1 Trillion Dollar Refund – How To Spoof PDF Signatures

Vladislav Mladenov*
Vladislav.Mladenov@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Christian Mainka*
Christian.Mainka@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Karsten Meyer zu Selhausen
Karsten.MeyerzuSelhausen@hackmanit.de
Hackmanit GmbH

Martin Grothe
Martin.Grothe@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Jörg Schwenk
Joerg.Schwenk@rub.de
Ruhr University Bochum, Chair for Network and Data Security

ABSTRACT

The Portable Document Format (PDF) is the de-facto standard for document exchange worldwide. To guarantee the authenticity and integrity of documents, digital signatures are used. Several public and private services ranging from governments, public enterprises, banks, and payment services rely on the security of PDF signatures.

In this paper, we present the first comprehensive security evaluation on digital signatures in PDFs. We introduce three novel attack classes which bypass the cryptographic protection of digitally signed PDF files allowing an attacker to spoof the content of a signed PDF. We analyzed 22 different PDF viewers and found 21 of them to be vulnerable, including prominent and widely used applications such as Adobe Reader DC and Foxit. We additionally evaluated eight online validation services and found six to be vulnerable. A possible explanation for these results could be the absence of a standard algorithm to verify PDF signatures – each client verifies signatures differently, and attacks can be tailored to these differences. We, therefore, propose the standardization of a secure verification algorithm, which we describe in this paper.

All findings have been responsibly disclosed, and the affected vendors were supported during fixing the issues. As a result, three generic CVEs for each attack class were issued [50–52]. Our research on PDF signatures and more information is also online available at https://www.pdf-insecurity.org/.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

PDF, signature

ACM Reference Format:
Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 2019. *1 Trillion Dollar Refund* – How To Spoof PDF Signatures. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3339812

*Both authors equally contributed to this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3339812

1 INTRODUCTION

Introduced in 1993 by Adobe Systems, the Portable Document Format (PDF) was designed as a solution for the consistent presentation of documents, independent of the operating system and hardware. Today, the PDF format has become the standard for electronic documents in our daily workflow. The total number of PDF files in the world is hard to guess, but according to Adobe System's Vice President of Engineering, *Phil Ydens*, there were about 1.6 billion PDF files on the web in 2015 [3], whereby 80% were created in the same year. This leads him to estimate that about 2.5 trillion PDF files were created since 2015. Whether this is correct or not, PDF files are heavily used in everyone's life – for exchanging information, for creating and archiving invoices and contracts, for submitting scientific papers, or for collaborating and reviewing texts.

PDF Digital Signatures. The PDF specification supports digital signatures since 1999 in order to guarantee that the document was created or approved by a specific person and that it was not altered afterward. PDF digital signatures are based on asymmetric cryptography whereby the signer possess a public and private key pair. The signer uses his private key to create the digital signature. Any document modification afterward invalidates the signature and leads to an error message thrown by the corresponding PDF viewer or validation service. PDF *digital signatures* must not be confused with *electronic signatures*, which are the electronic equivalent of handwritten signatures; this is done by basically adding an image of the signer's handwritten signature into the document. Electronic signatures do not provide any cryptographic protection so that spoofing attacks are trivial and not further considered.

In 2000, President Bill Clinton enacted a federal law facilitating the use of electronic and digital signatures in interstate and foreign commerce by ensuring the validity and legal effect of contracts. He even approved the eSign Act by digitally signing it [35]. Since 2014, organizations delivering public digital services in an EU member state are required to support digitally signed documents, which are even admissible as evidence in legal proceedings [48]. In Austria, every governmental authority digitally signs any document [17, §19]. In addition, any new law is legally valid after its announcement within a digitally signed PDF. Several countries like Brazil, Canada, the Russian Federation, and Japan also use and accept

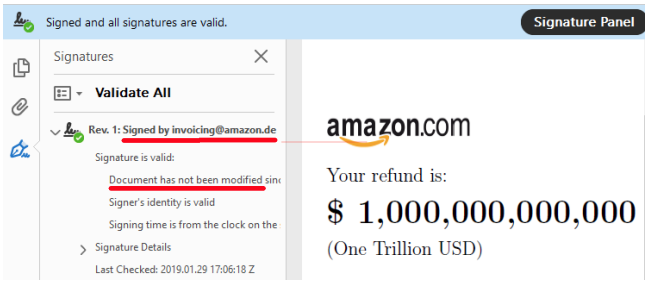


Figure 1: Validly signed PDF document by Amazon with a spoofed content. Adobe Acrobat DC claims that the 'document has not been modified since the signature was applied'.

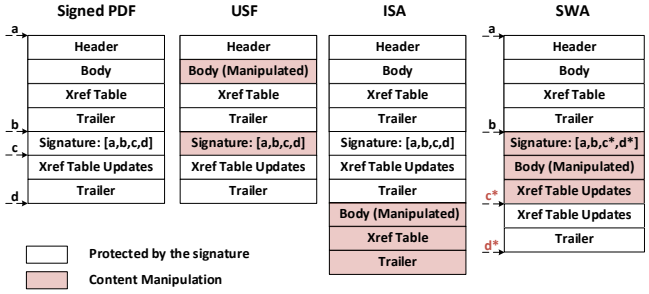


Figure 2: An overview of the attacks introduced in this paper: USF, ISA, and SWA. Each attack relies on a different injection point for malicious content without invalidating the digital signature.

digitally signed documents [53]. Outside governmental services digitally signed PDFs are used by the private sector to sign invoices and contracts: e.g., invoices by Amazon, Decathlon, Sixt, and even more are concluded secretly between companies. Even in the academic world, PDF signatures are used to sign scientific papers (e.g., ESORICS proceedings) as evidence of the paper's submission state. According to Adobe Sign, the company processed 8 billion electronic and digital signatures in 2017 alone [1].

We thus raise the question: *Is it possible to spoof a digitally signed PDF document in a way such that the spoofed document is indistinguishable from a valid one?.*

Novel Attacks on PDF Signatures. In this paper, we show how to spoof a digitally signed PDF document. The only requirement of our attacks is access to a signed PDF (e.g., an Amazon invoice). Given such a PDF, our attacks allow an attacker to change the PDF's content arbitrarily without invalidating its signature – see Figure 1. Plausible attack scenarios which may abuse the vulnerabilities could include, for example, the manipulation of the billing date on the digitally signed receipt to extend the warranty period of a product or changing the contract's information to attain more resources than agreed upon.

We systematically analyze the verification process of PDF signatures in different desktop applications as well as in server implementations, and we introduce three novel attack classes, see Figure 2. Each of them gives a blueprint for an attacker to modify a validly signed PDF file in such way that for the targeted viewer, the displayed content is altered without being detected by the viewer's signature verification code – all elements in the GUI related to signature verification are *identical* to the original, unaltered document.

On a technical level, each attack class abuses a different step in the signature validation logic.

(1) The **Universal Signature Forgery (USF)** manipulates meta information in the signature in such a way that the targeted viewer application opens the PDF file, finds the signature, but is unable to find all necessary data for its validation. Instead of treating the missing information as an error, it shows that the contained signature is *valid*.
(2) The **Incremental Saving Attack (ISA)** abuses a legitimate feature of the PDF specification, which allows *updating* a PDF file by appending the changes. The feature is used, for example, to store PDF annotations, or to add new pages while editing the file. The main idea of the ISA is to use the same technique for changing elements, such as texts, or whole pages included in the signed PDF file to what the attacker desires. The PDF specification does not forbid this, but the signature validation should indicate that the document has been altered after signing. We introduce four variants of ISA masking the modification made without raising any warnings that the document was manipulated.
(3) The **Signature Wrapping Attack (SWA)** targets the signature validation logic by relocating the originally signed content to a different position within the document and inserting new content at the allocated position. We introduce three different variants of SWA which we used to bypass the signature validation.

Large-Scale Evaluation. We provide the first large-scale evaluation covering 22 different PDF viewers installed on Windows, Linux, or MacOS. We systematically analyzed the security of the signature validation on each of them and found signature bypasses in 21 of 22 of the viewers, including Adobe Reader DC and Foxit. Additionally, we analyzed eight online validation services supporting signature verification of signed PDF files. We found six of them to be vulnerable against at least one of the attacks, and included, among others, DocuSign – one of the worldwide leading cloud services providing electronic signatures and ranked #4 on the Forbes Cloud 100 [15]. The results are reasoned by the fact that:

(1) There is almost no related work regarding the security of digitally signed PDF files, even though integrity protection is part of the PDF specification since 1999.
(2) The PDF specification does not provide an implementation guideline or a best-practices document regarding the signature validation. Thus, developers implement a security critical component without having a thorough understanding regarding the actual risks.

Contributions. The contributions of this paper are:

- We developed three novel attack classes on PDF signatures. Each class targets a different step in the signature validation process and enables an attacker to bypass a PDF's integrity protection completely, shown in section 4.
- We provide the first in-depth security analysis of PDF applications. The results are alarming: out of 22 popular desktop

viewers, we could bypass the signature validation in 21 cases, as seen subsection 5.1.
- We additionally analyzed eight online validation services used within the European Union and worldwide for validating signed documents. We could bypass the signature validation in six cases, shown in subsection 5.2.
- Based on our experiences, we developed a secure signature validation algorithm and communicated it with the application vendors during the responsible disclosure process, as seen in section 6.
- By providing the first in-depth analysis of PDF digital signatures, we pave the road for future research. We reveal new insights and show novel research aspects regarding PDF security, shown in section 8.

