CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Digital Signature Verification in PDF
Student:	Tomáš Stefan
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2018/19

Instructions

1. Study the current theory, recommendations and best practices of digital signatures.
2. Research the current state of the art in digital signatures of PDF documents on the Linux operating systems.
3. Discuss the security aspects of signature verification, e.g. certificate storage, revocation, etc.
4. Write a C or C++ library for verification of digital signatures in PDF's.
5. Demonstrate the use of this library by developing a command line application for PDF signature verification.
6. Discuss your results, with focus on their security aspects.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague November 7, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Digital Signature Verification in PDF

Tomáš Stefan

Department of Computer Systems
Supervisor: Ing. Josef Kokeš

May 10, 2018

Acknowledgements

I would like to express my deep gratitude to my supervisor Ing. Josef Kokeš for his constructive suggestions and quick reviews. I would also like to thank my family for their support throughout my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 10, 2018 ....

Citation of this thesis

Stefan, Tomáš. *Digital Signature Verification in PDF*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

The subject of the presented thesis is the area of digital signatures with special attention to their use in PDF files. A short introduction containing basic principles and a summary of essential features are provided, as well as the basics of the PDF file structure. Different types of digital signatures in PDF files are described in more detail. A way to verify the validity of the basic PDF digital signature in Linux is demonstrated with a library written in C and a simple command line application.

Keywords digital signature, verification, PDF, C library, Linux

Abstrakt

Tématem předkládané práce je oblast digitálního podpisu se zvláštním důrazem na jeho použití v PDF souborech. Práce poskytuje stručný úvod do problematiky včetně základních principů, podstatných vlastností, popisuje také základní strukturu PDF souboru. Podrobněji se zabývá různými typy digitálních podpisů, které se v PDF souborech vyskytují. Způsob ověření základního druhu digitálního podpisu PDF souboru v Linuxu je demonstrován pomocí knihovny v jazyce C a jednoduché konzolové aplikace.

Klíčová slova elektronický podpis, ověřování, PDF, C knihovna, Linux

Contents

List of Figures

List of Tables

Introduction

The Portable Document Format (PDF) is a widespread format used to store and interchange various documents and publications. The main advantage of this format over its competitors is the portability. With PDF, it is almost certain that the document looks exactly the same for everyone, and on various devices (included printed sheets).

These days, PDF is not just about showing text and images; it can contain much more (forms, music, video, controllable three-dimensional models, and also some security features). The security features available in PDF include content encryption, digital signatures, and timestamps. This thesis will focus on the area of digital signatures in PDF.

The goal of the thesis is to introduce digital signatures in PDF, discuss the security aspects of signature verification, and finally provide a C/C++ library for verification of digital signatures in PDF together with a proof-of-concept command line application using this library, both targeting primarily the Unix-like OS.

The main motivation for looking into details of the subject is the lack of implementations of this feature in most of the libraries and applications in the Linux world.

Chapter 1 provides a general introduction to the digital signature world, basic principles of signing documents and signature verification. The structure of the PDF file with the focus on the digital signatures, their types and transform methods is described in chapter 2. The overview of applications and tools that work with PDF files in Linux, especially with regard to their support for the digital signatures, is presented in chapter 3. The practical contribution of the author in the form of a library for the digital signature verification in a PDF document is described in chapter 4 together with a simple command line application that demonstrates the functionality. Finally, the achieved results are subjected to a critical evaluation.

Chapter 1

Digital signatures

The most common way of authentication that has been in use for centuries is a handwritten signature. It is a form of behavioral biometric and is easier to forge than other modern biometric methods such as fingerprint or iris scan. The process of verification of handwritten signatures was improved with modern technologies. In addition to a static (off-line) signature, where the verification is performed on the resulting image, there is a new dynamic (on-line) type of signature written using a digitizing tablet, which also takes into account dynamic properties such as pressure, time, and a number of strokes. [1]

However, all this progress was not sufficient for some of the needs in current applications. And that is where the digital signature comes to play.

For simplicity, let us imagine digital signatures as some kind of "magic" that ensures a few important properties to the signed data. The following holds not only in the context of PDF but generally with various kinds of objects. These properties are:

Authenticity

means everyone can verify that the declared person really created the digital signature over some data (it is called signing the data). It is important to note that digital signature cannot exist on its own. It always has to be assigned to some data.

Data integrity

ensures it is always possible to decide whether the signed data were manipulated in any way or not. And it does not matter if the manipulation is a complete rewrite of the document or just addition of one space at the end that people will not even notice.

Non-repudiation

provides another safety mechanism important for real-world application of digital signatures. It means the author of the digital signature cannot deny signing the data. [2]

There is also one special kind of digital signature called timestamp. The timestamp proves the signed data were created before a specified time. It does not tell exactly when. It could be a second before or it could be a ten years before. But after the timestamp has been applied to the data, there is no doubt the document existed at the moment of signing.

After putting all that together, we have everything we need as an electronic equivalent to a handwritten signature. It is even better in some aspects, but everything has its pros and cons.

1.1 Signature in general

The process of applying a digital signature begins with hashing the data. There is a specific category of functions called hash functions. These hash functions take variable length data (from messages with zero length to really large like whole drive images) as an input and produce a constant length output that is relatively short. The output is called a message digest or sometimes is referred to as a hash. The whole original message has to be involved in some way to ensure data integrity while using the hash allows us to reduce the size of the signature as well as the time needed[1] for both creation and verification of the signature.

Not all hash functions are suitable in the context of digital signatures. With so-called secure hash functions it is computationally infeasible to:

- find two different messages that will produce exactly the same output of a hash function (collision of the first kind)
- find a different message to the given one that will result in exactly the same output of a hash function (collision of the second kind) [3]

Although the two problems seem to be very similar, their complexity is significantly different. To find a collision of the first kind with a 50% probability, the total number of messages to try is approximately $2^{n/2}$, where n is the length of the message digest (in bits). This is a much lower number than expected. The reason behind this is called the birthday paradox. However, to find a collision of the second kind with a 50% probability, the total number of messages to try is much higher, close to 2^n. [4]

To give an example what this really means, let us consider finding a collision with a 256-bit hash function, e.g. SHA-256. The number of possible

[1]The asymmetric encryption is a relativelly slow process.

messages to go through to have a 50% probability of finding a collision of the first kind is approximately 2^{128}. However, with added restriction of finding a collision to the given message, the number increases close to 2^{256}.

Details of some commonly used hash functions in digital signatures are shown in Table 1.1. [5, 6]

Table 1.1: Common hash functions in digital signatures

Algorithm	Input size (bits)	Hash size (bits)
SHA-1	$< 2^{64}$	160
SHA-256	$< 2^{64}$	256
SHA-384	$< 2^{128}$	384
SHA-512	$< 2^{128}$	512
RIPEMD160	$< 2^{64}$	160

When talking about secure hash functions, it is also important to note that SHA-1 is already considered deprecated and other hash functions should be used instead. The reason behind is that the first collision has already been found. The project Shattered succeeded in finding two different PDF files producing the same message digest with SHA-1 (collision of the first kind). [7]

Another important concept is encryption, especially asymmetric encryption. There, people operate with two keys. One is called a public key and can be shared with everyone else. This one is used for encryption of the data. The other key is called a private key, shall be kept secret and the original data can be restored only with the knowledge of this private key. [3]

In a typical use case, one person (traditionally called Alice) provides her public key over a (possibly insecure) channel to other people (like uploading to a web page). Another person (Bob) wants to send a secret message to Alice. To do so, he encrypts the data using Alice's public key and sends the message. Only Alice knows the private key that can decrypt the message.

In digital signatures, the asymmetric cryptography is used in exactly the opposite way. Again, everyone can know the public key and the private key shall be kept secret. However, the process of signing is done using the private key and verification with the public key. The signing is, in fact, an encryption of the hash using the private key and attaching the value to the original data. And verification of the digital signature includes decryption of the signature and comparison to a computed value over the same data. The signing phase is shown in Figure 1.1, and the verification phase is in Figure 1.2.

