usage and the other test case vice versa. Surprisingly, test case 29 failed whereas test case 28 passed. Apparently cURL with GnuTLS accepts certificates regardless of the mandatory key usage consense, as long as the extended key usage is indicating the correct usage. In contrast, when compiled with OpenSSL, cURL passes all test cases. The mean test execution duration of six test runs (3 × with GnuTLS, 3× with OpenSSL) was approximately 13 seconds. Certificates that are rejected by cURL are rejected for the correct reason.

GNU Wget is a free and handy command-line tool for accessing and downloading data from remote sites. It easily allows integration in script programming and has found wide adoption. Wget can be compiled with different external TLS libraries. Our evaluation used Wget version 1.17.1 compiled with OpenSSL version 1.0.2g. As shown in Table 7.3, the application proved excellent certificate validation behaviour and passed

all test cases. In addition, Wget always reported the correct rejection reason whenever it encountered an invalid certificate. The mean test execution duration of three test runs was approximately 8 seconds.

	W3M v.0.5.3	cURL v.7.47.0 GnuTLS v.3.4.10	cURL v.7.47.0 OpenSSL v.1.0.2g	Wget v.1.17.1
Test ID				
1	✓	✓	✓	✓
2	✓	✓	✓	✓
3	✓	✓	✓	✓
4	✓	✓	✓	✓
5	✓	✓	✓	✓
6	✓	✓	✓	✓
7	✓	✓	✓	✓
8	✓	✓	✓	✓
9	✓	✓	✓	✓
10	✓	✓	✓	✓
11	✓	✓	✓	✓
12	✓	✓	✓	✓
13	✓	✓	✓	✓
14	✓	✓	✓	✓
15	✓	✓	✓	✓
16	-	-	-	-
17	✓	✓	✓	✓
18	✓	✓	✓	✓
19	✓	✓	✓	✓
20	✓	✓	✓	✓
21	✓	✓	✓	✓
22	✓	✓	✓	✓
23	✓	✓	✓	✓
24	✓	✖	✓	✓
25	✓	✓	✓	✓
26	✓	✓	✓	✓
27	✓	✓	✓	✓
28	✓	✓	✓	✓
29	✓	✖	✓	✓
30	✓	✓	✓	✓
31	✓	✓	✓	✓
32	✓	✓	✓	✓

Table 7.3: Comparison of Certificate Validation in Applications

7.2.2 Exemplary Regression Testing of OpenSSL

This section demonstrates the regression testing use case of the workbench by analyzing the certificate validation behaviour of OpenSSL [6] over multiple versions. Therefore the *OpensslAdapter* class was implemented to allow instrumentation of the *s_client* programm - a lightweight TLS client that can connect to a given host and port combination - which is contained by default in OpenSSL distributions. The OpenSSL source code of multiple versions was downloaded and compiled, and the workbench was executed on each downloaded version. Our evaluation compares the releases 1.1.0, 1.1.0*a*, 1.1.0*b*, 1.1.0*c* and 1.1.0*d*, which are published on the official project website [7]. Downloading, compiling and installing these OpenSSL releases from source and thereafter initiating the testing process was automated by a shell script. Table 7.4 illustrates that the certificate validation behaviour did not change over these versions that were exemplarily analyzed. All versions are passing every test case except for test case 30, as OpenSSL is not deploying hostname validation by default. All certificates that are rejected by OpenSSL are rejected for the correct reason.