Responsible Disclosure. In cooperation with the BSI-CERT, we contacted all vendors, provided proof-of-concept exploits, and helped them to fix the issues. As a result, the following three generic CVEs for each attack class covering all affected vendors were issued [50–52].

2 PDF BASICS

This section deals with the foundations of the Portable Document Format (PDF). We give an overview of the file structure and explain how the PDF standard for signatures is implemented.

2.1 Portable Document Format (PDF)

A PDF consists of 4 parts: *header*, *body*, *xref table*, and a *trailer*, as depicted in Figure 3.

Header. The *header* is the first line within a PDF and defines the interpreter version to be used. The provided example uses version `PDF 1.7`.

Body. The *body* specifies the content of the PDF and contains text blocks, fonts, images, and metadata regarding the file itself. The main building blocks within the body are *objects*, which have the following structure: Each object starts with an object number followed by a generation number. The generation number should be incremented if additional changes are made to the object.

```
1 0 obj
  ↑  ↑  ↑
  |  |  └── Constant String "obj"
  |  └───── Generation Number
  └──────── Object Number
...
endobj
```

Listing 1: Example of an object declaration within the body.

In the example depicted in Figure 3, the *body* contains four objects: *Catalog*, *Pages*, *Page*, and *stream*. The *Catalog* object is the root object of the PDF file. It defines the document structure and can additionally declare access permissions. The *Catalog* refers to a *Pages* object which defines the number of the pages and a reference to each *Page* object (e.g., text columns). The *Page* object contains information on how to build a single page. In the given example, it only contains a single string object "Hello World!".

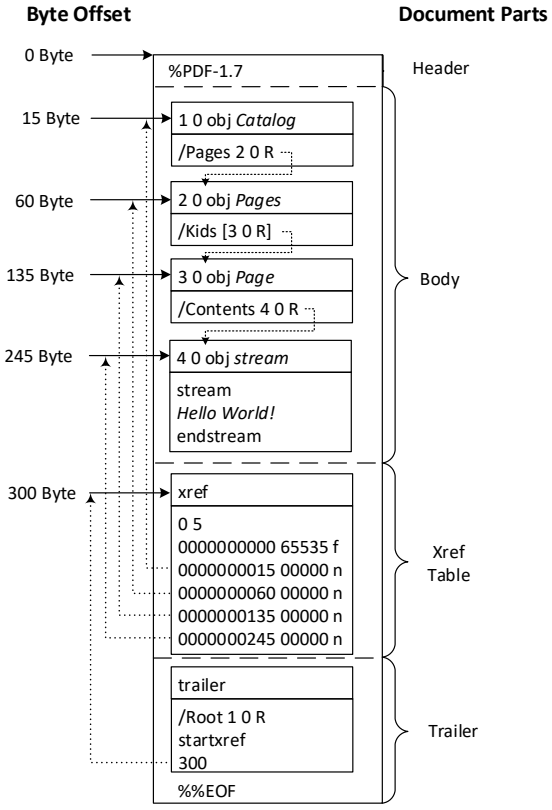


Figure 3: A simplified example of a PDF file's internal structure. We depict the object names after the *obj* string for clarification.

Xref table. The *Xref table* contains information about all PDF objects. An *Xref table* can contain one or more sections.

- Each *Xref table* section starts with a line consisting of two integer entries a b (e.g., "0 5" as shown in Figure 3) which indicates that in the *Xref table* the following $b = 5$ lines describe objects with ID (also known as object numbers) ranging from $a \in \{0, \ldots, b-1\} = \{0, \ldots, 4\}$
- Each object entry ($a \in \{0, \ldots, b-1\}$) in the *Xref table* has three entries x y z, where x defines the byte offset of the object from the beginning of the document; y defines its generation number, and $z \in \{'n', 'f'\}$ describes whether the object is in-use ($'n'$) or not ($'f'$, say "free"). For example, the line "0000000060 00000 n" is the third line after "0 5" and, thus, describes the in-use object with object number 2 and generation number 0 at byte offset 60 (see "2 0 obj" in Figure 3).

Trailer. After a PDF file is read into memory, it is processed from the end to the beginning. Thus, the *Trailer* is the first processed content of a PDF file. It contains references to the *Catalog* (`1 0 R`) and the *Xref table*.

2.2 Creating a PDF Signatures

This section explains how a digitally signed PDF file is built.

Incremental Saving. PDF Signatures rely on a feature of PDF called *incremental saving* (also known as incremental updates), allowing the modification of a PDF file without changing the previous content.

In Figure 4, an original document (shown on the left side) is being modified via incremental saving by attaching a new *body*, as well as a new *Xref table*, and a new *Trailer* at the end of the file. Within the *body*, new objects can be defined. A new *Pages* object

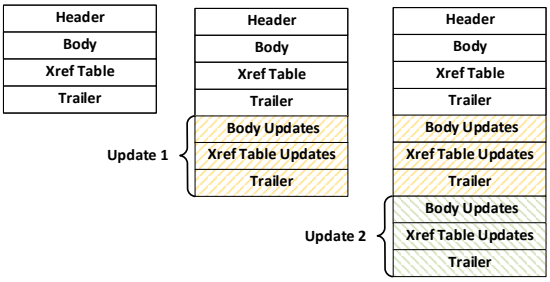


Figure 4: Multiple incremental savings applied on a PDF file.

can be defined, referencing two pages, for example, `/Kids [3 0 R 3 0 R]`. For reasons of simplicity, the same content (`3 0 R`) was used twice here. The *Xref table* contains only a description of the newly defined objects. The new *Trailer* contains a reference to the *Catalog* (it could be the old *Catalog* or an updated one), the byte offset of the new *Xref table*, and the byte offset of the previously used *Xref table*. This scheme is applied for each incremental saving.

Structure of a Signed PDF. The creation of a digital signature on a PDF file relies on incremental saving by extending the original document with objects containing the signature information.

In Figure 5, an example of a signed PDF file is shown. The original document is the same document as depicted in Figure 3. By signing the document, an incremental saving is applied and the following content is added: a new *Catalog*, a *Signature* object, a new *Xref table* referencing the new object(s), and a new *Trailer*. The new *Catalog* extends the old one by adding a new parameter *Perms*, which defines the restrictions for changes within the document. The *Perms* parameter references to the *Signature* object.

The *Signature* object (`5 0 obj`) contains information regarding the applied cryptographic algorithms for hashing and signing the document. It additionally includes a `Contents` parameter containing a hex-encoded PKCS7 blob, which holds the certificates as well as the signature value created with the private key that corresponds to the public key stored in the certificate. The `ByteRange` parameter defines which bytes of the PDF file are used as the hash input for the signature calculation and defines two integer tuples:

(a, b) : Beginning at byte offset `a`, the following `b` bytes are used as the first input for the hash calculation. Typically, a=0 is used to indicate that the beginning of the file is used while a+b is the byte offset where the PKCS#7 blob begins.

(c, d) : Typically, byte offset c is the end of the PKCS#7 blob, while c+d points to the last byte range of the PDF file and is used as the second input to the hash calculation.

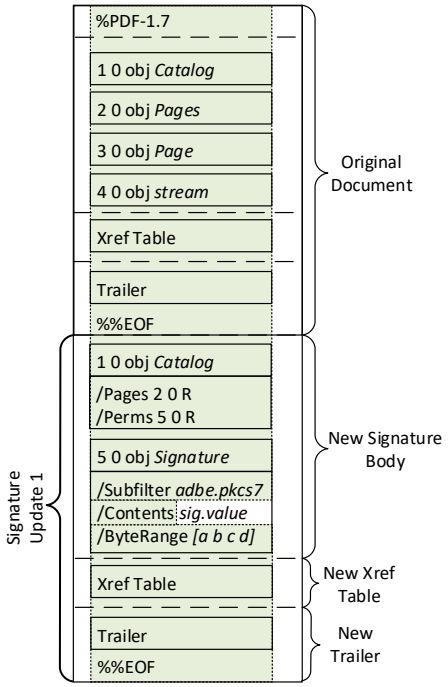


Figure 5: A simplified overview of a signed PDF file.

According to the specification, it is *recommended* to sign the whole file except for the PKCS#7 blob (located in the range between a+b and c) [21].