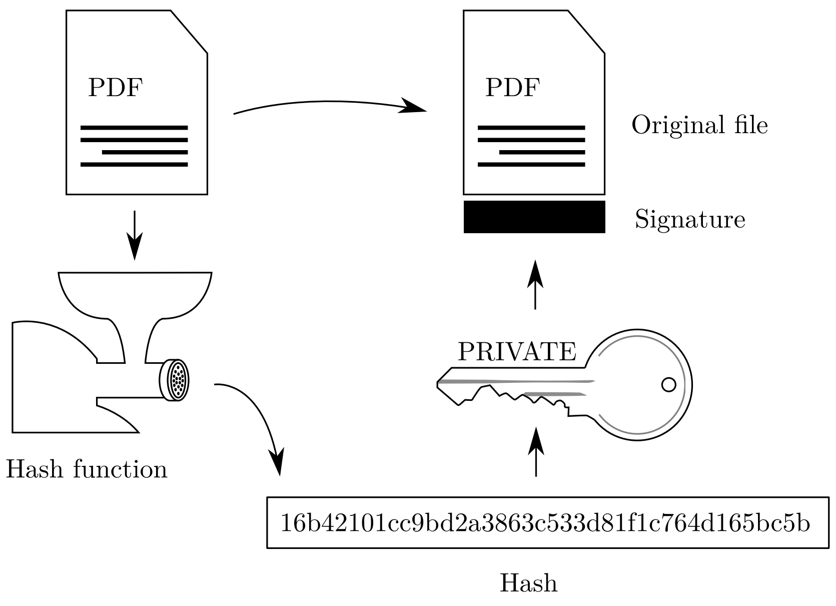


Figure 1.1: Signature generation

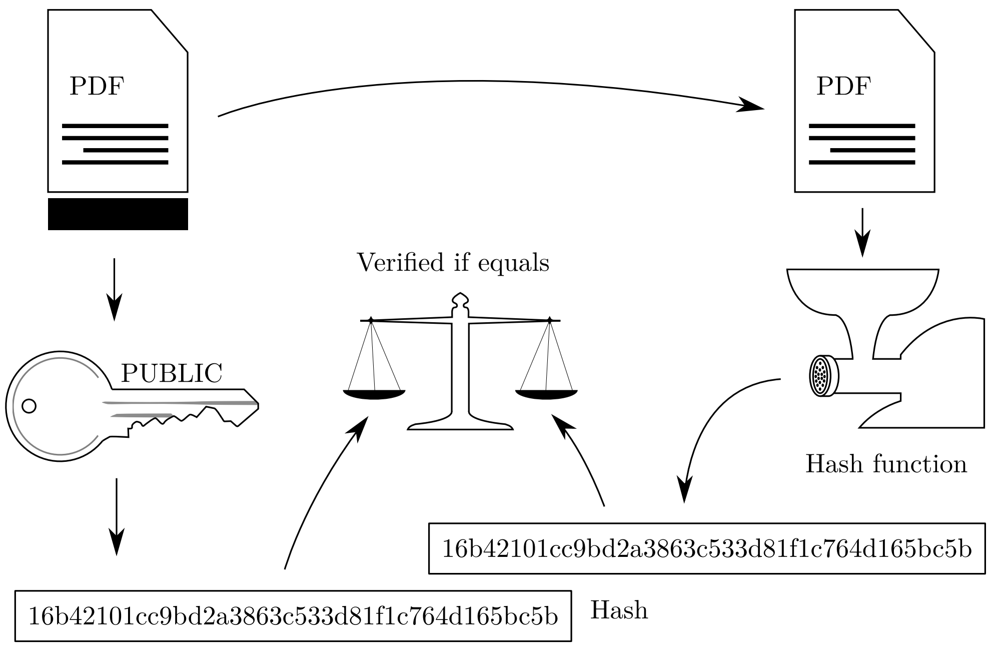

PDF
PDF
Verified if equals
PUBLIC
Hash function
16b42101cc9bd2a3863c533d81f1c764d165bc5b
16b42101cc9bd2a3863c533d81f1c764d165bc5b
Hash

Figure 1.2: Signature verification

1.2 Public Key Infrastructure

The Public Key Infrastructure (PKI) is an entire complex system intended to make secure communication possible in an untrustworthy environment without previous interaction (exchange of keys, etc.). The price for this is that both parties have to trust some other body, typically called the Certification Authority (CA). The CA is responsible for verifying end-user identities and for issuing and revocation of certificates.

The certificate is composed of the public key and identification of the holder, serial number, dates of creation and expiry, and other information – all digitally signed with the issuer's private key. The important point is that it has no knowledge of holder's private key. The verification is performed using the certificate of the issuer which contains the public key. In real world applications, the CA usually does not sign the end-user certificate directly, because in case the private key is compromised, all the issued certificates can no longer be trusted. Typically, one or more intermediate CAs are set up that create the chain of trust leading to the root CA (with a certificate signed by itself). Its private key is used quite seldom and can be securely stored (e.g. on a hardware security module stored at a very safe place). The verification has to follow the whole chain up to the commonly agreed root CA. Consulting the CRL (Certificate Revocation List) or using other means like the OCSP protocol to check for revoked certificates should be an integral part of the process. [8]

1.3 Security aspects

The certificate storage is a place used for saving trusted root CA certificates. Typically, there is a certificate storage provided by the operating system (Windows, Linux, etc.) for other applications. Nevertheless, some applications come with their own storages (e.g. Firefox web browser). In order to simplify the initial setup for users who do not know anything about PKI, the storage is usually pre-filled with trusted CA certificates. This approach greatly simplifies the initial setup (or avoids it completely) and, for example, allows the user to start web browsing with HTTPS immediately. The set of trusted root authorities is kept up-to-date together with the other parts of the operating system or the application. The disadvantage is that someone else decides for the user whom does the user trust and whom not, regardless of their opinion. We have already seen cases of commonly trusted CAs losing their credit because of severe security flaws in the past. [9]

1.3.1 Symantec CA

Cases of CAs that finally had to be removed from trusted stores include DigiNotar or StartCom/WoSign. However, the most recent affair at the time of writing also known to the general public concerns the Symantec CA. The list of some of its known wrongdoings:

- issuance of RSA 1024-bit certificate expiring after deadline
- unauthorized EV (Extended Validation Certificate) issuance by RAs (Registration Authority)
- test certificate misissuance – for unregistered domains or domains owned by someone else
- domain validation vulnerability – in some cases an attacker could obtain certificate to somebody else's domain
- numerous violations found during the 2015 audit
- SHA-1 issuance after the deadline
- unallowed cross-signing with other CA
- UniCredit sub CA failing to follow BRs (Baseline Requirements)
- audit reports (required by BR) regularly issued late
- CrossCert misissuances
- GeoRoot program audit issues

For more information see [10]. In response to these issues, the major web browsers decided to remove a trust for Symantec. The first step is distrusting the certificates issued before June 1, 2016. In Google Chrome, this will be done in version 66 (April 2018) and in Mozilla Firefox with a version 60 (May 2018). The second step is to remove the trust completely. This Symantec cut-off will be done in Chrome with a version 70 (October 2018) and in Firefox with a version 63 (October 2018). [10, 11]

Finally, it should be noted that the whole business of the Symantec CA was taken over by DigiCert, Inc., completely reviewed, redesigned and made subject to their own strict regulations. This made it possible for the trademark Symantec to live on, but the security processes behind it are completely new. [12]

1.3.2 Revocation

Another problematic part is a revocation of issued certificates. There are many reasons to revoke an issued certificate, ranging from technical problems like compromising the private key to standard procedures, e.g. when an employee is leaving a company. The information of revocation needs to be distributed to all subjects that could be possibly verifying the specific certificate (mostly this means the whole world). There are a few approaches to achieve this and none of them is fundamentally better than the other:

CRL (Certificate Revocation List) – a list of all revoked certificates published by the CA. The main problems are the big size of the list for great CAs, possible delay between the revocation and publishing the updated list, because lists are cached and updated periodically, and unavailability of the list due to network problems or Denial of Service attacks.

OCSP (Online Certificate Status Protocol) – an alternative to the CRL with an advantage of cutting down requirements for the network bandwidth, client resources and the delay from the act of revocation to its availability to consumers. The client requests the revocation status at the OCSP responder (a server published in the certificate) and that responds to a certificate with a reply of *good*, *revoked*, or *unknown*. The drawback is that OCSP is even more prone to network problems as it requests the status each time, not just doing periodic sync as is the case with the CRL. It is not resistant to MitM (Man in the Middle) and replay attacks.

OCSP-Stapling is another approach supported by major browsers and ideally shall be enforced by OCSP Must-Staple flag to prevent the MitM attack. This moves the step of consulting the OCSP responder from the client to the server which staples the signed response in the handshake with the client. [13, 14, 15]

Another step to hardening current PKI infrastructure is a technology called the **Certificate Transparency** (CT) that should allow much faster detection of any bad behaviour. This system consists of three parts – logs, monitors, and auditors. The main idea is that there are some logs with all the certificates issued by publicly trusted CAs. They allow anybody to find a malicious certificate or detect the CA with unfair practices. The public logs use the Merkle hash tree data structure and allow only appending new records. The addition can be done by anybody, same as querying for log records to verify log correct behaviour, or that the specific certificate is present. The monitors are publicly run servers that watch over the logs. They perform periodic checks and look for suspicious certificates. Finally, the auditors are verifying that the logs are cryptographically consistent and also can verify that a particular certificate is present in the log. Nowadays, the accepted policy for publicly trusted CAs is to publish all the issued certificates in the CT logs. The auditor part could be integrated directly into applications (like web browsers) to check for the status of the certificate provided by the other peer. [16]

Discovery of the asymmetric cryptography and the on-going research in strong hash functions made it possible to securely and provably authenticate the source of digital data. With the help of the digital certificates and the whole PKI based on globally accepted and trusted CAs, we can effectively use this authentication worldwide. However, despite the solid mathematical grounds of the technology the real world environment brings new challenges, often caused by an improper implementation, insufficient ease of use for the general public and intentional or unintentional misuse for whatever reason. Yet, the challenges are being addressed by professionals and the system is being globally successfully used.