[6] OpenSSL Cryptography and SSL/TLS Toolkit, `https://www.openssl.org`; last accessed on 2017/02/27
[7] OpenSSL Source Downloads, `https://www.openssl.org/source/old/`; last accessed on 2017/02/27

Version	1.1.0	1.1.0a	1.1.0b	1.1.0c	1.1.0d
Released	25-Aug-2016	22-Sep-2016	26-Sep-2016	10-Nov-2016	26-Jan-2017
Test ID					
1	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓
6	✓	✓	✓	✓	✓
7	✓	✓	✓	✓	✓
8	✓	✓	✓	✓	✓
9	✓	✓	✓	✓	✓
10	✓	✓	✓	✓	✓
11	✓	✓	✓	✓	✓
12	✓	✓	✓	✓	✓
13	✓	✓	✓	✓	✓
14	✓	✓	✓	✓	✓
15	✓	✓	✓	✓	✓
16	-	-	-	-	-
17	✓	✓	✓	✓	✓
18	✓	✓	✓	✓	✓
19	✓	✓	✓	✓	✓
20	✓	✓	✓	✓	✓
21	✓	✓	✓	✓	✓
22	✓	✓	✓	✓	✓
23	✓	✓	✓	✓	✓
24	✓	✓	✓	✓	✓
25	✓	✓	✓	✓	✓
26	✓	✓	✓	✓	✓
27	✓	✓	✓	✓	✓
28	✓	✓	✓	✓	✓
29	✓	✓	✓	✓	✓
30	✓	✓	✓	✓	✓
31	✓	✓	✓	✓	✓
32	✖	✖	✖	✖	✖

Table 7.4: Certificate Validation Regression Testing of OpenSSL

Chapter 8

Discussion

This chapter discusses the impact of the weaknesses discovered in the evaluation of applications in Section 7.2. This chapter explains which vulnerability can result from the respective validation weakness and exemplarily illustrates how an attacker can exploit that vulnerability.

cURL 7.47.0 compiled with GnuTLS 3.4.10 as backend TLS library was discovered to deploy incromprehensive key usage validation. Incorrectly validating the key usage extension [14, Section 4.2.1.3] or the extended key usage extension [14, Section 4.2.1.12] of a certificate breaks the concept of key usage restrictions. Limiting the competence of a specific key material to one or several distinct purposes, such as authentication, key encipherment, key agreement or sub-certification, leverages the separation of concerns paradigm and potentially increases security. Using different keys for different purposes might help that the disclosure of a single private key only negatively affects the context of its designated purpose, while other key usage scenarios remain unaffected [23, Section IX.D]. For example, the disclose of a private key of a Certification Authority designated for key encipherment might allow the attacker to impersonate as the CAs website, but does not necessarily imply the loss of trust in certificates issued by this CA, as issued certificates have been signed with another key material designated for subcertification. Consequently, not validating key usage restrictions eliminates that possibility of loss limitation. Another feature of key usage restrictions is the opportunity for restricting privileges. For example, the internal CA of a huge software company can authorize one if its development departments to use their key material for code-signing, but not for server authentication for the company's website. Not validating key usage correctly eliminates that possibility of privilege restriction.

The second weakness discovered during evaluation is the lack of hostname validation by the OpenSSL library in default configuration. This behaviour is actually less an security flaw of the library itself, but rather an mannerism and questionable design

choice of the OpenSSL developers. The test results of the workbench show that the tested OpenSSL correctly validates inputted certification paths, thus correctly reporting to relying applications whether the inputted certification path is valid and trustworthy. Consequently, applications that use the OpenSSL API are ensured that the TLS entity presenting the certification path is really the entity it claims to be. However, hostname validation is one important additional step: it ensures that the pretended entity, whose authenticity was approved by validating the certification path, is indeed the correct entity we intended to communicate with. If this verification is omitted, an authenticated active Man-in-the-Middle (MitM) can interfere the communication between two end points and thereby violate all protection goals, e.g. confidentiality or data integrity, without being recognized.
Developers that use the OpenSSL library must be aware of this mannerism of not executing hostname validation by default. And even if aware, they must not forget to do the respective method calls; or elsewise risking a severe MitM vulnerability.

