



\$1 Yacht Sale: Tampering with Signed PDFs in Plain Sight

Enzo Brum¹, Frederico Schardong¹, Ricardo F. Custódio¹

¹Laboratório de Segurança em Computação (LabSEC)
Universidade Federal de Santa Catarina (UFSC)

enzo.brum@grad.ufsc.br, frederico.schardong@ufsc.br,
ricardo.custodio@ufsc.br

Abstract. *The Portable Document Format (PDF) is a popular way of sharing electronic documents. It is a format used across fields to represent official documents, many of which require strong guarantees that no unauthorized modifications are made to their contents. Digital signatures can be used to protect both the integrity and authenticity of a document, allowing an entity, whether a person, business, or service, to testify to one specific state of the document such that any non-permitted modification, for example altering the value of a contract or a health prescription, invalidates the signature. In this paper we introduce two novel attack classes that abuse flaws in the PDF specification to visually modify a page of a signed document without invalidating the signature or warning the user: Appearance Substitution Attack (ASA) and Signature Field Duplication Attack (SFDA). While previous works already explored incremental updates, we tap into modifications to the signature field itself, a specific kind of update that has yet to be fully explored. Our evaluation found that 13 out of 16 tested GUI applications and 2 popular verification libraries were at least partially vulnerable to one of the attack classes. We also propose countermeasures and specification improvements for fixing both.*

1. Introduction

The Portable Document Format (PDF) allows documents to be represented consistently across devices and is the de facto standard for electronic documents, common in countless workflows around the globe. Between 2025 and 2026, more than 320 billion documents were opened and more than 8 billion were signed using Adobe Reader [Adobe Inc. 2026a]. PDFs are used to represent contracts, invoices, agreements, and many other kinds of official documents. The format is specified by ISO 32000-2 [ISO 2020] and implemented by a wide range of programs [Adobe Inc. 2026b, Foxit Software Inc. 2026, The Document Foundation 2026].

To ensure that the state of a PDF document is trustworthy, the standard defines PDF signatures, which use asymmetric cryptography over the signed bytes of the document to attest that a given state was approved by the signer and to protect it from subsequent modifications. Their use is legally recognized in Brazil by Law 14.063 [Brazil 2020], in the European Union by the eIDAS Regulation [EU 2014], and in the United States by the Electronic Signatures in Global and National Commerce Act (ESIGN) [U.S. Congress 2000], each granting signed digital documents evidentiary weight comparable to handwritten ones.

PDF signatures, however, are not unbreakable [Mladenov et al. 2019, Mainka et al. 2021, Rohlmann et al. 2021]. In this paper we present two attacks

that visually alter a page of a signed document without invalidating the signature or triggering modification warnings from verifiers: the Appearance Substitution Attack (ASA) and the Signature Field Duplication Attack (SFDA). As shown in Figure 1, an attacker can change the value of a signed contract while the verifier continues to report the signature as valid. The only requirement is access to the signed document.

Existing attacks have either abused verifier flaws to hide changes introduced through such updates [Mladenov et al. 2019], required attacker access to the document before signing [Mainka et al. 2021], or exploited specific permission levels of modification-control mechanisms [Rohlmann et al. 2021]. Both ASA and SFDA, instead, target the signature field itself. While ASA directly modifies an existing signature field, the SFDA duplicates one while placing attacker-controlled visual content on any page of the document. Although prior work [Rohlmann et al. 2021] flagged the combination of incremental updates and signature fields as dangerous, the potential of that combination had not been fully explored.

The contributions of this paper are three fold: (i) we tested 15 GUI applications and 2 popular programming libraries, finding 15 of the 18 to be at least partially vulnerable to of the two attacks; (ii) we propose countermeasures at the specification, verifier, and tooling levels; and (iii) release an open-source tool that detects both attacks in a given PDF. The remainder of this paper is organized as follows. Section 2 introduces the PDF concepts relevant to the attacks, while Section 3 defines the threat model. Then, Section 4 presents ASA and SFDA in detail, Section 5 describes the testing methodology, and Section 6 reports our evaluation. Next, Section 7 discusses countermeasures, while Section 8 relates the attacks to prior work, Section 9 details our ethical considerations and disclosure process, and finally Section 10 concludes the paper.