Chapter 2

Signatures in PDF

2.1 Portable Document Format

The Portable Document Format was developed by Adobe Systems; version 1.0 was published in 1993. Growing popularity of the originally proprietary format led to ISO standardization in 2008 (PDF 1.7 as ISO 32000-1) and 2017 (PDF 2.0 as ISO 32000-2).

Now, let us have a look at the PDF file structure.[2] The file consists of various objects spread across the file referenced by offsets. The reader can effectively use random access to the required data which can significantly improve performance, especially in case of big files. On the other hand, there is a different challenge when using PDF files in a network environment (Internet) – the need to present a part of the information even before the whole contents has been downloaded. A special kind of PDF files (linearized PDF) can be used to an advantage in such situations. There, the objects are placed sequentially within the file, which allows for on-the-fly processing. [17, Annex F]

There is a one-line header with a PDF version at the beginning of the document. It starts with characters `%PDF-` followed by version numbers (major and minor) separated with a dot. At the moment versions 1.0 to 1.7 and 2.0 are applicable. For example, the header can contain `%PDF-1.7` or `%PDF-2.0`. [17, Section 7.5.2]

The header is followed by a file body, composed of a sequence of indirect objects. Inside the body, there is the actual content of the document. [17, Section 7.5.3]

The PDF standard [17, Section 7.3] defines eight basic types of objects:

Boolean

Values `true` or `false` representing a logical value.

[2]Here we assume PDF files with a cross-reference section only. There is also a variant called a cross-reference stream not considered here for the sake of simplicity.

Numeric

Integers or real numbers, e.g., `789`, `-4.56`.

String

Literal strings are enclosed in brackets. Only 8-bit character values may appear inside this data type. Example of a literal string: `(text)`.

Hexadecimal strings are enclosed in angle brackets and shall contain only hexadecimal characters. This is useful for including binary data. Example of a hexadecimal string: `<3ca5f81147>`.

Name

A unique sequence of 8-bit characters starting with a slash (`/`). The slash is not a part of the name. Example: `/SigFlags`.

Array

A heterogenous one-dimensional collection enclosed in square brackets – `[...]`. For example: `[/Name (John) false 32]`.

Dictionary

An associative table containing pairs of *key* and *value*. The *key* shall be a name object. Example: `<</Size 9 /Lang (en)>>`.

Stream

A stream begins with a dictionary containing information about the stream such as compression method. The stream data are enclosed in keywords `stream` and `endstream`. Example:

```
<</Type /XObject /Subtype /Image /Width 52 ... >>
stream
a75c838855b331c4...
endstream
```

Null

The keyword used for a null object (a way to tell there is nothing here) is `null`.

The cross-reference table is an important part that permits random access to objects inside the file. The table starts with a keyword `xref` followed by two numbers. The first number stands for the first object in the subsection and the other equals the count of entries in the subsection. The cross-reference table can have one or more subsections. For each additional subsection there are those two numbers followed by 20-byte entries, one per line. The exact format is `oooooooooo ggggg n eof`. The first ten digit number (ten times `o`) is the byte offset in the decoded stream from the beginning of the file. The second number (five times `g`) is the generation number of the object. The single `n` is the flag meaning in use entry. The other possible value of the flag is `f` meaning free entry. [17, Section 7.5.4]

An example of cross-reference table with two subsections:

```
xref
0 2                 % subsection beginning at object 0 and containing
                    % two entries
0000000000 65535 f  % first entry is always free with generation number
                    % 65535
0000001325 00000 n  % object number 1, offset 1325, generation number 0,
                    % in use entry
14 1                % beginning of another subsection starting at the
                    % object 14 and containing one entry
0000002518 00001 n  % object number 14, offset 2518, generation number
                    % 1, in use entry
```

Finally, before the end, there is a file trailer which is used to quickly find an offset of the cross-reference table and some other useful information. It begins with a keyword `trailer` followed by a direct dictionary object. After that, there are the last three lines of the file: keyword `startxref`, a number specifying the byte offset in the decoded stream of last cross reference section from the beginning of the file and `%%EOF` meaning the very end of the file. [17, Section 7.5.5]

Example of a file trailer:

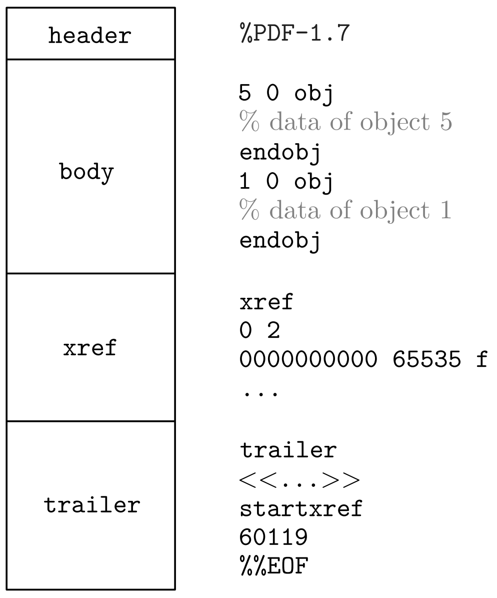
```
trailer
<</Size 11 /Root 10 0 R /Info 8 0 R
/ID[<77fa14e5e882571aeb5fd2f92d2ea001><77fa1...>]>>
startxref
12051
%%EOF
```

An illustration of the layout of individual parts mentioned above inside of PDF document is in Figure 2.1.

2.1.1 Incremental updates

When updating a PDF document incrementally, the changes should be appended to the end of the file. This update style leaves the original content untouched, only new data appear at the end along with a new trailer and cross-reference section. This is advantageous in the field of digital signatures, where the hash computed over the specified byte range preserves its value. [17, Section 7.5.6]

The illustration of incremental update can be seen in the Figure 2.2.



```
%PDF-1.7

5 0 obj
% data of object 5
endobj
1 0 obj
% data of object 1
endobj

xref
0 2
0000000000 65535 f
...

trailer
<<...>>
startxref
60119
%%EOF
```

Figure 2.1: PDF file structure

2.2 Signature in PDF

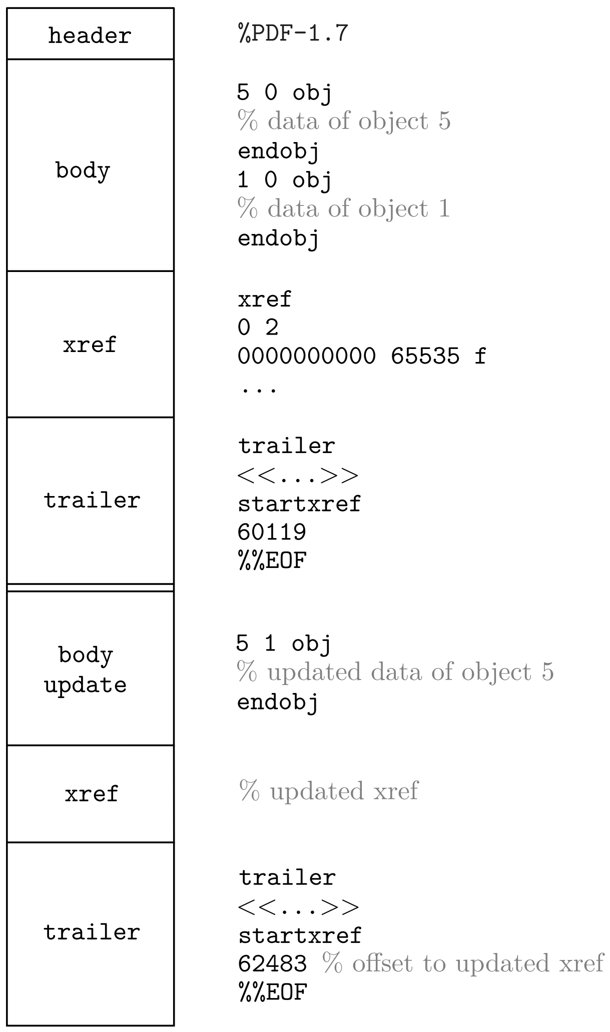
According to the PDF standard [17, Section 12.8.1]:

> *"Digital signatures in PDF support four activities:*
>
> - *the addition of a digital signature to a document,*
> - *the verification of the validity of a signature added to a document,*
> - *the addition of DSS*[3] *dictionaries and of validation related information (VRI) to allow for later verifications (see 12.8.4.4, "Validation-related information (VRI)"), and*
> - *the addition of document timestamp dictionaries (DTS) to allow for later verifications (see 12.8.5, "Document timestamp (DTS) dictionary")."*

The second activity (the verification of the signature validity) is the primary focus of this thesis.