2.3 Verifying a signed PDF File

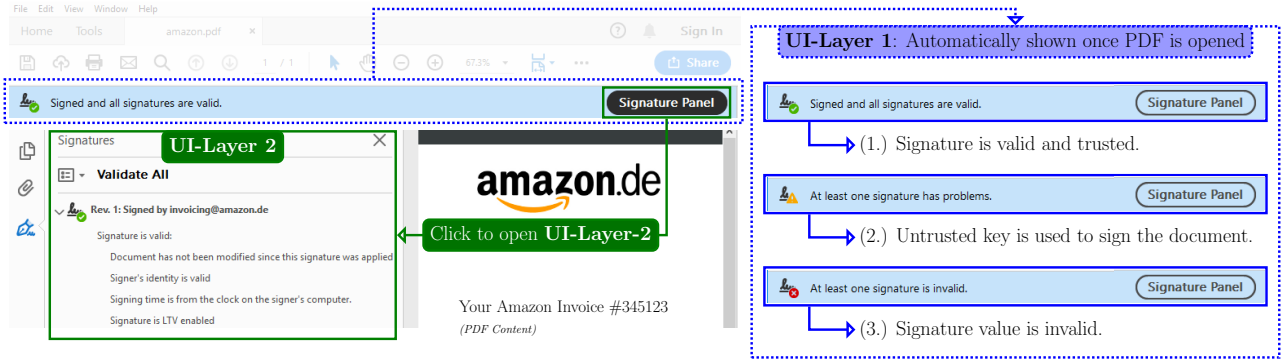
If a signed PDF file is opened with a desktop application that supports signatures, it immediately starts to verify it by: (1) extracting the signature from the PDF and applying the cryptographic operations to verify its correctness and (2) verifying if the used signing keys are trusted, e.g., an x.509 certificate. One thing that all applications had in common is that by default, they do not trust the operating system's keystore. Similar to web browsers such as Firefox, they distribute their own keystore and keep the list of trusted certificates up to date. Additionally, every viewer allows the utilization of a different keystore containing trusted certificates. This feature is interesting for companies using their own Certificate Authority (CA) and disallowing the usage of any other CA. As a result, similar to key pinning, the viewer can be configured to trust only specific certificates.

3 ATTACKER MODEL

In this section, we describe the attacker model including the attackers' capabilities and the winning conditions.

Victim. A victim can be either a human who opens the file using a certain PDF desktop application or a website offering an online validation service.

Attacker Capabilities. It is assumed that the attacker is in possession of a signed PDF file. The attacker does not possess the proper private key that was used to sign it. Also, we assume that the victim



(a) A screenshot of Adobe Acrobat DC is depicted after opening a signed PDF document. A signature validation bar (*UI-Layer 1*) is automatically shown. A signature panel (*UI-Layer 2*) can be opened by pressing the corresponding button. The panel provides more details, e.g., the error message or email address of the signer.

(b) There are 3 validation states: (1.) A *green* icon indicates a *valid and trusted* signature. (2.) If the icon appears in *yellow*, the key used to sign the PDF is untrusted, e.g., because a self-generated certificate is used. (3.) The *red* icon indicates an *invalid* signature, e.g., if the PDF file is modified.

Figure 6: PDF signature validation with two UI-Layers.

only trusts specific certificates (e.g., via the trust store) and the attacker does not possess a single private key that is trusted by the victim. Thus, malicious PDF files which are digitally signed by the attacker with a self-generated or untrusted certificate will be not verified successfully by the viewer. Apart from this restriction, the attacker can arbitrarily modify the PDF file, for example, by changing the displayed content.

The attacker finally sends the modified PDF file to a victim, where the file is then processed.

Winning Conditions. For the successful execution of this attack, we have defined two conditions:

Cond. 1) When opening the PDF file, the target application, i.e., the viewer or online service, shows a UI displaying that it is validly signed and is identical to the originally unmodified signed PDF file.

Cond. 2) The viewer application displays content which is different from the original file.

For viewer applications, both winning conditions must be met. For the online validation services, only the first condition must be fulfilled because online services do not show the content of a PDF file. Instead, they generate a report containing the results of the verification, see Figure 11. Therein, the services show whether the PDF file is validly signed.

Desktop viewer applications differ substantially in displaying the results of the signature verification. To classify if an attack is successful and to determine if the victim could detect the attack, we defined two different UI-Layer:

- *UI-Layer 1* represents the UI information regarding the signature validation which is immediately displayed to the user after opening the PDF file. It is shown without any user interaction. Examples for Adobe Acrobat DC UI-Layer 1 are presented in the top part of the purple box in Figure 6.
- *UI-Layer 2* provides extended information regarding the signature validation. It can be accessed by clicking on the respective menu option. Examples for Adobe Acrobat DC UI-Layer 2 are displayed in the bottom-left part of the green box in Figure 6.

If the information presented on the *UI-Layer 2* states that the signature is invalid or the document has been modified after the application of the signature, the attack can still be classified as successful for *UI-Layer 1*.

In Figure 6, an example of a successful signature validation on *UI-Layer 1* and *UI-Layer 2* is presented. After opening the PDF file, the information *Signed and all signatures are valid* is displayed. Further information is revealed by clicking on the Signature Panel and can be seen in the green box of UI-Layer 2.

Self-Signed PDFs. We do not consider self-signed PDF as a legitimate attack and neither use nor rely on them because a self-signed PDF can clearly be distinguished from a PDF signed with a trusted certificate; cf. green and yellow icon in Figure 6.

4 HOW TO BREAK PDF SIGNATURES

In this section, we present three novel attack classes on PDF signatures: Universal Signature Forgery (USF), Incremental Saving Attack (ISA), and Signature Wrapping Attack (SWA). All attack classes bypass the PDF's signature integrity protection, allowing the modification of the content arbitrarily without the victim noticing. The attacker's goal is to place *malicious content* into the protected PDF file, such that the previously defined winning conditions for viewer applications and online validation services are satisfied.

During the security analysis, we designed many broken PDF files for each attack class which are clearly violating the PDF specification in order to bypass the signature verification process.

We also learned that nearly every PDF viewer has a high level of error-tolerance so that these PDF files could be successfully opened even if required parameters are missing. We can only assume that

this is due to the individual interpretation of the PDF specification by each vendor.

4.1 Universal Signature Forgery (USF)

The main idea of Universal Signature Forgery (USF) is to disable the signature verification while the application viewer still shows a successful validation on the UI layer. This attack class was inspired by existing attacks applied to other message formats like XML [42] and JSON [33]. Such attacks either remove all signatures or use insecure algorithms like none in JSON signatures. For PDFs we estimated two possible approaches – either to remove information within the signature which makes the validation impossible, or to remove references to the signature to avoid the validation. Removing references did not lead to any successful attack. Thus, we concentrated on manipulations within the signature. In this case, the attacker manipulates the signature object in the PDF file, trying to create an invalid entry within this object. Although the signature object is provided, the validation logic is not able to apply the correct cryptographic operations. This leads to the situation that a viewer shows some signature information even though the verification is being skipped. In the end, we define 24 different attack vectors, eight of them are depicted in Figure 7.

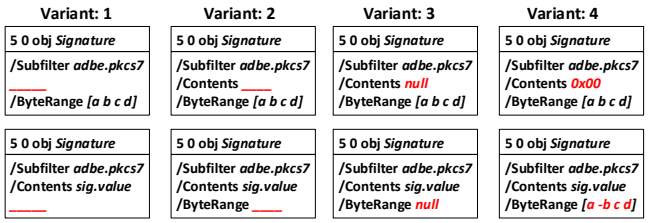


Figure 7: Different USF attack variants manipulating the signature object entries to bypass the signature validation.

In the given example, the attack vectors target two values: a) the entry `Contents` contains the key material as well as the signature value and b) the entry `ByteRange` defines the signed content in the file. The manipulation of these entries is reasoned by the fact that we either remove the signature value or the information stating which content is signed. In **Variant 1**, as depicted in Figure 7, either `Contents` or `ByteRange` are removed from the signature object. Another possibility is defined in **Variant 2** by removing only the content of the entries. In **Variants 3** and **4**, invalid values were specified and tested. Such values are for instance `null`, a zero byte (`0x00`), and invalid `ByteRange` values like negative or overlapping byte ranges. Providing such tests is common for penetration testers since many implementations behave abnormally when processing these special byte sequences.

4.2 Incremental Saving Attack (ISA)

This class of attack relies on the *incremental saving* feature. The idea of the attack is to make an incremental saving on the document by redefining the document's structure and content using the *Body Updates* part. The digital signature within the PDF file protects precisely the part of the file defined in the `ByteRange`. Since the incremental saving appends the *Body Updates* to the end of the file, it is not part of the defined `ByteRange` and thus not part of the signature's integrity protection. To summarize, the signature remains valid, although the *Body Updates* changed the displayed content.

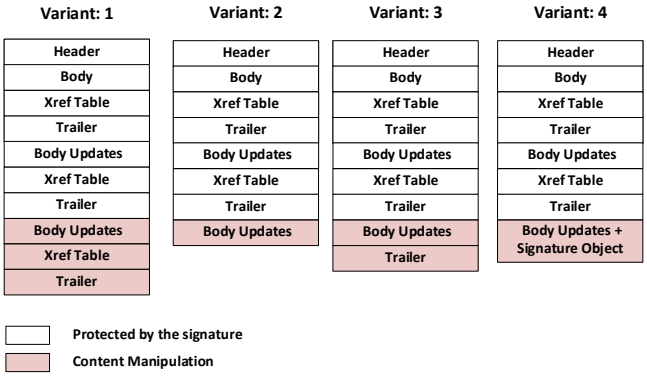


Figure 8: Bypassing the signature protection by using incremental saving. In (1), the main idea of the attack is depicted, while (2)-(4) are variants to obfuscate the manipulations and prevent a viewer to display warnings.

During our research, we elaborated four variants of ISA. These variants are reasoned by the fact that some vendors recognized that incremental saving is dangerous when concerning PDF signatures. These vendors implemented countermeasures to detect changes after the document's signing. As part of our black-box analysis, we were able to determine these countermeasures and find generic bypasses that worked for multiple viewers which we describe below.