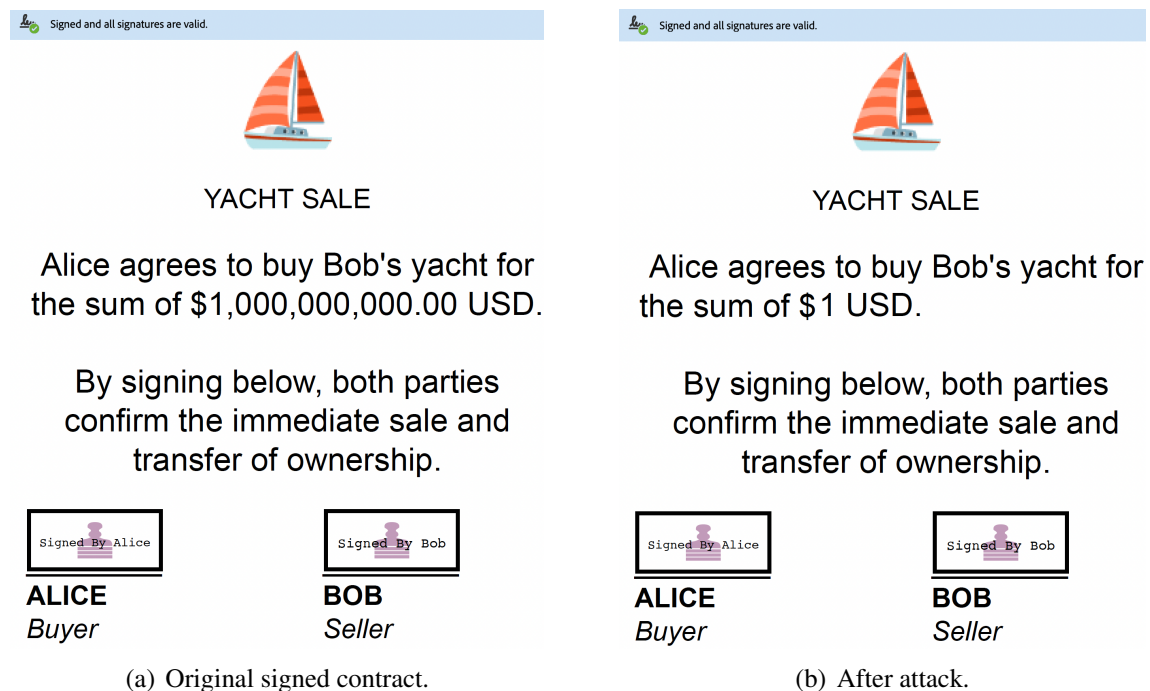


Figure 1. Example of a contract in which the contracted value has been altered after signing, while the signature is still reported as valid.

2. Background

2.1. PDF Structure and Incremental Updates

A PDF document consists of four parts: (i) a header declaring the PDF version; (ii) a body containing the document's objects; (iii) a cross-reference (xref) table mapping each object to its byte offset; and (iv) a trailer pointing to both the xref table and the document's root object. An object is a number, a string, a name, an array, a dictionary, or a stream. In the body, each object appears either as a *direct value*, written inline at the point of use, or as an *indirect object*, declared with a unique identifier of the form `N M obj` and referenced from other objects by `N M R`. In Figure 2(a), the Catalog, a dictionary, is declared as `1 0 obj` and references a Pages dictionary (`2 0 R`), which links to the individual pages of the document, and an AcroForm dictionary (`3 0 R`), which describes any interactive form fields that users can fill or modify. The Catalog is the entry point of the document, and any indirect object is reached following its references.