If the signature is present in the file, the signature information is stored in a *signature dictionary*. Creating multiple digital signatures for a PDF file is possible. The PDF standard [17, Section 12.8.1] allows for these types of signatures:

[3]Document Security Store



```
%PDF-1.7

5 0 obj
% data of object 5
endobj
1 0 obj
% data of object 1
endobj

xref
0 2
0000000000 65535 f
...

trailer
<<...>>
startxref
60119
%%EOF

5 1 obj
% updated data of object 5
endobj

% updated xref

trailer
<<...>>
startxref
62483 % offset to updated xref
%%EOF
```

Figure 2.2: PDF incremental update

Certification signature

is applied by the author of the document and besides the proof of authorship provides an instrument for defining restrictions for later modification of the document. Only one signature of this type is allowed.

Approval signature

is usually applied after the certification signature by the recipients, typically expressing agreement with the contents. More of them are allowed.

Timestamp signature

provides reliable information about the time of signing. It may also be used to extend the time when it is possible to verify some previously applied signature. There may be any number of timestamp signatures applied to the document.

Usage rights signature

allows setting rights for the manipulation with specific parts of the documents, such as modification of annotations or forms. Only one signature of this type is permitted, and it shall be the first signature if used. The usage rights signature is deprecated in PDF version 2.0.

Three of these signatures, the certification, approval, and timestamp are placed in the *signature dictionary* together with the `ByteRange` entry. The last one, the usage rights signature, is referenced from the `UR3` entry in the *permissions dictionary.* [17, Section 12.8.1]

Some types of PDF documents may evolve during their life-cycle: form fields may be filled-in, annotations added, etc. The original author may define which changes are permitted without invalidating the signature. This is done by adding the *transform method* to the signature dictionary (more later in section 2.3). These define permissions for later document modifications. The certification signature shall have the `DocMDP` transform method or also the `FieldMDP` is possible. The approval signature may have only the `FieldMDP` transform method. And finally, the usage rights signature always has the `UR` transform method. [17, Section 12.8.2]

The *signature dictionary* contains two important entries for determining of how to manipulate with the signature. The first one is `Filter` with the value of the preferred signature handler. The value may not necessarily match with the signature handler used. The other entry – `SubFilter` – exactly identifies the signing procedure and allows for a signature handler interoperability. In case it is missing, only the corresponding signature handler can be used. The value of `SubFilter` is the exact identification of the signature used. [17, Section 12.8.3.1]

Table 2.1: PKCS#1 signature properties

SubFilter	adbe.x509.rsa_sha1
Message Digest	SHA-1 SHA-256 SHA-384 SHA-512 RIPEMD160
RSA	up to 4096-bit

2.2.1 PKCS#1

The Public-Key Cryptographic Standard #1 [18], besides other things (mainly RSA cryptography algorithm), defines the signature based on RSA. The only allowed PKCS#1 signature type in PDF is with the `SubFilter` value *adbe.-x509.rsa_sha1* (see Table 2.1 for details). This value has been deprecated in PDF version 2.0. It means that PDF writers are discouraged from using it, but PDF readers should process it for backward compatibility. [17, Section 12.8.3]

Important entries in the *signature dictionary* used with the PKCS#1 signature are listed here:

Type (name, optional); value *Sig*

SubFilter (name, optional); value *adbe.x509.rsa_sha1*

Contents (byte string, required); DER-encoded PKCS#1 binary data object containing the signature value

Cert (array/byte string, required); an array of byte strings, or just a byte string if the chain contains only one certificate; X.509 certificate chain, where the first one is the signing certificate

ByteRange (array, required); an array of pairs of integers, each item holding starting offset and length in bytes [17, Section 12.8.1]

2.2.2 PKCS#7 (CMS)

All defined `SubFilter` values using the Public-Key Cryptographic Standard #7 (Cryptographic Message Syntax) [19] together with other properties can be found in Table 2.2. The signature with `SubFilter` identification *adbe.pkcs7.sha1* has been deprecated in PDF version 2.0. The `Contents` entry in the *signature dictionary* is the DER-encoded CMS binary data object. The signer's certificate is not stored in the `Cert` entry as is the case with

Table 2.2: PKCS#7 signature properties

SubFilter	adbe.pkcs7.detached ETSI.CAdES.detached ETSI.RFC3161	adbe.pkcs7.sha1
Message Digest	SHA-1 SHA-256 SHA-384 SHA-512 RIPEMD160	SHA-1 SHA-256 SHA-384 SHA-512 RIPEMD160
RSA	up to 4096-bit	up to 4096-bit
DSA	up to 4096-bit	up to 4096-bit
ECDSA	ANSI X9.62, Elliptic Curve Digital Signature Algorithm	No

adbe.x509.rsa_sha1, but it is included directly in the CMS object itself. [17, Section 12.8.3.3]

The CMS object should contain:

- signer's X.509 certificate together with the full chain; if the chain cannot be added at the time of signing, it may be attached later as an incremental update to the Document Security Store (DSS)
- timestamp information (unsigned)
- revocation information (signed)
- one or more attribute certificates [17, Section 12.8.3.3]

The signatures with the `SubFilter` value of *ETSI.CAdES.detached* are referred to as PAdES (PDF Advanced Electronic Signatures) and are useful for long-term validation. They use one of the two CMS profiles compatible with CAdES (CMS Advanced Electronic Signatures), either PAdES-E-BES (Basic Electronic Signature) or PAdES-E-EPES (Explicit Policy Electronic Signature). These profiles correspond to those defined in ETSI EN 319 122-2 (CAdES-E-BES and CAdES-E-EPES). [17, Section 12.8.3.4]

To validate a PAdES signature, one is supposed to complete the following steps:

1. Compare the hash value of the signer's certificate with the hash value in the *signing-certificate* or *signing-certificate-v2* attribute and in case of the match, verify that the document digest is correctly signed using the public key from signer's certificate.
2. Validate the certification path according to RFC 5280 clause 6; if the signature handler knows for sure all the validation information existed at some point in the past, that time shall be used; otherwise, the handler has to use the current time
3. Validate the path of the certificate used for the timestamp according to RFC 5280 clause 6
4. Perform revocation checks of the certification path [17, Section 12.8.3.4]

2.3 Transform methods

Transform methods are techniques that allow some specified changes to the document without invalidating the signature. These changes could be filling-in forms (all, or just some), or adding annotations. The first step of verification is again the comparison of the hash. Then the signature handler has to take into account all the incremental updates to the document and for each change between the signed and recent version check whether it is allowed. [17, Section 12.8.2]

2.3.1 DocMDP

The `DocMDP` transform method [17, Section 12.8.2.2] allows to specify changes that are permitted and changes that are forbidden in the document. The `DocMDP` can be used only with the certification signature. If applied, the value of `TransformMethod` in the *signature reference dictionary* is *DocMDP* (name object), and value (optional) of `TransformParams` is *DocMDP transform parameters dictionary.* Possible entries in this dictionary:

Type (name, optional); the value is *TransformParams*

P (number, optional); access permissions for the document; possible values:

1 no changes are permitted, any changes invalidate the signature

2 permitted only filling in forms, instantiating page templates, and signing (default value)

3 in addition to *2*, permitted changes are annotation creation/deletion/modification

V (name, optional); the version of this dictionary; currently the only valid value is *1.2* (also the default value)

2.3.2 UR

The `UR` transform method [17, Section 12.8.2.3] is used with the usage right signature and together with the signature type it has been deprecated in PDF version 2.0. In order to verify a document with this transform method applied, the signature handler should first verify the byte range digest. The next step is to examine the current version of the document for any disallowed changes since the signature was applied. Value of the `TransformParams` is the *UR transform parameters dictionary.* Possible entries in this dictionary:

Type (name, optional); the value is *TransformParams*

Document (array, optional); an array of names; the only defined value is *FullSave* that permits to save the document; usage rights that permit modification of the document implies *FullSave*

Msg (string, optional); text with additional information, such as the reason for adding usage rights (UR)

V (name, optional); the version of this dictionary; currently the only valid value is `2.2` (also the default value, if not specified)

Annots (array, optional); an array of names specifying additional rights on the annotations; possible values: *Create, Delete, Modify, Copy, Import, Export, Online, Summary View*

Form (array, optional); an array of names with additional permissions for form fields. Possible values:

Add	add form field
Delete	delete form field
FillIn	fill-in form and save
Import	import form data in FDF, XFDF, or text (CSV/TSV) format
Export	export form data to FDF, XFDF
SubmitStandalone	submit form data without being open in Web browser
SpawnTemplate	instantiation of new pages from templates
BarcodePlaintext	encode form data to barcode
Online	SOAP or Active Data Object

Signature (array, optional); an array of names; currently the only defined name is *Modify* that permits the addition of a new signature to an existing signature form field and also erasing signed signature form field

EF (array, optional); an array of names; usage rights for named embedded files; possible values: *Create, Delete, Modify, Import*

P (boolean, optional); *true* means restrictions are valid; *false* (default value) means all restrictions are ignored

2.3.3 FieldMDP

The `FieldMDP` transform method [17, Section 12.8.2.4] detects changes in the values of a list of form fields. Value of the `TransformParams` is the *FieldMDP transform parameters dictionary.* Possible entries in this dictionary:

Type (name, optional); the value is `TransformParams`

Action (name, required); together with `Fields` entry specifies which form fields invalidate signature if changes are made to them

All no changes allowed to any of the form fields

Include changes are not allowed only for the form fields specified in the `Fields` entry

Exclude changes are allowed only for the form fields specified in `Fields` entry

Fields (array, required if `Action` is *Include* or *Exclude*); text strings containing field names

V (name, optional); the version of this dictionary; currently the only valid value is `1.2` (also the default value)

Several years of service and hundreds of millions of documents in circulation prove the PDF to be an exceptionally successful standard for storing the electronic documents. The possibility to use the digital signatures is certainly not the least important among the numerous features it provides. The standard continuously evolves and provides new features with respect to the advancements in modern cryptography and security practices. One can sign the whole document or its parts only, control the accepted contents processing (filling in forms, making additions, annotations, etc.) in detail, follow the document evolution in time and much more.