Variant 1: ISA with *Xref table* and *Trailer*. For Variant 1 of the ISA class, as depicted in Figure 8, only two of the evaluated signature validators were susceptible to the attack. This is not very surprising since this type of modification is exactly what a legitimate PDF application would do when editing or updating a PDF file. A digital signature in PDF is designed to protect against this behavior; the signature validator recognizes that the document was updated after signing it and shows a warning respectively. To bypass this detection, we found two possibilities. (1) We included an empty *Xref table*. This can be interpreted as a sign that no objects are changed by the last incremental saving. Nevertheless, the included updates are processed and displayed by the viewer. (2) We used an *Xref table* that contains entries for all manipulated objects. We additionally added one entry which has an incorrect reference (i.e., byte offset) pointing to the transform parameters dictionary, which is part of the signature object. The result of these manipulations is that the viewer application does not detect the last incremental saving. No warning is shown that the document has been modified after signing it but the PDF viewer displays the new objects.

Variant 2: ISA without *Xref table* and *Trailer*. Some of the viewers detected the manipulation by checking if a new *Xref table* and *Trailer* were defined within the new incremental saving. By removing the *Xref table* and the *Trailer*, a vulnerable validator does not recognize that incremental saving has been applied and successfully verifies the signature without showing a warning. The PDF

file is still processed normally by displaying the modified document structure. The cause of this behavior is that many of the viewers are error tolerant. In the given case, the viewer completes the missing *Xref table* and *Trailer* and processes the manipulated *body*.

Variant 3: ISA with a *Trailer*. Some of the PDF viewers do not open the PDF file if a *Trailer* is missing. This led to the creation of this attack vector containing a manipulated *Trailer* at the end of the file. To our surprise, the *Trailer* does not need to point to an *Xref table* but rather to any other byte offset within the file. Otherwise, the verification logic detects the document manipulation.

Variant 4: ISA with a copied signature and without a *Xref table* and *Trailer*. The previous manipulation technique was improved by copying the *Signature* object within the last incremental saving. This improvement was forced by some validators which require any incremental saving to contain a signature object if the original document was signed. Otherwise, they showed a warning that the document was modified after the signing.

By copying the original *Signature* object into the latest incremental saving, this requirement is fulfilled. The copied *Signature* object, however, covers the old document instead of the updated part. To summarize, a vulnerable validator does not verify whether each incremental saving is signed, but only if it contains a signature object. Such verification logic is susceptible to ISA.

4.3 Signature Wrapping Attack (SWA)

The Signature Wrapping Attack (SWA) introduces a novel technique to bypass signature protection without using incremental saving. During our research, we observed that the part of the document containing the signature value is excluded from the signature computation and thus it is not integrity protected. The `ByteRange` defines the exact size of this unprotected space. Consequentially, we focused on manipulations on the `ByteRange` entries to increase the size of the unprotected space and allowing the injection of malicious content.

The main idea is to move the signed part of the PDF to the end of the document while reusing the `xref` pointer within the signed *Trailer* to an attacker manipulated *Xref table*. To avoid any processing of the relocated part, it can be optionally wrapped by using a stream object or a dictionary. We distinguish two variants of SWA.

Variant 1: Relocating the second hashed part. Each `ByteRange` entry of the `Signature` object defines two hashed parts of the document. The first variant of the attack relocates only the second hashed part. In Figure 9, two documents are depicted. On the left side, a validly signed PDF file is depicted. The first hashed part begins at byte offset *a* and ends at offset *a+b*, the second hashed part ranges from offset *c* until *c+d*. On the right side, a manipulated PDF file is generated by using SWA as follows:

Step 1 (optional): The attacker deletes the padded zero bytes within the `Contents` parameter to increase the available space for injecting manipulated objects.[1]

Step 2: The attacker defines a new `/ByteRange [a b c* d]` by manipulating the `c` value, which now points to the second signed part placed on a different position within the document.

Step 3: The attacker creates a new *Xref table* pointing to the new objects. It is essential that the byte offset of the newly inserted *Xref table* has the same byte offset as the previous *Xref table*. The position is not changeable since it is referenced by the signed *Trailer*. For this purpose, the attacker can add a padding block (e.g., using whitespaces) before the new *Xref table* to fill the unused space.

Step 4: The attacker injects malicious objects which are not protected by the signature. There are different injection points for these objects. They can be placed before or after the malicious *Xref table*. If Step 1 is not executed, it is only possible to place them *after* the malicious *Xref table*.

Step 5 (optional): Some PDF viewers need a *Trailer* after the manipulated *Xref table*, otherwise they cannot open the PDF file or detect the manipulation and display a warning message. Copying the last *Trailer* is sufficient to bypass this limitation.

Step 6: The attacker moves the signed content defined by `c` and `d` at byte offset `c*`. Optionally, the moved content can be encapsulated within a stream object.

Noteworthy is the fact that the manipulated PDF file does not end with `%%EOF` after the `endstream`. This was necessary due to the reason that some validators throw a warning that the file was manipulated after signing due to an `%%EOF` after the end of signed document (byte offset of EOF > c+d). To bypass this requirement, the

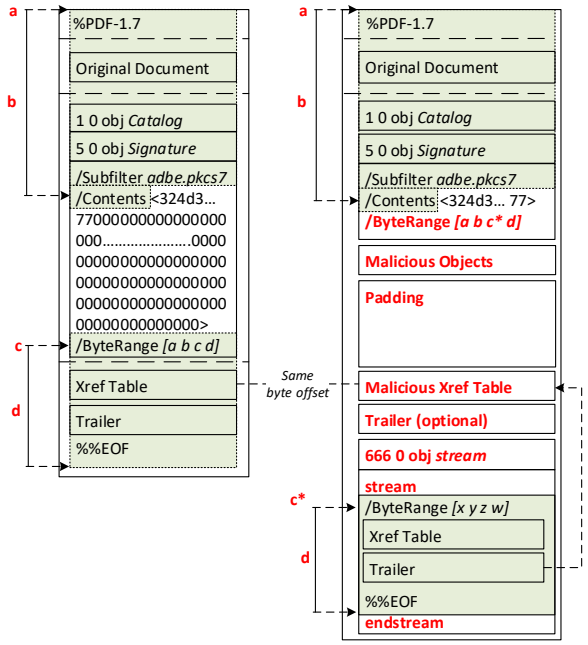


Figure 9: A comparison of the original document and the manipulated document by using the Signature Wrapping Attack (SWA). Malicious objects are placed before the malicious *Xref table* table by deleting unused zero bytes in `Contents`.

[1] During signing, the size of the signature value (and the corresponding certificate) is not known and thus it is roughly estimated. The unused bytes are later filled with zero Bytes.

PDF file is not correctly closed. However, it will still be processed by any viewer.

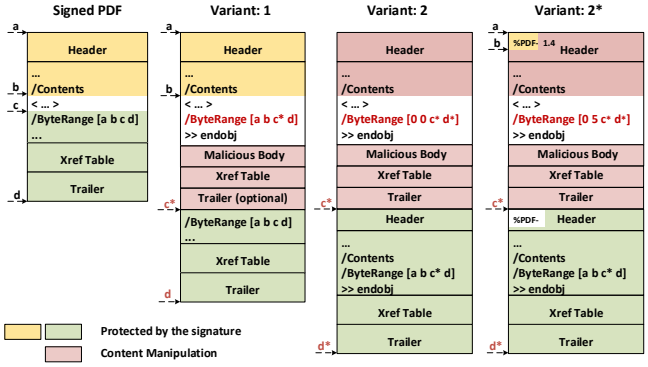


Figure 10: File structures of a signed PDF file before and after different Signature Wrapping attacks were applied.

Variant 2: Relocating the both hashed parts. The first variant of SWA only relocates the second hashed part. This approach has the disadvantage that manipulations in the first section are impossible since the first hash part commonly protects the beginning of the file (offset $a = 0$) up to the signature object. The second variant of SWA relocates both hashed parts by concatenating part 1 and part 2. The attack algorithm is similar to Variant 1, except for two differences:

- In Step 2, the attacker changes all original values in `/ByteRange` to a^*=0, b^*=0, c^* and d^*=b+d. In other words, he defines the first hashed part to begin at byte offset `a*=0`, having length `b*=0`. He then chooses an arbitrary wrapper position `c*`, and sets its length (d) to the sum of both hashed parts (`b+d`).
- In Step 6, the attacker copies the first hashed part (byte offsets a to a+b) concatenated with the second hashed part (byte offsets c to c+d) at byte offset $c*$.

The algorithm is based on our evaluation result that all tested viewer applications verified if the first entry of the `/Byterange` equals zero. This makes it impossible to move the first hashed part to an arbitrary position because of $a > 0$ and leads to a warning. For this reason, we used the trick to concatenate both hashed parts to a single unit. By this means, the value of a could remain zero. Surprisingly, no viewer verified whether $b > 0$, but even in such a case, we can apply SWA. A lightly different **Variant 2*** can be created by using the fact that the beginning of every PDF file starts with `\%PDF-` followed by the specified interpreter version, e.g., *1.7*. Therefore, a byte range from byte offsets a=0, . . . , b=5 can always be used. A comparison of all SWA variants is depicted in Figure 10.