Chapter 9

Conclusion and Future Work

This thesis elaborated that correct validation of X.509 certificates is crucial for security in communication systems, as X.509 constitudes a Public Key Infrastructure (PKI) that is widely used to establish trust between entities. Such an infrastructure has only significance if all participating entities correctly validate certificates. The thesis elaborated that certificate validation is an extensive and consequently error-prone process, which needs to be accomplished by every application that implements X.509. By analyzing the formal certificate validation process specified in RFC 5280 [14], it was pointed out that correctly validating certificates not only consists of validating basic information included in a single certificate, but also consists of validating whole certification chains up to a ultimately trusted root certificate, meanwhile considering all chain dependencies. Based on this analysis, test cases for testing the certificate acceptance and rejection behaviour of applications were extracted and phrased into a test suite.
As consequence of the complexity of the validation process, this thesis deduced that a testing tool is needed to enable automated, systematic and reproducible testing of certificate validation routines in applications. In the proceeding of the thesis, a system approach for such a testing tool was developed and functional and technical requirements were identified. Subsequently, the concrete instantiation of such a testing tool, destined for testing TLS-enhanced applications, was designed and named *X.509 workbench*. This workbench was implemented and offers high flexibility and extensibility, as it is not bound to a predetermined list of applications that can be placed into its testing environment, nor is the workbench bound to a specific test suite that is executed on any application in question. Nevertheless, the test cases identified during analysis phase were implemented as a set of test certificates and constitude a default test suite that can be used by the workbench out of the box.
In the evaluation chapter the workbench was exemplarily applied to several command-line applications. While two command-line tools successfully passed all test cases, it was spotted that the application cURL compiled with GnuTLS as backend TLS library spuriously accepts certificates with wrong key usage indication, whereas combined with

OpenSSL as TLS library, cURL correctly rejected certificates with wrong key usage indication. Moreover, testing the *s_client* tool which is part of the OpenSSL library directly without a third-party application in between, it turned out that OpenSSL deploys no hostname validation by default. This circumstance leaves great responsibility on applications to deploy their own - and hopefully correct - hostname validation mechanisms; or elsewise risking a severe Man-in-the-Middle vulnerability. Even recent releases of OpenSSL that support hostname validation do not apply it by default, thus leaving developers in responsibility to manually opt-in that functionality [29]. Consequently, developers must be aware of this mannerism and must not forget to do the respective method calls.

The workbench which is developed within the bounds of this thesis is explicitly designed to easily allow integration of future work in terms of custom test suites and application adapters. Future work might identify additional, more restrictive test suites, e.g. checking certificate policies, name constraints, certificate revocation states or any other critically marked optional certificate extensions. Besides custom test suites, future work might also apply the workbench to applications other than those examined in the evaluation chapter. Corresponding new adapter implementations can be given at workbench startup and dynamically loaded at runtime. Especially testing certificate validation routines inside Google's mobile operating system *Android* [1] and mobile apps might be subject to interesting future work.

The test result outputs generated by the workbench are formatted using the Extensible Markup Language (XML), thus allowing easy processing and integration into arbitrary other projects. Through aiming for flexibility and extensibility of the workbench, the author of this thesis hopes that the workbench becomes a useful and handy tool for analyzing certificate validation in applications, that can be integrated into the larger context of other security related projects.

[1] Android Operating System, `https://www.android.com`; last accessed on 2017/03/02

Bibliography

[1] S. Haunts, “Cryptography in .NET : RSA,” https://stephenhaunts.com/2013/03/26/cryptography-in-net-rsa/; last accessed on 2016/12/28.

[2] United States Naval Academy - Cyber Sciene Department, “Asymmetric Encryption,” https://www.usna.edu/CyberDept/sy110/lec/cryptAsymmEnc/lec.html; last accessed on 2016/12/28.

[3] I. Grigorik, “Transport Layer Security (TLS),” https://hpbn.co/transport-layer-security-tls/; last accessed on 2016/12/28.