Chapter 3

Signature support in applications

Although completely platform independent by design, there is little doubt the operating systems from Microsoft, either MS-DOS in the beginning or different versions of Microsoft Windows later on, were the primary target of PDF documents. That is not surprising, because it was and still is by far the most widespread operating system for desktop computers at home as well as in the commercial environment. The two applications from the Adobe Systems, Acrobat for the document creation and editing and free Acrobat Reader for reading them, became *de facto* reference implementations of such programs and helped significantly to the success of the PDF itself. Other operating systems, namely macOS and Linux, hold only a small portion of the desktop market but they still do have their use-cases and their importance. However, one can notice a gradual move towards using mobile solutions of different kinds (tablets, smartphones) and recently even growing popularity of on-line services also for handling the PDF documents.

In the Linux world, where both the kernel and the applications are developed in a rare symbiosis of the commercial sector and independent community (see e.g. [20]), the right and liberty to choose has always been considered a precious value. So it is no wonder one can find numerous applications to view a PDF file in Linux ranging from very popular and well-known applications such as Evince or Okular to those being used by a small community only. Generally speaking, their handling the digital signatures is not complete or is missing entirely.[4] The reason seems obvious: the full coverage of all the subtle niches is far from trivial and the demand for the feature from users is

[4] One should pay some attention not to confuse adding the *electronic signature* and the *digital signature* to the document. The former term (often in the form *e-signature*) stands for adding a graphical representation of the hand-written signature of the author/reviewer to the document, usually in the form of a scanned image; the latter means real authentication of the document contents based on cryptography techniques.

Table 3.1: PDF viewers under Linux

Program	Open Source	Free	Digital Signature	Note
Evince	✓	✓	✗	[23]
Foxit PhantomPDF	✗	✗	✓	[24]
Foxit Reader[5]	✗	✓	✓	[24]
Ghostscript	✓	✓	✗	[25]
KPDF	✓	✓	✗	[26]
Master PDF Editor Commercial	✗	✗	✓	[27]
Multivalent	✓	✓	✗	[28]
Okular	✓	✓	✗	[29]
PDF.js	✓	✓	✗	[30]
PDF Studio Pro	✗	✗	✓	[31]
qpdfview	✓	✓	✗	[32]
Xpdf	✓	✓	✗	[33]
Zathura	✓	✓	✗	[34]

not too large [21]. Fortunately enough, there are examples of the contrary, but mostly in a closed source paid software. The LibreOffice project, a popular multi-platform general purpose home and office document handling software, is one of the quite rare notable exceptions in an open source world that can correctly handle both signing of PDF files and verifying their digital signatures [22]. Table 3.1 provides a summary of popular PDF viewers. The list tries to be more representative than exhaustive. The Adobe Systems discontinued the development of Acrobat/Acrobat Reader for the Linux platform in 2013.

There are also some other tools that make work with digital signatures in Linux possible. Many of them are written in Java. Even if they were primarily developed for other operating systems (mainly Microsoft Windows) they work well in Linux due to the multi-platform nature of Java, provided that they do not rely on platform-specific features[6]. Quite often this is acceptable, but sometimes the inter-operability problems, unavailability of Java runtime, higher system resources consumption or not completely seamless system integration may prove to be an issue. The Java virtual machine is a rather complex piece of software with security issues of its own which is also a point to consider. One can arrive at a very similar conclusion when running a native Microsoft Windows application under Linux using the Wine compatibility layer. Undoubtedly, the true native Linux solution would be indispensable in many situations and environments. Popular tools (except PDF viewers) ca-

[5]non-commercial use only
[6]for example system calls

Table 3.2: PDF signature related software

Program	Open Source	Free	Note
CAcert PDF Signer	✓	✓	command line Java application [35]
DigiSigner	✗	✗	commercial GUI application with limited free use [36]
iText	✓	✓	PDF library for Java and .NET environment [37]
jPdfSign	✓	✓	command line Java application [38]
jSignPdf	✓	✓	discontinued Java application [39]
LibreOffice	✓	✓	home and office document handling software [22]
OpenSignPDF	✓	✓	Java application [40]
PDFBox	✓	✓	Java library [41]
Poppler	✓	✓	PDF rendering library slowly getting digital signature support [42]
Portable Signer	✓	✓	GUI application written in Java [43]

pable of working with PDF signatures in Linux are listed in Table 3.2.

The overview provided here suggests that the possibilities of working with digital signatures in PDF files in Linux environment are somewhat limited, especially when running Java is not an option for whatever reason.

The expectations that the situation might rapidly improve with the implementation of digital signature features in Poppler still wait for fulfillment. Poppler is a general purpose PDF rendering library that is used by many projects, Evince, Okular, LibreOffice, and Zathura being among them. The progress is relatively slow and in spite of about a decade of work, it still has not reached the desired state [21].

The current Internet is even able to offer online services for the verification of the digital signatures in PDF documents, the Secured Signing company from New Zealand being an example [44]. When accessed with a standard web browser they are platform independent from their nature. Typically, they provide only a limited (if any) free access. Before using them, one should thoroughly consider the fact that uploading the documents for verification may result in sharing private and possibly restricted information with some other subject which makes it potentially dangerous. Document treatment of this kind may even violate legal regulations like the General Data Protection Regulation in the European Union.

CHAPTER 4

Implementation

In order to demonstrate the process of verifying the digital signature of a PDF document in practice and possibly to provide a little contribution to improve the above described somewhat bleak situation, we decided to develop a lightweight library with limited external dependencies (beside standard run-time libraries always present in the operating system not more than a cryptographic library). The new library should be able to verify the basic type of signature – PKCS#1. Further extensions of its capabilities are possible, however, due to the rapidly increasing complexity of the PDF document processing remain outside the scope of the presented work. It was explained in chapter 2 that comprehensive PDF structure awareness and custom file parsing would be required.

4.1 Design and preliminaries

It was necessary to make several decisions on technologies and tools used in the development in the beginning.

The form of a library was selected as a final result because it appears to be both the most lightweight and the most flexible solution which allows for a complete separation of the digital signature processing from a user interface of any kind. The library is generated in the form of a shared object (`.so`) which can be used by an application directly during the link/load-time or be loaded dynamically during the run-time. It also allows for simple versioning of the library.

The C language seems to be the most natural choice for writing a library for common use. The C compiler is readily available for all targets one can imagine (at least in the form of a cross-compiler in case of embedded systems). The resulting code has a small memory footprint, it is fast, efficient and possibilities of interfacing to it from other languages are practically endless. The advantages of higher-level programming languages like Java, Go, or C# would not out-weight the mentioned features in a project of this scale.

The library for the verification of digital signatures needs to perform some cryptographic operations. Namely, the asymmetric cryptography (for decryption of the signature) and a few popular hash functions need to be supported. Using a well-tested code with a long-term credit is the only reasonable option for a subtle subject which the cryptography surely is, which is why an external cryptographic library is used to perform these operations. The considered options were the ones most popular with the Linux OS – BoringSSL, GnuTLS, NSS, OpenSSL, and mbedTLS. Eventually, we chose OpenSSL; however, it is still possible to adapt the code for a different library later with little efforts.

The whole system of PKI is based on the concept of trusted root CAs. The library will have no pre-set trusted root anchors by default which gives the user full flexibility and detailed control of the verification process. The two possibilities to define the trusted root CAs employed in the library will be either using the trusted store provided by the operating system or defining the custom set of root certificates. It is upon the user to choose the favorite option, after taking into account things like regular updating, existence of the private CA (e.g. for internal documents within a company), adherence to only a few selected root CAs, and the like.

4.2 PDF-Sigil library (libpdfsigil)

The part **sigil** in the library name is derived from the Latin word *sigillum*, which means **seal** or **stamp**, i.e. something that has been used to prove the origin of the document, product, etc. for centuries.

The compiled library consists of a single file `libpdfsigil.so` and directly depends on the standard C library and the OpenSSL cryptographic library, only. It does not require any sort of initial setup (by means of the configuration file, calling a dedicated function or similar). The code of the library is entirely written in a plain C language. The compilation should be possible with an arbitrary ANSI C compliant compiler; there are no special requirements for amendments made in the latest versions of the C standard. We used the GNU Compiler Collection (GCC), and CMake build system.