5 EVALUATION

In this section, we present the results of our evaluation. We applied various manipulations based on the three presented attack classes to a validly signed test document. Afterward, we conducted black-box security tests to evaluate whether native applications or online validation services in the scope of this paper can be successfully attacked using our attack classes.

5.1 Applications

In the first phase of our evaluation, we searched for desktop applications validating digitally signed PDF files. We analyzed the security of their signature validation process against our three attack classes.

The 22 applications listed in Table 1 fulfill these requirements. We evaluated the latest versions[2] of the applications on all supported platforms (Windows, MacOS, and Linux).

Results. During our evaluation, we identified vulnerabilities in 21 of the 22 evaluated applications. These vulnerabilities allow us to bypass the document integrity protection provided by the signature completely and to manipulate the displayed content of signed PDF files. There was only one application which could not successfully be attacked: the last Linux version of *Adobe Reader* (9.5.5) which was released in 2013. All other applications could be successfully attacked using at least one attack vector. The SWA class turned out to be the most successful. It led to successful attacks on 17 applications, while ISA could be used to successfully attack half of the evaluated applications and USF succeeded for four applications.

In the following section, we present interesting results as an example for each attack class. The complete results are depicted in Table 1.

Universal Signature Forgery. USF attacks were successful against four applications. However, two of these applications are *Adobe Acrobat Reader DC* and *Adobe Reader XI*. Surprisingly, these two applications are not vulnerable to any attack we evaluated except the USF attack. To bypass the protection of the applied digital signature in *Adobe Acrobat Reader DC* and *Adobe Reader XI* an attacker only needs to remove the `/ByteRange` entry of the signature object which specifies the part of the document protected by the signature, or replace its value with *null*.[3] Afterwards, he can arbitrarily change the displayed content of the document. Nevertheless, both applications showed a blue banner stating that the document is "Signed and all signatures are valid". The applications also informed the user in the signature panel that the document "has not been modified since the signature was applied" although the manipulated content was displayed.

Incremental Saving Attack. By using the ISA class, it was possible to attack 11 of the 22 evaluated applications successfully. For example, *PDF Studio Viewer 2018* and *Perfect PDF 10 Premium* inform the user that the document has been changed after the application of the signature when a regular incremental saving is applied to a signed document. However, it is sufficient to delete the *Xref table* and trailer of the incremental saving and add the keyword `startxref` as a comment at the end of the file to create a successful attack for these applications. When the manipulated document is opened, the applications display the exchanged content but still inform the user that the applied signature is valid and the document has not been changed since it was applied.

We found two even easier bypasses of the document integrity protection for *LibreOffice*. Both bypasses are based on Variant 1 of the ISA class, whose structure is very similar to regular incremental saving. The manipulated files both contain body updates, a new *Xref table*, and a new trailer but differ in the contents of the *Xref*

[2] Which were available at the beginning of our evaluation.
[3] According to the PDF reference v1.7, a dictionary entry whose value is *null* should be treated similar to a non present entry [21, p. 63].

PDF Viewer	Version	OS	PDF Signature			Comments
			USF	ISA	SWA	
Adobe Acrobat Reader DC	2018.011	Win10, MacOS	●	✓	✓	Error when a visible signature is clicked, for invisible signatures this is not a problem
Adobe Reader 9	9.5.5	Linux	✓	✓	✓	
Adobe Reader XI	11.0.10	Win10, MacOS	●	✓	✓	Error when a visible signature is clicked, for invisible signatures this is not a problem
eXpert PDF 12 Ultimate	12.0.20	Win10	✓	✓	●	
Expert PDF Reader	9.0.180	Win10	✓	✓	●	
Foxit Reader	9.1.0; 9.2.0	Win10, Linux, MacOS	✓	●	●	No signature verification on Linux and MacOS available (latest version 2.4.1)
LibreOffice (Draw)	6.0.6.2; 6.0.3.2, 6.1.0.3	Win10, Linux, MacOS	✓	◑	✓	Detects ISA when certificate is trusted
Master PDF Editor	5.1.12/24	Linux, Win10, MacOS	✓	●	✓	Attack only on Linux and Windows successful. On MacOS the original, not manipulated signature was already invalid.
Nitro Pro	11.0.3.173	Win10	✓	◑	●	Detects ISA when certificate is trusted
Nitro Reader	5.5.9.2	Win10	✓	◑	●	Detects ISA when certificate is trusted
Nuance Power PDF Standard	3.0.0.17	Win10	✓	●	✓	
PDF Architect 6	6.0.37	Win10	✓	✓	●	
PDF Editor 6 Pro	6.4.2; 6.6.2	Win10, MacOS	◑	●	●	USF successful on UI-Layer 1; ISA and SWA only on Windows successful. On MacOS the original, not manipulated signature was already invalid.
PDFelement 6 Pro	6.7.1; 6.8.0	Win10, MacOS	◑	●	●	USF successful on UI-Layer 1; ISA and SWA only on Windows successful. On MacOS the original, not manipulated signature was already invalid
PDF Studio Viewer 2018	2018.0.1	Win10, Linux, MacOS	✓	●	●	
PDF Studio Pro	12.0.7	Win10, Linux, MacOS	✓	●	●	
PDF-XChange Editor	7.0.326	Win10	✓	✓	●	
PDF-XChange Viewer	2.5	Win10	✓	✓	●	
Perfect PDF 10 Premium	10.0.0.1	Win10	✓	●	●	
Perfect PDF Reader	13.0.3	Win10	✓	●	●	
Soda PDF Desktop	10.2.09	Win10	✓	✓	●	
Soda PDF	9.3.17	Win10	✓	✓	●	
Total Successful Attacks			**4/22**	**11/22**	**17/22**	
			Summary Signature Vulnerabilities: 21/22			

✓: Secure/Attack fails; ●: Insecure/Attack successful; ◑: Limited attack success

Table 1: Evaluation results of 22 PDF Viewer showing critical vulnerabilities in 21 of them.

table. In contrast to regular incremental saving, the *Xref table* of the first bypass is empty and only consists of the keyword `xref`. We presume that *LibreOffice* assumes that the incremental saving does not add new objects due to the empty *Xref table*. The second bypass uses an *Xref table* which does not only contain entries for the body updates, as would be the case for regular incremental saving, but also entries for all objects added to the file when the signature was applied. *LibreOffice* seems to assume that the signature is part of this manipulated incremental saving, and therefore informs the user that the document was not modified after the signature was applied.

Signature Wrapping Attack. Attacks based on the SWA attack class were the most successful against the viewer applications. All but five applications are vulnerable to attacks of this class. It was even possible to attack 14 applications with a single manipulated document successfully. This document was created by adding new objects manipulating the displayed content of the document, a new *Xref table*, and a new trailer in between the two signed byte ranges. Two things are essential for the attack to work: (1) The *Xref table* contains entries for all added objects, and objects present in the second signed byte range. (2) The last trailer of the file, as well as the newly added one, must reference the correct byte offset of the new *Xref table*. The second signature wrapping approach – moving the signed data to the end of the file – led to another interesting, however not successful, result. When the different test files, which were created for this signature wrapping approach and led not to

successful attacks, are opened in *eXpert PDF 12 Ultimate*, *PDF Architect 6*, *Soda PDF Desktop* or *Soda PDF* the application's signature panel states that "some modifications have been made in the document". Some of these test files are called "Revision 1" and some are called "Revision 2" in the signature panel. The application's behavior when the "View Signed Version" option in the signature panel is selected differs for these two revisions. For all files called Revision 1, the option opens a new tab showing the original file's content ("Hello World!" for our test files). However, for all files called Revision 2, the opened tab also displays the manipulated content ("Hello Attacker!"), and the signature panel now states that "after adding the signature, the document has not been modified". This implies to the user that the opened document has been altered after the signature was applied; nevertheless, the content displayed in the new tab after clicking on "View Signed Version" is the original file content. These attacks are not classified as successful because the attacker model specifies that both UI-layers must not state that the document was modified after the application of the signature when the manipulated document is opened.

5.2 Online Validation Services

In the second phase of our evaluation, we focused on online validation services. These services are used to verify the integrity and validity of signed PDF documents. Thus, validating the signature of PDF documents can be automated and outsourced to these services.

One of the most prominent vendors of validation services is DocuSign. Aside from its online validation service, DocuSign also offers a cloud PDF viewer and a signing application used by most companies of the Fortune 500 list. Prominent examples include Dell, eBay, VISA, Microsoft, Nike, and the *USENIX* Association [4, 13].

We additionally evaluated services used in different EU countries (e.g., Austria [38] or Slovenia [10]) to evaluate multiple signatures types (PAdES, CAdES, and XAdES) for the eIDAS regulation [48].

Test Setup. We evaluated each online validation service as follows. First, we uploaded a validly signed PDF file (`document_signed.pdf`) to the service by using the available upload functionality. The service then generates a report containing details regarding the signature validity status. Another output was not provided in any case, especially the content of the PDF file is not displayed.

We then modified the signed PDF file using different variants of all three attack classes successively. If one of these attack vectors results in a report that is indistinguishable from the report of `document_signed.pdf`, we classify the attack as successful. An example of a successful attack is presented in Figure 11.

Results. We analyzed eight free and publicly available validation services against all three attack classes. The signature validation could be bypassed on six services (cf. Table 2).