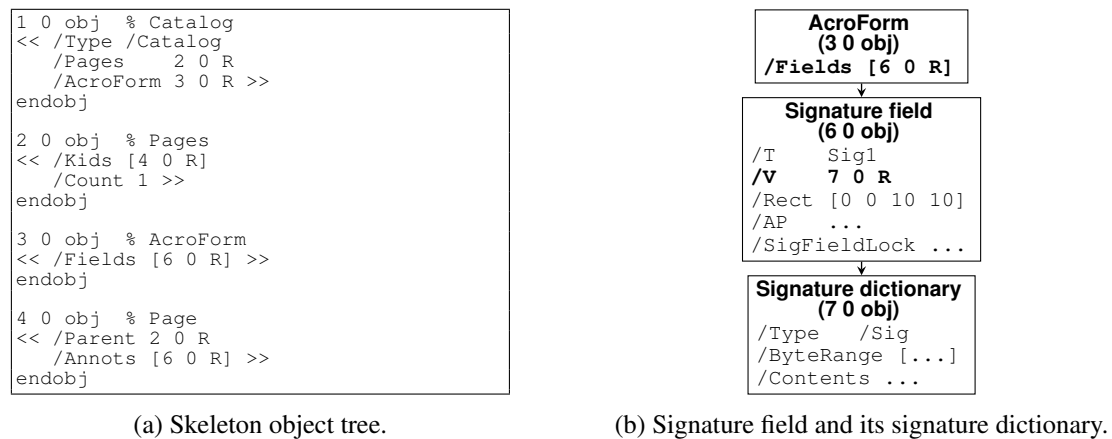


Figure 2. PDF object structure. (a) The Catalog references the Pages and AcroForm dictionaries, and the signature field (object 6) is reached from both the AcroForm's /Fields and its page's /Annots. (b) That signature field and the signature dictionary it references through /v.

When handling updates, a PDF processor can append changes to the file rather than overwriting it. Modified objects are appended after the %EOF marker, and a new xref and trailer are added, creating a new revision of the document. Most PDF readers merge all revisions into a unified view, though some (such as Adobe Acrobat Reader [Adobe Inc. 2026b]) also expose tools for inspecting each revision individually.

2.2. Signature Field

A signature field is a specialized form field that acts as a container for a PDF digital signature. Each signature field is registered in the document's AcroForm dictionary via the /Fields entry and is associated with a single page. Its most relevant attributes are (i) /T, a name uniquely identifying the field within the document; (ii) /V, the reference to the signature dictionary that defines the digital signature itself; (iii) /AP, which defines the field's visual appearance; (iv) /Rect, which specifies the field's position and dimensions on the page; and (v) /SigFieldLock, which controls field-level modification permissions (Section 2.3). Figure 2(b) illustrates this structure.

The PDF specification calls *filling* the operation that supplies input to a form field. For a signature field, filling means updating at least the /V entry by attaching a new signature dictionary, but it can also mean more.

A document is digitally signed by appending a new signature field, linked to a page and to a new signature dictionary, as an incremental update. Each signature dictionary contains the user’s certificate and a byte range covering the full document at signing time; the digital signature is computed over that byte range using the user’s private key and stored within the signature dictionary. As shown in Figure 3, appending signatures as incremental updates supports multiple signatures on the same document. A new signature can be appended without invalidating any prior one, since the bytes covered by earlier signatures remain unchanged.