The public interface consists of several functions plus some type and constant definitions, all provided by the header files (`sigil.h` for publicly exported functions, `types.h` for structure definitions and `constants.h` for definitions of various parameters and return codes). The process of verification of a digital signature in a PDF document is performed as a sequence of calls to the library functions which update the internal context of the type `sigil_t`, the structure that keeps track of defined parameters and results of previous steps during processing. Typically, the verification procedure follows these steps:

1. *Initialization* The call to `sigil_init` prepares the context for all subsequent operations.
2. *Data input* Providing the input data (i.e. the PDF document for the signature verification) can be done with the call to `sigil_set_pdf_path` (the document is referenced by its pathname within the mounted filesystem), `sigil_set_pdf_file` (the parameter is a valid file descriptor), or `sigil_set_pdf_buffer` (the document contents is passed as a memory buffer).
3. *Trusted roots* Setting the CAs that should be trusted is another essential step as there are no such items defined by default. It can be done with the help of `pdf_set_trusted_system` which will use all CAs in the operating system trusted store or alternatively by providing a custom set of trusted root certificates - `sigil_set_trusted_file` for a single root anchor and `sigil_set_trusted_dir` for all certificates available in the specified directory.
4. *Verification* The verification process is started with the call to the function `sigil_verify`.
5. *Results* Once the verification is carried out the results can be accessed using `sigil_get_result`. However, there are also other functions providing more detailed information. They may be useful especially in case of the verification failure.
6. *Clean-up* The final step in the sequence is done by the function `sigil_free` which makes sure all the allocated resources are properly freed.

In order to perform the verification of another PDF document, one should repeat all the described steps from initialization up to clean-up.

All the aforementioned functions indicate possible errors as their return value – `sigil_err_t` is a numerical type, possible values can be found in the header file. Using the predefined constants in the application source code makes it self-explanatory and well understandable. `ERR_NONE` which evaluates to 0 means the required operation was finished successfully; any other value means the failure of some kind. Using the function `sigil_err_string` is a straightforward way to obtain a text description of the error. More details about the error codes, function parameters, and function usage can be found in the documentation.

4.3 Command-line tool (pdf-sigil)

In order to demonstrate the use of the library, a command-line tool named **pdf-sigil** was developed.

When running the `pdf-sigil` command with a `-h` or `--help` argument, it prints the usage help. The PDF file for the verification is provided after the `-f` (`--file`) argument. Trusted root CAs need to be set in order to verify the signing certificate. This tool provides the same options for setting the trusted storage as the library – arguments `-ts` (`--trusted-system`) for using the storage provided by the operating system, `-tf` (`--trusted-file`) for setting the trusted certificate inside of a file, and `-td` (`--trusted-dir`) for all certificates inside the directory.

By default, the program prints the overall verification status together with a few additional pieces of information – SubFilter value, the hash function used, and intermediate verification results. This output can be extended with detailed information about the signing certificate with an argument `-ci` (`--cert-info`).

The return value is set according to the overall verification status: 0 for successful verification, 1 in all other cases regardless of the reason for failure. If no more than the return value is important (typically when used within shell scripts) the output can be suppressed entirely with an argument `-q` (`--quiet`). Appendix B provides a few example outputs created by the tool.

4.4 Results

The **PDF-Sigil** library was developed as a part of the presented thesis, making it possible to verify digital signatures in PDF documents. The library brings one external dependency only (OpenSSL for cryptographic operations) and is capable of verification of the certification signatures of the PKCS#1 type. PDF-Sigil has no trusted CAs pre-configured; the user is given full control over the trusted roots (either use the operating system-wide trusted store or provide the own set of trusted root certificates).

Testing PDF documents of various size and structure were provided with the certification signature. We used a free personal certificate from Comodo (dedicated primarily to the e-mail encryption and signing using S/MIME).[7] It is a 2048-bit RSA certificate with an SHA-256 certificate signature which expires in one year. The chain of trust goes through the intermediate `COMODO RSA Client Authentication and Secure Email CA` and is anchored in the globally accepted root `COMODO RSA Certification Authority`. Some documents were subsequently changed which yielded a document digest mismatch. Results of our tests were compared to the signature verification performed by Acrobat Reader DC 2018 running on Microsoft Windows 10 where the required trust anchors were configured. Some tests were carried out in an environment with the shifted system time (beyond the signer's certificate validity period). The document validation status obtained from the `pdf-sigil` command-line

[7]`https://www.comodo.com/home/email-security/free-email-certificate.php`

utility completely agreed with our expectations and exactly matched with the reference tool in all cases.

It was discovered during the tests that LibreOffice – free open-source software previously declared as capable of full-featured digital signature handling – cannot verify the PKCS#1 type of signature and claims "*This document has an invalid signature*". As described in the section 2.2.1 this type is still valid (although deprecated in PDF 2.0 and with declining usage) and Adobe Reader DC declared exactly the same documents as properly signed.

We explained in the section 1.3.2 that the signer's certificate revocation check is an important part of the whole digital signature validation process. However, this has not been implemented in PDF-Sigil yet. The library is nearly ready to verify the certificate using the CRL once this list is made available. The root CAs usually update their revocation list once a week; consequently, the most appropriate attitude seems to have a separate service responsible for handling the CRLs, including regular downloads of the updates with verifying their integrity and storage. Employing the OCSP or CT log auditing properly is a challenging task of its own and details of handling them differ even among the major web browsers.

Chapter 2 showed there are rich possibilities with digital signatures in PDF documents. The developed library implements validation of the very basic signature type only and should be understood more as a first step than a full-featured tool. The revocation checks mentioned above and handling PKCS#7 signatures should be the next development goals. Complete coverage of the approval signatures and transform methods will require full document parsing which may become quite complex and complicated work. Here, one might re-consider some of the original design decisions. The task might be solved more easily with the help of some document parsing library. Using modern C++ with its standard containers, smart pointers and move semantics might better keep the source code clean, understandable and secure while keeping the simple interface and good performance.

4.5 Security

The security of any application can be evaluated from various perspective – we may call the points of view for example the theoretical grounds, the hardware and software environment and the application security.

All the applications that have some relation to security use some procedures and techniques based on the research in the cryptography, algebra, number theory and other related scientific subjects. Their security, recommended practice in usage, weakness evaluation, etc. are being thoroughly studied by hundreds of professionals all around the world and one can do no better than to listen to their opinion and follow their recommendations. Self-made solutions and amateur "improvements" most usually lead to security by obscurity or serious security flaws. The field is far from being rigid; the evolution is underway all the time. See for example the deprecation of the SHA-1 hash function and finding the first real hash collision not so far ago.

Computer programs “live” in some environment, they are run on some hardware, use some non-volatile storage, depend on the installed operating system, running services, third-party libraries, many a time on network protocols and services as well as routers, switches and other network elements and the like. Taking this part of security seriously means to carry out a proper system and network administration including regular software updates, setting up and keeping strict security policies (including even things like restricted physical access to devices). The hardware vulnerabilities should not be underestimated as well, the recent Spectre/Meltdown affair serves as an excellent example.

Finally, the application itself should be designed to minimize any potential of leaking the sensitive data and to take all available counter-measures against any malicious activity affecting the security. Developers with enough experience, good coding style, continual testing on different levels, independent code reviews or audits are possible remedies that allow reducing eventual security vulnerabilities (and other problems as well). The application should be acquired from reliable sources (either in the form of binaries or source codes), its integrity validated and should be installed appropriately. This includes right configuration, setting up user access rights, updating dependencies, etc. Regular maintenance is a must and thorough monitoring may reveal issues even before they become a problem. However, there is a big difference between an essential information system in a corporate environment and a common application for home users.

From the specific point of view of the PDF-Sigil library one can think of two main potential security issues, the first being possible exposure of the contents of the verified PDF document and the other eventual incorrect evaluation of the digital certificate validity (i.e. letting a valid signature fail or passing an invalid signature as correct).

The PDF-Sigil library is written in the C language which is somewhat prone to security problems like buffer overflows, memory over-reads, segmentation faults, jump/call misplacements, etc. when not employed correctly. The PDF document is stored within an internal buffer allocated on the process heap during its processing. Efforts were exerted to perform all the allocations and disposals properly; the buffer is even completely invalidated (rewritten with garbage) before giving it back to the operating system in order to avoid any chance of leaking data to other processes. The driving application should follow the recommended sequence of steps and must not forget to call the clean-up function in all situations. Special care was taken so that even the maliciously crafted PDF document (e.g. with object offsets in the document cross-reference table leading outside the valid range) would not confuse the signature validation algorithm. Here, the infamous Heartbleed vulnerability in OpenSSL[8] publicly disclosed in 2014 [45] and affecting millions of devices is the lecture to learn from.