To summarize, two of the analyzed services [9, 38] were vulnerable to SWA and five services [9, 10, 12, 14, 20] could be bypassed using the ISA class. This is contrary to the results from the evaluation of viewer applications, where we could find more applications vulnerable to SWA.

One interesting challenge during the evaluation was to find a clear indication in the report whether a signature is valid. For example, the DSS Demonstration WebApp [14] prints out two fields containing the verification report: *Indication* and *Signature scope*, see

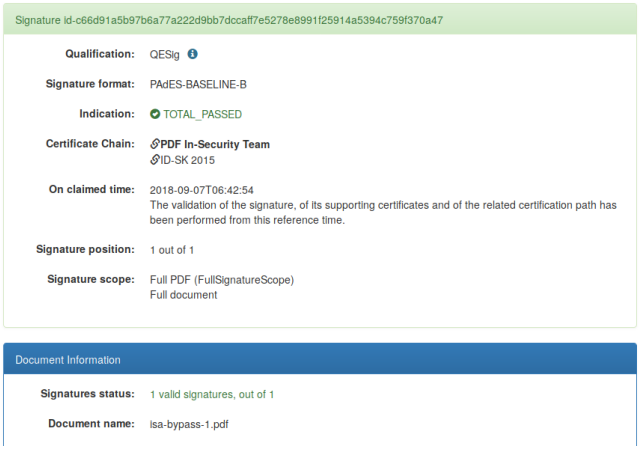


Figure 11: Validation report created by the Digital Signing Service for a manipulated but signed PDF file [14].

Figure 11. The *Indication* field summarizes the results of the digital signature validation. In our case, the result is: *TOTAL PASSED*. With respect to USF and SWA we received a warning or a error message if the attacks are detected. Regarding ISA, the *Signature Scope* contains information indicating whether the complete document is signed or not. In case that the ISA attack is detected, the validation service should print out that the scope is *partial* and only parts of the document are signed. According to our evaluation, version 5.2 of the DSS Demonstration WebApp is susceptible against ISA since it returns a *Full PDF* as *Signature scope* even if the document was modified via incremental saving in **Variant 2**. Along with all EU validation services, we analyzed DocuSign – one of the worldwide leading cloud services – was the only service vulnerable against both attacks ISA and SWA.

6 HOW TO FIX PDF SIGNATURES

In this section, we propose concrete countermeasures to fix the previously introduced attacks. We,therefore, carefully studied the main reasons for the attacks on PDF signatures and were able to identify two root causes: (1) The specification does not provide any information with a concrete procedure on how to validate signatures. There is no description of pitfalls and any security considerations. Thus, developers must implement the validation on their own without best-common-practice information. (2) The error tolerance of the PDF viewer is abused to create non-valid documents bypassing the validation, yet is correctly displayed to the user.

The Verification Algorithm. When considering a proper countermeasure, we defined an algorithm which addresses USF, ISA, and SWA but does not negatively affect the error tolerance of the PDF viewers (cf. Listing 2). It describes a concrete approach on how to compute the values necessary for the verification and how to detect manipulations after the PDF file was signed. The specified algorithm must be applied for each signature within the PDF document.

Signature Validation Service	Version	PDF Signature			Comments
		USF	ISA	SWA	
DocuSign [9]	v1 REST API with PDFKit.NET 18.3.200.9768	✓	●	●	
eRTR Validation Service [38]	v 2.0.3	✓	●	●	
DSS Demonstration WebApp [14]	WebApp 5.2	✓	●	✓	
DSS Demonstration WebApp [7]	WebApp 5.3.1	✓	✓	✓	
Evrotrust (free) [12]	12.0.20	✓	●	✓	
Ellis [20]	version 0.9.1, build 1526594400	✓	●	✓	
VEP.si [10]	2017-06-26	✓	●	✓	
SiVa Sample Application [11]	release-2.0.1	–	–	–	Could not be evaluated since valid documents were shown invalid due to PKI issues
Total Successful Attacks		**0/8**	**5/8**	**2/8**	
		Summary Signature Vulnerabilities: 6/8			

✓: Secure/Attack fails; ●: Insecure/Attack successful; ◑: Limited attack success

Table 2: Evaluation results of eight online signature validation services showing six of them vulnerable.

As an input, it requires the PDF file as a byte stream and the signature object.

```
1  INPUT: PDFBytes, SigObj
2
3  // ByteRange is mandatory and must be well−formated
4  byteRange = SigObj.getByteRange
5  // Preventing USF:
6  if (byteRange == null OR byteRange.isEmpty) return false
7  // Parse byteRange
8  if (byteRange.length≠4) return false
9  for each x in byteRange { if x ≠ instanceof(int) return false}
10 a, b, c, d = byteRange
11 // BytRange must cover start of file
12 if (a ≠ 0) return false;
13 // Ensure that more than zero bytes are protected in hashpart1
14 if (b ≤ 0) return false
15 // Ensure that sencond hashpart starts after first hashpart
16 if (c ≤ b) return false
17 // Ensure that more than zero bytes are protected in hashpart2
18 if (d ≤ 0) return false
19 // Preventing ISA. ByteRange must cover the entire file.
20 if ((c + d) ≠ PDFBytes.length) return false;
21 // The pkcs7 blob starts at byte offset (a+b) and goes to offset c
22 pkcs7Blob = PDFBytes[(a+b):c]
23 // Preventing USF. Pkcs7Blob value is not allowed to be null or empty.
24 if (pkcs7Blob == null OR pkcs7Blob.isEmpty) return false
25 // pkcs7Blob must be a hexadecimal string [0−9,a−f,A−F]
26 if (pkcs7Blob contains other chars than [0−9,a−f,A−F]) return false
27 // Parse the PKCS#7 Blob
28 sig, cert = pkcs7.parse(pkcs7Blob)
29 // Select (a+b) bytes from input PDF begining at byte a=0, i.e. 0 ... a+b−1
30 hashpart1 =PDFBytes[a:(a+b)]
31 // Select (c+d) bytes from input PDF begining at byte c, i.e. c ... c+d−1
32 hashpart2=PDFBytes[c:(c+d)]
33 // Verify signature
34 return pkcs7.verify(sig, cert, hashpart1||hashpart2)
```

Listing 2: Pseudo-code preventing USF, ISA and SWA.

In Line 4, we first extract the `ByteRange` from the signature object. To prevent USF, we ensure that `ByteRange` is not `null` or `empty` in Line 7.

Lines 9-22 then validate the values a, b, c, and d of the `ByteRange`. First, Line 10 ensures that it contains exactly four values to minimize an attacker's attack surface. Line 11 additionally ensures that each `ByteRange` value is an integer. Lines 14 to 20 ensure that `ByteRange` satisfies the following condition: $0{=}a < b < c < (c{+}d)$, which is equivalent to $a{=}0$ *and* $b > 0$ *and* $c > b$ *and* $d > 0$. Enforcing this condition ensures that the signature always covers the beginning of the file ($a = 0$), prevents signed blocks of length zero ($b > 0, d > 0$), and ensures that both signed blocks are non-overlapping ($c > b$). Finally, we verify that `ByteRange` covers the entire file (Line 22) in order to detect ISA. Lines 24-29 parse the `Contents` parameter of the signature object, which is a PKCS#7 blob.

The critical aspect is that we interpret everything that is not covered by the `ByteRange` as the `Contents` parameter of the PDF signature. Theoretically, the check in Line 27 should never fail, because we previously verified $(a{+}b){=}b$ *and* $b < c$. Thus it holds that pkcs7Blob.length > 0. Nevertheless, we leave this line here due to its importance for preventing SWA. Line 29 additionally ensures that only hex characters can be in the unprotected part of the PDF file, preventing further unwanted modifications of the file.

Lines 31-32 parse the PKCS#7 blob and extract the information to be used for the signature verification. Lines 34-38 determine the bytes of the input PDF that are signed.

Finally, Line 41 calls the PKCS#7 verification function and returns the validity status of the signature.

Drawback. Specifying the algorithm in Listing 2 requires a change in the PDF specification which defines `ByteRange` as an optional parameter [21, Section 8.7]. In this case, the signature value will be computed only over the signature dictionary leaving the entire document unprotected. Such a feature allows an even more powerful attack since the attacker can create validly signed documents by only injecting the signed signature dictionary without a `/ByteRange`. Currently, none of the evaluated viewers supports this feature.

Additionally, the algorithm leads to one usability issue if multiple signatures are provided. Although these signatures are valid, only the one covering the entire document will be displayed as valid. This problem can be addressed by providing additional information to the user that some of the signatures are valid but cover only a specific revision and not the entire document. Adobe uses a similar approach for the signature validation. All Adobe viewers show information about the document revision protected by a signature and allow only to open this revision. Thus, a user can easily verify which information is signed and which is not.

Responsible Disclosure. After discovering the vulnerabilities we created a security report containing the description of the attacks, a list with the affected implementations, a proof-of-concept exploit

for each successful attack vector, and the pseudo-code preventing the attacks [34]. On the 8th of November, we sent the report to the BSI-CERT team who distributed it to all affected vendors and governmental organizations dealing with PDF [34]. During the responsible disclosure process, we supported BSI-CERT and the vendors to fix the issues. The complete information relating to our research on PDF signatures was published February 25, 2019 on https://www.pdf-insecurity.org/. To support all vendors, we also published all available exploits. Some vendors already integrated these files in their test environments.