[4] “Transport Layer Security,” https://upload.wikimedia.org/wikipedia/commons/6/61/TLS_protocol_stack.jpg; last accessed on 2016/12/28.

[5] “Cross-certification between two PKIs,” https://en.wikipedia.org/wiki/X.509; last accessed on 2017/02/06.

[6] CVE Details, “OpenSSL Vulnerability Statistics,” http://www.cvedetails.com/product/383/Openssl-Openssl.html?vendor_id=217; last accessed on 2017/03/07.

[7] CVE Details , “GnuTLS Vulnerability Statistics,” http://www.cvedetails.com/product/4433/GNU-Gnutls.html?vendor_id=72; last accessed on 2017/03/07.

[8] J. Schmidt, “GnuTLS Goto Fail,” 2014, https://www.heise.de/security/meldung/Sicherheitsluecke-GnuTLS-jetzt-mit-goto-fail-2133192.html; last accessed on 2017/03/07.

[9] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, “The most dangerous code in the world: Validating ssl certificates in non-browser software,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382204

[10] Y. Ding, “SSL sicher implementieren,” *Datenschutz und Datensicherheit-DuD*, vol. 38, no. 12, pp. 857–861, 2014.

[11] C. Eckert, *IT-Sicherheit: Konzepte, Verfahren, Protokolle*, 8th ed. Oldenbourg, 2013.

[12] T. Roeder, “Asymmetric-Key Cryptography,” https://www.cs.cornell.edu/courses/cs5430/2013sp/TL04.asymmetric.html; last accessed on 2016/12/28.

[13] Apache Software Foundation, “SSL/TLS Strong Encryption: An Introduction,” http://httpd.apache.org/docs/2.4/en/ssl/ssl_intro.html; last accessed on 2016/12/28.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008, updated by RFC 6818. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[15] M. Nystrom and B. Kaliski, “PKCS #10: Certification Request Syntax Specification Version 1.7,” RFC 2986 (Informational), Internet Engineering Task Force, Nov. 2000, updated by RFC 5967. [Online]. Available: http://www.ietf.org/rfc/rfc2986.txt

[16] J. Schaad, “Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF),” RFC 4211 (Proposed Standard), Internet Engineering Task Force, Sep. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4211.txt

[17] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP,” RFC 6960 (Proposed Standard), Internet Engineering Task Force, Jun. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6960.txt

[18] T. Roeder, “Survival guides - TLS/SSL and SSL (X.509) Certificates,” 2016, http://www.zytrax.com/tech/survival/ssl.html; last accessed on 2016/12/28.

[19] C. Adams, S. Farrell, T. Kause, and T. Mononen, “Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP),” RFC 4210 (Proposed Standard), Internet Engineering Task Force, Sep. 2005, updated by RFC 6712. [Online]. Available: http://www.ietf.org/rfc/rfc4210.txt

[20] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[21] K. Kiyawat, “Do Web Browsers Obey Best Practices When Validating Digital Certificates?” Ph.D. dissertation, Northeastern University Boston, 2014.

[22] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, “Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps,” in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS’14*. Citeseer, 2014.

[23] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *IEEE Symposium on Security and Privacy*, 2014.

[24] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 793–804. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786835

[25] S. Lloyd, "Understanding Certification Path Construction," *PKI Forum White Paper*, 2002.

[26] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework," RFC 3647 (Informational), Internet Engineering Task Force, Nov. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3647.txt

[27] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Internet X.509 Public Key Infrastructure: Certification Path Building," RFC 4158 (Informational), Internet Engineering Task Force, Sep. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4158.txt

[28] Individual Contributors, "Cryptography.io Python Library Reference," https://cryptography.io/en/latest/x509/reference/#x-509-certificate-builder; last accessed on 2017/03/10.

[29] OpenSSL Software Foundation , "OpenSSL Wiki," https://wiki.openssl.org/index.php/Hostname_validation; last accessed on 2017/03/02.