Even if we accept that the validation procedure of the digital signature is implemented properly in the library, there are still some external factors influencing the expected result. For certificate manipulations and message digest computation, the library relies on the OpenSSL library which is usually installed system-wide and should be kept up-to-date by the system administrator. The same responsibility concerns setting the current time (usually regularly synchronized over the network with trusted time servers) which is important for checking the certificate expiry. The key element in the PKI are the trusted root anchors, i.e. the set of certificates of these root CAs. The PDF-Sigil library may use the system storage, then again the administrator should maintain it according to local conditions (regular updates of public CAs, custom policies with private CAs). As an alternative, the trusted roots may be provided as files and the local security policies should define the filesystem access rights (users with a right to modify the certificates vs. users with the read-only access).

The functions exported from the library are expected to be called sequentially. There is no built-in support to avoid the race conditions when using the functions concurrently from several threads with the same context. The behaviour in such circumstances is undefined and the risk of security consequences is high. It is the responsibility of the calling application to make sure that proper synchronization takes place in the multi-threaded environment. There are, on the other hand, no hidden static variables in the library and the function calls with different contexts do not mutually interfere (the internally used OpenSSL functions are safe from this point of view). Apparently, the main application is supposed not to modify the context in any way.

[8]The CVE-2014-0160 vulnerability nicknamed “Heartbleed” was a bug in the OpenSSL implementation of the TLS transport protocol which allowed the attacker with a help of the malformed keep-alive packet to access the memory even with very sensitive information (private keys, passwords, etc.) without providing any credentials.

Conclusion

The first goal of this thesis was to study the current theory, recommendations and best practices of digital signatures. The chapter 1 along with the chapter 2 contains the theoretical part and provides an explanation of digital signatures in general, internal structure of PDF documents and all types of digital signatures possible in PDF, as well as applicable transform methods. The security aspects of signature verification are discussed as well.

The second goal of this thesis was to evaluate the current state of the art in digital signatures of PDF documents in the Linux operating systems. This is fulfilled in the chapter 3 containing a summary of PDF viewers running in Linux and their capability to work with digital signatures. The research revealed the majority of open-source applications is unable to handle digital signatures. Another part of this research turned our attention to other tools available and revealed that the libraries and applications with support for the digital signatures are mostly confined to Java. However, if someone cannot afford to run a Java environment for whatever reason, the situation can hardly be considered satisfying.

The next goal was to write a C or C++ library for verification of digital signatures in PDF together with a command line application demonstrating the use of this library. The achievements of the author together with a short documentation are described in the chapter 4. The PDF-Sigil library and the command line application are a part of this thesis.

Due to the fact that verification of digital signatures in PDF is a quite complicated matter, there is still plenty of space to improve this library. This thesis provides a majority of information needed to understand the task as well as the current state of implementation which allows for future enhancements and extensions. The biggest challenge which the library faces is the need to fully comprehend the PDF document structure in order to properly employ transform methods used together with a digital signature.

Finally, the security aspects are thoroughly discussed both generally and specifically for the developed library.

Bibliography

[1] Simner, M. L.; Leedham, C. G.; et al. *Handwriting and Drawing Research: Basic and Applied Issues.* Amsterdam, Netherlands: IOS Press, 1996, ISBN 978-90-5199-280-9.

[2] Hernández-Ardieta, L. *Enhancing the Reliability of Digital Signatures as Non-Repudiation Evidence under a Holistic Threat Model.* Dissertation thesis, University Carlos III of Madrid, Leganés, Feb. 2011. Available from: `https://e-archivo.uc3m.es/bitstream/handle/10016/11882/Tesis_Jorge_Lopez_Hernandez_Ardieta.pdf` [accessed 2018-04-15]

[3] National Institute of Standards and Technology. *Digital Signature Standard (DSS).* NIST FIPS 186-4, Gaithersburg, July 2013, doi:10.6028/NIST.FIPS.186-4. Available from: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf` [accessed 2018-03-25]

[4] Suzuki, K.; Tonien, D.; et al. Birthday Paradox for Multi-Collisions. In *Information Security and Cryptology – ICISC 2006*, Springer, Berlin, Heidelberg, Nov. 2006, ISBN 978-3-540-49112-5 978-3-540-49114-9, pp. 29–40, doi:10.1007/11927587_5. Available from: `https://link.springer.com/chapter/10.1007/11927587_5` [accessed 2018-04-02]

[5] National Institute of Standards and Technology. *Secure Hash Standard.* NIST FIPS 180-4, Gaithersburg, July 2015, doi:10.6028/NIST.FIPS.180-4. Available from: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf` [accessed 2018-03-25]

[6] Preneel, B.; Dobbertin, H.; et al. *The Cryptographic Hash Function RIPEMD-160.* 1997. Available from: `https://www.esat.kuleuven.be/cosic/publications/article-317.pdf` [accessed 2018-04-15]

[7] Stevens, M.; Bursztein, E.; et al. The First Collision for Full SHA-1. In *Advances in Cryptology – CRYPTO 2017*, volume 10401, edited by J. Katz; H. Shacham, Springer International Publishing, 2017, ISBN 978-3-319-63687-0 978-3-319-63688-7, pp. 570–596, doi:10.1007/978-3-319-63688-7_19. Available from: `http://link.springer.com/10.1007/978-3-319-63688-7_19` [accessed 2018-03-25]

[8] Adams, C.; Lloyd, S. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Second Edition, Addison-Wesley Professional, 2003, ISBN 978-0-672-32391-1.

[9] Perl, H.; Fahl, S.; et al. You Won't Be Needing These Any More: On Removing Unused Certificates from Trust Stores. In *Financial Cryptography and Data Security*, edited by N. Christin; R. Safavi-Naini, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ISBN 978-3-662-45472-5, pp. 307–315.

[10] Mozilla Foundation. CA:Symantec Issues. *Wiki Mozilla [online]*, revision 2017-10-16. Available from: `https://wiki.mozilla.org/CA:Symantec_Issues` [accessed 2018-04-23]

[11] Google, Inc. Chrome's Plan to Distrust Symantec Certificates. *Security Google Blog [online]*. Available from: `https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html` [accessed 2018-04-23]

[12] DigiCert, Inc. DigiCert Completes Acquisition of Symantec's Website Security and Related PKI Solutions. *DigiCert [online]*. Available from: `https://www.digicert.com/news/digicert-completes-acquisition-of-symantec-ssl` [accessed 2018-04-29]

[13] Cooper, D.; Santesson, S.; et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [online]*. Number 5280 in Request for Comments, May 2008, doi:10.17487/RFC5280. Available from: `https://www.rfc-editor.org/info/rfc5280` [accessed 2018-04-24]

[14] Santesson, S.; Myers, M.; et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP [online]*. Number 6960 in Request for Comments, June 2013, doi:10.17487/RFC6960. Available from: `https://www.rfc-editor.org/info/rfc6960` [accessed 2018-04-24]

[15] Hallam-Baker, P. *X.509v3 Transport Layer Security (TLS) Feature Extension [online]*. Number 7633 in Request for Comments, Oct. 2015, doi:10.17487/RFC7633. Available from: `https://www.rfc-editor.org/info/rfc7633` [accessed 2018-04-24]

[16] Laurie, B.; Langley, A.; et al. *Certificate Transparency [online]*. Number 6962 in Request for Comments, June 2013, doi:10.17487/RFC6962. Available from: `https://www.rfc-editor.org/info/rfc6962` [accessed 2018-04-24]

[17] International Organization for Standardization. *Document Management — Portable Document Format — Part 2: PDF 2.0*. ISO 32000-2, 2017. Available from: `https://www.iso.org/standard/63534.html` [accessed 2018-03-25]

[18] Jonsson, J.; Kaliski, B. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 [online]*. Number 3447 in Request for Comments, Feb. 2003, doi:10.17487/rfc3447. Available from: `https://www.rfc-editor.org/info/rfc3447` [accessed 2018-04-06]

[19] Housley, R. *Cryptographic Message Syntax (CMS) [online]*. Number 5652 in Request for Comments, Sept. 2009, doi:10.17487/RFC5652. Available from: `https://rfc-editor.org/rfc/rfc5652.txt` [accessed 2018-04-15]

[20] Kroah-Hartman, G.; Corbet, J. *Linux Kernel Development Report [online]*. 2017. Available from: `https://go.pardot.com/l/6342/2017-10-24/3xr3f2/6342/188781/Publication_LinuxKernelReport_2017.pdf` [accessed 2018-05-05]

[21] Freedesktop. 16770 - Support for Digital Signature. *Bugs Freedesktop [online]*. Available from: `https://bugs.freedesktop.org/show_bug.cgi?id=16770` [accessed 2018-04-16]

[22] The Document Foundation. Applying Digital Signatures. *LibreOffice Help [online]*, revision 2016-12-27. Available from: `https://help.libreoffice.org/Common/Applying_Digital_Signatures` [accessed 2018-04-14]

[23] GNOME Github Mirror. Evince: Document Viewer. *GitHub [online]*, revision 2018-04-13. Available from: `https://github.com/GNOME/evince` [accessed 2018-04-15]