7 RELATED WORK

At the beginning of our research phase, we gathered and studied the existing related work to PDF and file format security. This work can be separated into the following four categories.

PDF Malware and PDF Masking. In 2010, Raynal et al. provided a comprehensive study on malicious PDFs abusing legitimate features in PDFs leading to Denial-of-Service (DoS), Server-Side-Request-Forgery (SSRF), and information leakage [37]. Additionally, the authors considered potential security issues regarding the signature verification by criticizing the design of the certificate trust establishment. In 2012, Hamon et al. published a study revealing weaknesses in PDFs leading to malicious URI invocation [49]. In 2013 and 2014, multiple vulnerabilities in Adobe Reader were reported abusing the support of insecure PDF features, JavaScript, and XML [22, 40]. Inführ [23] published a summary of the supported languages, file formats and features in PDFs leading to these security issues. In 2018, Franken et al. evaluated the security of third-party cookies policies [16]. Part of the evaluation revealed weaknesses in two PDF reader by forcing them to call arbitrary URIs. In the same year, multiple vulnerabilities in Adobe Reader and different Microsoft products were discovered leading to URI invocation and NTLM credentials leakage [24, 39].

Besides PDF malware, research has been provided on content masking. In 2014, Albertini discovered new attack classes by combining a PDF and a JPEG into a single polyglot file [2]. In 2017, Markwood et al. introduced a novel attack related to content masking by using font encoding [31].

PDF Malware Detection. As a result of the discovered attacks during the recent years, different security tools were implemented detecting maliciously crafted documents [8, 26, 28, 30, 41, 43]. Such tools rely on the detection of known attack patterns and structural analysis of PDFs.

In 2016, Carmony et al. build a JavaScript reference extractor for detecting parsing confusion attacks [6]. In 2017, Tong et al. introduced a concept for a robust PDF malware detection based on machine learning algorithms [46]. In the same year, Tong et al. published a framework based on these algorithms and capable of detecting PDF malware [47]. Maiorca et al. provided an overview of the current PDF malware techniques and analyzed the existing security tools by comparing them [29]. This paper mentions the Incremental Saving (IS) feature for the first time in conjunction with attacks, but up until our research, the feature has not been combined with attacks on PDF signatures.

PDF Signatures. While studying the related work, we discovered a gap in existing security analysis. We were able to find only a few articles directly related to the security of PDF signatures.

In 2008 and 2012, Grigg et al. described the risks associated to *electronic signatures* [18, 19] based on the missing cryptographic signature allowing an attacker to forge any signature.

In 2012, Popescu et al. presented a proof-of-concept bypass for a specific digital signature [36]. The attack is based on a polymorphic file containing two different files – a PDF and TIFF. The risk exists if a victim signs the document unaware of the hidden content inside the file. In 2015, Lax et al. documented potential security topics related to digitally signed documents [27]. The authors concentrated on issues related to the signature generation process like malware, signed documents containing dynamic content like macros or JavaScript, and polymorphic documents similar to [36]. In 2017, Stevens et al. discovered an attack against SHA-1 [45] breaking the collision resistance. For the proof-of-concept, the authors created two different PDF files containing the same digest value. As a result, an attacker could create a PDF file with new content without invaliding the digital signature. In his master thesis, Stefan et al. provided an in-depth analysis of PDF signatures [44]. The author also implemented a library verifying PDF signatures. However, the security considerations addressed only known attacks related to PDFs and none of our discovered attack classes.

Signature Bypasses in different Data Structures. In 2002 Kain et al. addressed possible threats related to digitally signed documents like MS Word, MS Excel, or PDFs resulting from PKI issues, dynamic content loaded from a website, and code execution by supported programming languages within documents [25]. In the paper, the authors briefly describe the possibility to create an unsigned PDF document which is visually identical to the signed one, but they do not deliver any proof-of-concept exploit and do not evaluate if and how this can be achieved. In 2005, Buccafurri et al. describe a file format attack where the attacker forces two different views of the same signed document which contains an image as `BMP` and HTML code [5]. Depending on the file extension, the content of the image or the HTML code is processed. PDF files are mentioned as a possible target for such an attacker, but no concrete ideas are described.

The general concept of SWA – the relocation of the hashed part of a document – has been applied to XML-based messages before. In 2005, McIntosh and Austel described an XML rewriting attack on SOAP web services [32] and was adapted to SAML-based Single Sign-On in 2012 [42]. However, the adaption to PDF is much more complicated because the hashed part of the file is located using a byte range instead of an object identification number and has not been found in any previous work.

Attacks that exclude a document's signature have been applied to SAML [42] and JSON [33]. In contrast to our USF attack, these vulnerabilities simply remove the signature of the document in order to bypass the validation logic. This would work identically for PDFs, but a victim expects to open a signed file, and he will become suspicious if no signature information is shown once he opens the document. Thus, USF is a more advanced variant of signature exclusion adapted to PDF.

8 NEW RESEARCH DIRECTIONS

In this paper, we provide the first step into the security analysis of PDF signatures. We discovered further potential targets for attacks opening new research directions and challenges.

PKCS-based Attacks. The signature value is either a DER-encoded PKCS#1 binary data object or a DER-encoded PKCS#7 binary data object. Considering the complexity of both formats, the question arises if the verification of the PKCS object is correctly implemented. The goal of PKCS-based attacks is the creation of an *always valid* object. The impact of such an attack would be equal to the impact of USF, whereby any modification of the signed document is possible.

Additionally, the PKCS object contains the certificates used during the verification. If untrusted certificates are used, security warnings are displayed to the user. Thus, an attacker is not able to create a validly signed and trusted document. Future research should concentrate on the certificate validation by targeting this step and forcing the validation to accept an untrusted certificate.

Transformation Method Attacks. The PDF specification defines three different transformation methods applied on the document before signing it: *DocMDP*, *UR*, and *FieldMDP*. The transformation methods define which objects are included and excluded in the computation of the digital signatures. In this paper, we focused on the *DocMDP* transformation which is the short term for *modification, detection, and prevention* and permits changes by filling in forms, instantiating page templates and signing. Any other modification invalidates the signature.

DocMDP allows further adjustments regarding permitted and forbidden changes depending on different parameters. Future research should investigate if such restrictions are correctly applied and if they can be bypassed. Additionally, the transformation methods *UR*, protecting the defined usage rights, and *FieldMDP* detects changes in contained form fields should be also analyzed. Since these transformation methods process the data which should be signed differently than *DocMDP*, an in-depth security analysis could discover further vulnerabilities.

PDF Advanced Electronic Signatures. Motivated by the idea of eGovernment, the European Union published the PDF Advanced Electronic Signatures (PAdES) specification, which extends the PDF signature specification. For the significance of sensitive documents exchanged within governmental services, it is essential to analyze the current specification and the existing implementations.

In our evaluation, we discovered vulnerabilities in online validation services by adapting our attack vectors on PAdES documents. Since our attacks abuse features in the PDF specification, it is not surprising that PAdES signatures are also affected. It is essential that future research analyzes the PAdES specification carefully and evaluates the security of the specification itself. In this paper, we did not provide such an analysis.

Content Masking. Markwood et al. introduced techniques to bypass topic matching algorithms, plagiarism detection, and document indexing [31] by creating malicious fonts and constructing new word and character maps to mask the malicious content. In the context of signed PDFs, content masking attacks abuse dissimilarities between the signed and displayed content. For example, by defining new fonts and thus changing the presentation of some characters, the IBAN in an invoice document can be changed.

Another attack idea is to abuse the error tolerance of the viewer. During our tests, we detected presentation differences of the same document by using different viewers. The error-tolerance can be abused by an attacker validly signing one document, for example, a contract and distributing it to multiple parties. If these use different viewers, they may accept different contracts.

Verification UI Forgery. Similar to content masking attacks, an attacker can try to create a UI forging the view of a signed document. The PDF specification supports multiple interactive forms like *button fields*, *rich text string*, and *form actions*. Such features facilitate the creation of a UI imitating a signature panel where the results of the signature validation are usually displayed. As a result, an attacker could create a malicious document which appears trustworthy after opening. These kinds of attacks have already been described in the web context by Zalewski [54]. Researchers should concentrate on features defined in Section 12 in the PDF specification [21].

9 CONCLUSION

The PDF specification is a very complex standard. Unfortunately, when it comes to cryptography and, as in our case to digital signatures, it lacks concrete implementation guidelines and documents describing the best current practices. Our investigation reveals that almost all desktop applications fail to validate PDF signatures correctly. We identified three main reasons for this: (1) The specification itself does not enforce a strict policy, e.g., it does not enforce a signature to cover the whole document. This could be abused by SWA and relocating the signed content to a different position. (2) PDF applications are error tolerant and process the content of a PDF file even if it is not standard compliant. We heavily abused this behavior with ISA and created non-standard compliant documents that force a viewer application to believe that it has not been updated; however, an attacker could manipulate the document. (3) Even if the above aspects are correctly handled, as in the case of Adobe, there can be mere programming mistakes that break the whole cryptography. In the case of USF, an unexpected missing of mandatory information leads to a valid signature.

Our evaluation of PDF viewer applications and online validation services has alarming results. In 95% of all analyzed viewer applications, at least one of the problems, which were identified above, occurs and allows an attacker to stealthy manipulate contents of a signed PDF file. Analogous results could be found for online validation services in 75% of the tested cases. We responsibly disclosed our findings via the BSI-CERT to all vendors and proposed a validation algorithm to prevent our attacks.