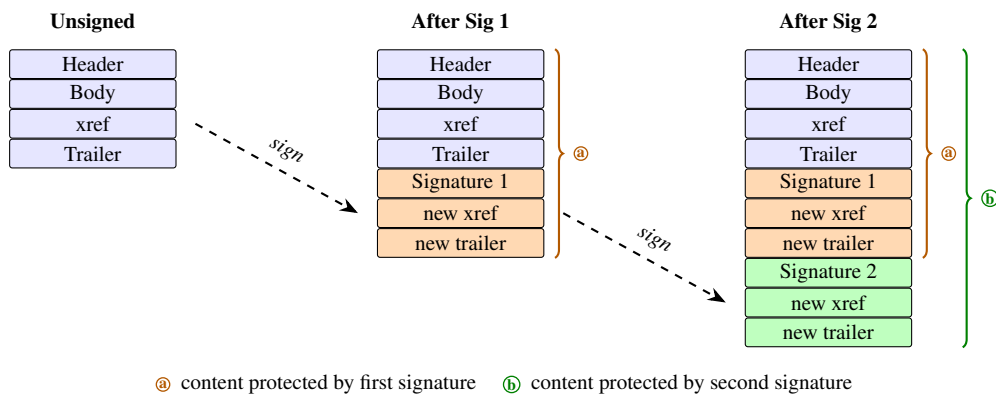


Figure 3. Each signing applies an incremental update; the new signature covers all bytes that existed at signing time, so earlier signatures remain valid.

2.3. Modification Permission Mechanism

Because incremental updates preserve all prior bytes, visual content can be modified after signing without technically invalidating the signature. An attacker could, for example, alter a page or inject an annotation via an incremental update while leaving the signed byte range intact, all within the boundaries of the specification [ISO 2020]. In response, most verifiers [Adobe Inc. 2026b, Foxit Software Inc. 2026, The Document Foundation 2026] either treat such post-signing modifications as invalidating the signature or surface a warning to the user.

The specification also provides mechanisms that make signatures sensitive to specific classes of modifications. DocMDP defines which types of modification are permitted before the signature is considered broken; FieldMDP locks specific form fields so that any change to them invalidates the signature; and SigFieldLock can do both capabilities. DocMDP and FieldMDP reside in the signature dictionary, while SigFieldLock is referenced by the signature field itself and its value is copied into FieldMDP at signing time. DocMDP and SigFieldLock share three permission levels: (i) level 1 forbids any modification; (ii) level 2 permits form fill-in and additional signatures; and (iii) level 3 additionally permits annotation creation, modification, and deletion.

As described in [Rohlmann et al. 2021], PDF distinguishes between *approval* signatures, which attest to the document’s state, and a *certification* signature, which addition-

ally governs which modifications are permissible. A document may carry at most one certification signature, which must appear first; any number of approval signatures may follow. `DocMDP` is exclusive to certification signatures. `FieldMDP` and `SigFieldLock` may be used by either kind. Since PDF 2.0 [ISO 2020], `SigFieldLock` carries the same expressive power as `DocMDP`, allowing approval signatures to impose document-level modification restrictions as well.

3. Threat Model

Our threat model follows the model introduced by [Mladenov et al. 2019]. The attacker’s goal is to alter the visual content of a signed PDF while ensuring that signature verifiers continue to report the original signature as valid and raise no warning about post-signing modifications. We assume the attacker has obtained a PDF carrying a digital signature that the target verifier already trusts and considers valid.

The *victim* is the person, organization, or service that needs to establish whether a document carries a trusted signature. The *signer* is the party that originally signed the document and, in doing so, configured any modification-permission mechanisms.

We consider two classes of verifier: Graphical User Interface (GUI) applications and programming libraries. The distinction matters because each class exposes a different surface to the user, and our attacks succeed in different ways on each.

3.1. User Interface Verifiers

GUI verifiers show signature verification results through visual components, such as a top-bar banner and a details panel. We reuse the UI layer abstraction of [Mladenov et al. 2019] to refer to these components, restating them below.

UI-Layer 1 is the top-bar validation status. Most GUI verifiers display this banner immediately upon opening the document and use it to summarize whether all signatures in the document are valid. Owing to its prominence, it is typically the first cue a user inspects when assessing the document’s authenticity.

UI-Layer 2 carries detailed per-signature information, such as the signer’s name and certificate chain. It is generally not opened automatically, the user must open a dedicated panel before this information becomes visible.