[24] Foxit Software, Inc. PDF Solutions for All Your Needs. *Foxit Software [online]*. Available from: `https://www.foxitsoftware.com` [accessed 2018-04-15]

[25] Artifex Software, Inc. Ghostscript. *Ghostscript [online]*. Available from: `https://www.ghostscript.com` [accessed 2018-04-15]

[26] K Desktop Environment Inc. Society. KPDF - More than a Reader. *KDE [online]*, version 0.5.10. Available from: `https://kpdf.kde.org` [accessed 2018-04-15]

[27] Code Industry Ltd. Master PDF Editor 4. *Code Industry [online]*, version 4. Available from: `https://code-industry.net/masterpdfeditor` [accessed 2018-04-15]

[28] Tom Phelps. Multivalent. *Multivalent [online]*, revision 2009-10-28. Available from: `http://multivalent.sourceforge.net` [accessed 2018-04-15]

[29] KDE Github Mirror. Okular: KDE Document Viewer. *GitHub [online]*, revision 2018-04-13. Available from: `https://github.com/KDE/okular` [accessed 2018-04-15]

[30] Mozilla Foundation, Github Repository. PDF.js: PDF Reader in JavaScript. *GitHub [online]*, revision 2018-04-12. Available from: `https://github.com/mozilla/pdf.js` [accessed 2018-04-15]

[31] Qoppa Software. PDF Studio Pro. *Qoppa [online]*. Available from: `https://www.qoppa.com/pdfstudioviewer` [accessed 2018-04-16]

[32] Canonical Ltd. Qpdfview. *Launchpad [online]*, version 0.4.17beta1. Available from: `https://launchpad.net/qpdfview` [accessed 2018-04-16]

[33] Glyph, Cog, LLC. XpdfReader. *XpdfReader [online]*, version 4.0. Available from: `http://www.xpdfreader.com` [accessed 2018-04-16]

[34] Lipp, M.; Ramacher, S. Zathura - A Document Viewer. *Pwmt [online]*, version 0.3.9. Available from: `https://pwmt.org/projects/zathura` [accessed 2018-04-16]

[35] CAcert Inc. CAcert Software PdfSigner. *CACert SVN [online]*, revision 2699. Available from: `http://svn.cacert.org/CAcert/Software/PdfSigner` [accessed 2018-04-16]

[36] DigiSigner. Electronic Signature Service for Your Business. *DigiSigner [online]*. Available from: `https://www.digisigner.com` [accessed 2018-04-16]

[37] iText Group NV. Easy PDF Generation for Java or .NET Developers. *iText [online]*. Available from: `https://itextpdf.com` [accessed 2018-04-16]

[38] Stotz, J. P. jPdfSign. *Fraunhofer [online]*, version 0.3. Available from: `https://private.sit.fraunhofer.de/~stotz/software/jpdfsign` [accessed 2018-04-16]

[39] Cacek, J. JSignPdf. *SourceForge [online]*, version 1.6.2. Available from: `http://jsignpdf.sourceforge.net` [accessed 2018-04-16]

[40] Iacono, A.; Capizzi, S.; et al. Open Signature. *SourceForge [online]*. Available from: `http://opensignature.sourceforge.net/english.php` [accessed 2018-04-16]

[41] The Apache Software Foundation. PDFBox - A Java PDF Library. *Apache [online]*, version 2.0.9. Available from: `https://pdfbox.apache.org` [accessed 2018-04-16]

[42] Freedesktop. Poppler. *Freedesktop [online]*, version 0.63.0. Available from: `https://poppler.freedesktop.org` [accessed 2018-04-16]

[43] Pfläging, P. PortableSigner. *SourceForge [online]*, version 2.0. Available from: `http://portablesigner.sourceforge.net` [accessed 2018-04-16]

[44] Secured Signing. Free Verification Service. *Secured Signing [online]*. Available from: `https://www.securedsigning.com/products/signature-verification-service` [accessed 2018-04-30]

[45] Synopsys, Inc. The Heartbleed Bug. *Heartbleed [online]*. Available from: `http://heartbleed.com` [accessed 2018-05-03]

Appendix A

Acronyms

BR	Baseline Requirements
CA	Certification Authority
CAdES	CMS Advanced Electronic Signatures
CMS	Cryptographic Message Syntax
CRL	Certificate Revocation List
CT	Certificate Transparency
DSS	Document Security Store
DTS	Document Timestamp
GUI	Graphical User Interface
MitM	Man in the Middle
MDP	Modification Detection and Prevention
OCSP	Online Certificate Status Protocol
PAdES	PDF Advanced Electronic Signatures
PDF	Portable Document Format
PKCS	Public Key Cryptographic Standards
PKI	Public Key Infrastructure
RA	Registration Authority
UR	Usage Rights
VRI	Validation-Related Information

Program output

1. Valid signature with the store of trusted certificates provided by the operating system (`-ts`)

```
$ ./pdf-sigil -f test/subtype_adbe.x509.rsa_sha1.pdf -ts

  ____  ____  _____     ____  _       _ _
 |  _ \|  _ \|  ___|   / ___|(_) __ _(_) |
 | |_) | | | | |_ _____\___ \| |/ _` | | |
 |  __/| |_| |  _|_____|___) | | (_| | | |
 |_|   |____/|_|       |____/|_|\__, |_|_|
                                |___/

 =========================================

 VERIFICATION SUCCESSFUL

     subfilter:          adbe.x509.rsa_sha1 (PKCS#1)
     hash function:      SHA-1

     DATA INTEGRITY
     --------------
     original digest:    d5 8e 3b cd 7a 82 43 a9 51 2a a4 68 48 4a f1 e3 ac 50 0f 3a
     computed digest:    d5 8e 3b cd 7a 82 43 a9 51 2a a4 68 48 4a f1 e3 ac 50 0f 3a
     digest match:       YES

     CERTIFICATE
     -----------
     verified:           YES
```

2. Same case as *1*, however no trusted certificates were set up

```
$ ./pdf-sigil -f test/subtype_adbe.x509.rsa_sha1.pdf

  ____  ____  _____     ____  _       _ _
 |  _ \|  _ \|  ___|   / ___|(_) __ _(_) |
 | |_) | | | | |_ _____\___ \| |/ _` | | |
 |  __/| |_| |  _|_____|___) | | (_| | | |
 |_|   |____/|_|       |____/|_|\__, |_|_|
                                |___/

 ========================================

 VERIFICATION FAILED

     subfilter:          adbe.x509.rsa_sha1 (PKCS#1)
     hash function:      SHA-1

     DATA INTEGRITY
     --------------
     original digest:    d5 8e 3b cd 7a 82 43 a9 51 2a a4 68 48 4a f1 e3 ac 50 0f 3a
     computed digest:    d5 8e 3b cd 7a 82 43 a9 51 2a a4 68 48 4a f1 e3 ac 50 0f 3a
     digest match:       YES

     CERTIFICATE
     -----------
     verified:           NO
```

3. Same as *1* with the difference that the very last byte was changed from `0a` to `20` (hexadecimal).

```
$ ./pdf-sigil -f test/modified_pkcs1.pdf -ts

  ____  ____  _____     ____  _       _ _
 |  _ \|  _ \|  ___|   / ___|(_) __ _(_) |
 | |_) | | | | |_ _____\___ \| |/ _` | | |
 |  __/| |_| |  _|_____|___) | | (_| | | |
 |_|   |____/|_|       |____/|_|\__, |_|_|
                                |___/

 ========================================

 VERIFICATION FAILED

     subfilter:          adbe.x509.rsa_sha1 (PKCS#1)
     hash function:      SHA-1

     DATA INTEGRITY
     --------------
     original digest:    d5 8e 3b cd 7a 82 43 a9 51 2a a4 68 48 4a f1 e3 ac 50 0f 3a
     computed digest:    be bc fe ef 5b 19 ad 1a 43 6b 61 ee 34 ab a4 ef 88 53 9d 97
     digest match:       NO

     CERTIFICATE
     -----------
     verified:           YES
```

Appendix C

Contents of the enclosed CD

The enclosed CD contains full source codes of the PDF-Sigil library and the command-line tool `pdf-sigil`, complete text of the thesis both in its LaTeX sources and the resulting PDF as well as files for demonstration purposes. These are PDF documents with a valid or invalid digital signature and, for ease of use, also the compiled binaries for Linux x64 OS. The standard GNU C library and OpenSSL with all its dependencies are required in order to run `pdf-sigil` (the compilation was carried out on Fedora 28).

```
/........................................................CD root
├── readme.txt ......................................CD contents
├── BP_Stefan_Tomas_2018.pdf ......................thesis in PDF
├── bin ..............................................executables
│   ├── libpdfsigil.so ....................PDF-Sigil shared library
│   ├── pdf-sigil ................................command-line tool
│   ├── selftest ...............................libpdfsigil selftest
│   └── test .....................directory with testing PDF files
└── src ..................................................sources
    ├── pdf-sigil ........................implementation sources
    └── thesis ...........................LaTeX sources of this thesis
```