Concerning the digitalization of offices and eGovernment, we see a strong need for the improvement of the given specification and best practices. PDF security related to cryptographic features have been overlooked for too long. We, therefore, pointed out new research directions in the field of PDF security in order to address this issue.

10 ACKNOWLEDGEMENTS

We would like to thanks the CERT-Bund team for their great support during the responsible disclosure process. The research was supported by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1). Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

REFERENCES

[1] Adobe. 2018. *Adobe Fast Facts.* https://www.adobe.com/about-adobe/fast-facts.html

[2] Ange Albertini. 2014. This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats. *PoC 11 GTFO 0x03* (2014). https://www.alchemistowl.org/pocorgtfo/pocorgtfo03.pdf

[3] PDF association. 2018. *PDF in 2016: Broader, deeper, richer.* https://www.pdfa.org/pdf-in-2016-broader-deeper-richer/

[4] USENIX Association. 2018. *Board of Directors Out of Band Motion.* https://www.usenix.org/sites/default/files/2017-01_out-of-band_motion_signed.pdf

[5] Francesco Buccafurri. 2005. Digital Signature Trust vulnerability: A new attack on digital signatures. *Information Management & Computer Security* 4 (2005), 28–6. http://www.unirc.it/firma/en/Buccafurri_ISSA_1008.pdf

[6] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors.. In *NDSS*.

[7] European Commission. 2018. *DSS Demonstration WebApp v5.3.1.* https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/DSS

[8] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. 2014. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*. ACM, 47–57.

[9] Inc. DocuSign. 2018. *DocuSign Validation Service.* https://validator.docusign.com/

[10] EIUS doo. 2018. *VEP E-obrazci.* https://www.vep.si/validator/forms/document-verify

[11] eesti. 2018. *SiVa Demo application.* https://siva-arendus.eesti.ee/

[12] Evrotrust. 2018. *Validate a signature.* https://www.evrotrust.com/landing/en/a/validation

[13] FeaturedCustomers. 2018. DocuSign Customer. https://www.featuredcustomers.com/vendor/docusign/customers

[14] Agency for Digital Italy. 2018. *DSS Demonstration WebApp v5.2.* https://dss.agid.gov.it/validation

[15] Forbes. 2018. *Forbes Releases 2017 Cloud 100 List of the Best Private Cloud Companies in the World.* http://bit.ly/dokusign-forbesrank

[16] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 151–168. https://www.usenix.org/conference/usenixsecurity18/presentation/franken

[17] Bundesministerium für Digitalisierung und Wirtschaftsstandort. 2019. E-Government-Gesetz (E-GovG). https://www.ris.bka.gv.at/GeltendeFassung/Bundesnormen/20003230/E-GovG%2c%20Fassung%20vom%2004.02.2019.pdf

[18] Ian Grigg. 2008. Technologists on signatures: looking in the wrong place. http://financialcryptography.com/mt/archives/001056.html

[19] Ian Grigg. 2012. Signatures on fax & email - if you did not intend to be bound, why did you bother to write it? http://financialcryptography.com/mt/archives/001364.html

[20] Arhs Group. 2018. *Ellis Digital Signature.* https://ellis.arhs-spikeseed.com/

[21] Adobe Systems Incorporated. 2006. *PDF Reference, version 1.7* (sixth edition ed.).

[22] Alexander1 Inführ. 2014. *Multiple PDF Vulnerabilities – Text and Pictures on Steroids.* https://insert-script.blogspot.de/2014/12/multiple-pdf-vulnerabilites-text-and.html

[23] Alexander Inführ. 2015. *PDF – Mess with the Web.* https://2015.appsec.eu/wp-content/uploads/2015/09/owasp-appseceu2015-infuhr.pdf

[24] Alexander2 Inführ. 2018. *Adobe Reader PDF - Client Side Request Injection.* https://insert-script.blogspot.de/2018/05/adobe-reader-pdf-client-side-request.html

[25] K Kain, Sean W Smith, and R Asokan. 2002. Digital signatures and electronic documents: A cautionary tale. In *Advanced communications and multimedia security*. Springer, 293–307. http://www.ists.dartmouth.edu/library/74.pdf

[26] Pavel Laskov and Nedim Šrndić. 2011. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th annual computer security applications conference*. ACM, 373–382.

[27] Gianluca Lax, Francesco Buccafurri, and Gianluca Caminiti. 2015. Digital document signing: Vulnerabilities and solutions. *Information Security Journal: A Global Perspective* 24, 1-3 (2015), 1–14.

[28] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. 2015. A structural and content-based approach for a precise and robust detection of malicious pdf files. In *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. IEEE, 27–36.

[29] Davide Maiorca and Battista Biggio. In Press. Digital Investigation of PDF Files: Unveiling Traces of Embedded Malware. *IEEE Security and Privacy: Special Issue on Digital Forensics* (In Press). https://pralab.diee.unica.it/sites/default/files/maiorca17-sp.pdf

[30] Davide Maiorca, Giorgio Giacinto, and Igino Corona. 2012. A pattern recognition system for malicious pdf files detection. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 510–524.

[31] Ian Markwood, Dakun Shen, Yao Liu, and Zhuo Lu. 2017. PDF Mirage: Content Masking Attack Against Information-Based Online Services. In *26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC)*. 833–847.

[32] Michael McIntosh and Paula Austel. 2005. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 Workshop on Secure Web Services*. ACM Press, New York, NY, USA, 20–27.

[33] Tim McLean. 2015. Blog post: Critical vulnerabilities in JSON Web Token libraries. https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html

[34] Vladislav Mladenov, Christian Mainka, Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 2018. *Vulnerability Report: Attacks bypassing the signature validation in PDF.* Technical Report. Ruhr Univeristy Bochum, Chair for Network and Data Security. https://www.nds.ruhr-uni-bochum.de/research/publications/vulnerability-report-attacks-bypassing-signature-v/

[35] United States Government Printing Office. 2000. ELECTRONIC SIGNATURES IN GLOBAL AND NATIONAL COMMERCE ACT. https://www.govinfo.gov/content/pkg/PLAW-106publ229/pdf/PLAW-106publ229.pdf

[36] Dan-Sabin Popescu. 2012. Hiding Malicious Content in PDF Documents. *CoRR* abs/1201.0397 (2012). arXiv:1201.0397 http://arxiv.org/abs/1201.0397

[37] F. Raynal, G. Delugré, and D. Aumaitre. 2010. Malicious Origami in PDF. *Journal in Computer Virology* 6, 4 (2010), 289–315. http://esec-lab.sogeti.com/static/publications/08-pacsec-maliciouspdf.pdf

[38] RUNDFUNK UND TELEKOM REGULIERUNGS-GMBH. 2018. *RTR - Signatur-Prüfung.* https://www.signatur.rtr.at/de/vd/Pruefung.html

[39] Check Point Research. 2018. NTLM Credentials Theft via PDF Files. https://research.checkpoint.com/ntlm-credentials-theft-via-pdf-files/

[40] Billy Rios, Federico Lanusse, and Mauro Gentile. 2013. Adobe Reader Same-Origin Policy Bypass. http://www.sneaked.net/adobe-reader-same-origin-policy-bypass

[41] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th annual computer security applications conference*. ACM, 239–248.

[42] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. 2012. On Breaking SAML: Be Whoever You Want to Be. In *21st USENIX Security Symposium*. Bellevue, WA.

[43] Nedim Šrndić and Pavel Laskov. 2016. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security* 2016, 1 (2016), 22.

[44] Tomáš Stefan. 2017. Digital Signature Verification in PDF. https://dspace.cvut.cz/bitstream/handle/10467/76810/F8-BP-2018-Stefan-Tomas-thesis.pdf?sequence=-1

[45] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. In *Annual International Cryptology Conference*. Springer, 570–596.

[46] Liang Tong, Bo Li, Chen Hajaj, and Yevgeniy Vorobeychik. 2017. Feature Conservation in Adversarial Classifier Evasion: A Case Study. *CoRR* abs/1708.08327 (2017). https://pdfs.semanticscholar.org/f1f8/6dbd8b39c9601e6315214783343ca18377b4.pdf

[47] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, and Yevgeniy Vorobeychik. 2017. A Framework for Validating Models of Evasion Attacks on Machine Learning, with Application to PDF Malware Detection. *arXiv preprint arXiv:1708.08327* (2017). https://arxiv.org/pdf/1708.08327.pdf

[48] European Union. 2014. REGULATION (EU) No 910/2014 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32014R0910

[49] H. Valentin. 2012. Malicious URI resolving in PDF Documents. *Blackhat Abu Dhabi* (2012).

[50] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe and Jörg Schwenk",. 2018. CVE-2018-16042 (Universal Signature Forgery).

[51] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe and Jörg Schwenk",. 2018. CVE-2018-18688 (Incremental Saving Attack).

[52] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe and Jörg Schwenk",. 2018. CVE-2018-18689 (Signature Wrapping Attack).

[53] Wikipedia. 2019. Electronic signatures and law. https://en.wikipedia.org/wiki/Electronic_signatures_and_law

[54] Michal Zalewski. 2012. *The tangled Web: A guide to securing modern web applications.* No Starch Press.