3.2. Programming Libraries

Programming libraries expose a binary verdict and a detailed report describing every check performed for each verified signature. We collapse both into a single abstraction, the **API-Layer**, since they are returned together in the same function response and any developer consuming the library has access to both at once.

In this abstraction, the API-Layer answers two questions: whether the signature is considered valid, and whether any warning should be noted alongside the verdict. For instance, one of such warnings can be that the verifier detected an uncommon or post-signing modification but still accepts the signature as valid.

3.3. Winning Conditions

For each UI layer for GUI verifiers or the single API-Layer for library verifiers, we classify the outcome of an attack as follows [Mladenov et al. 2019]. The output is *vulnerable*

(●) when it reports the signature as valid without flagging a post-signing modification. It is *partially vulnerable* (◐) when it reports the signature as valid but raises a warning that the document was modified. It is *secure* (○) when it flags the signature as invalid.

For GUI verifiers, an attack is *successful* when both UI layers are vulnerable, and *partially successful* when UI-Layer 1 is vulnerable while UI-Layer 2 is either *partially vulnerable* or *secure*. We treat partial success as meaningful because an inattentive user may rely solely on the top-bar banner and overlook the warning surfaced in the detail panel, accepting a modified document despite the recorded discrepancy. For library verifiers, an attack is *successful* when the API-Layer is vulnerable, and *partially successful* when it is partially vulnerable; the same reasoning applies, since a developer may consume only the binary verdict and ignore the accompanying warnings.

4. Breaking Digital Signature Verifiers

Our attacks exploit two underspecified aspects of the PDF specification [ISO 2020]. By modifying a signature field’s existing entries, an attacker can place arbitrary visual content on a page, as in Figure 1, without invalidating the signature in affected verifiers.

4.1. Appearance Substitution Attack (ASA)

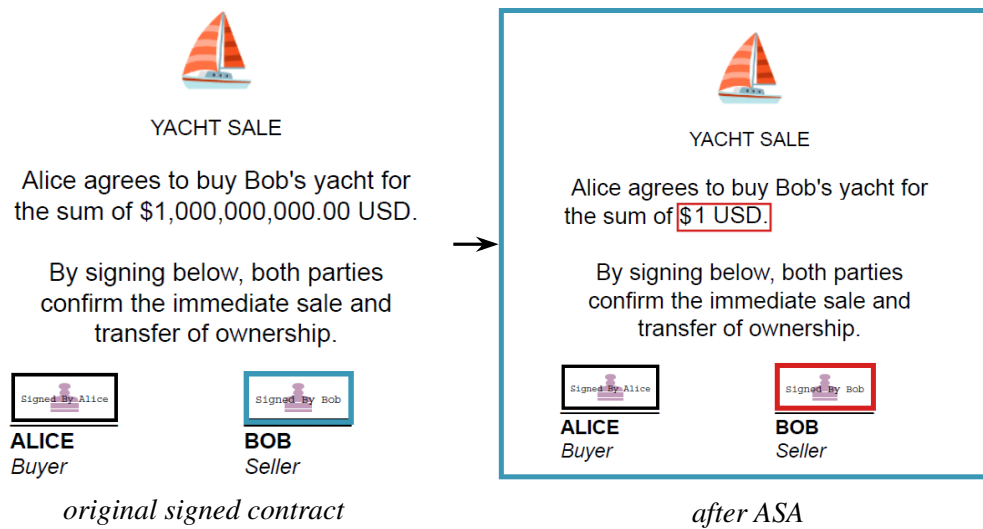
As described in Section 2.2, a signature field’s `/Rect` entry defines the region of the page occupied by the field, while its `/AP` entry defines the appearance drawn inside that region. For signatures with a visible component, `/Rect` typically identifies a small area of the page containing the signer’s name or stamp, and `/AP` stores the corresponding visual representation. For signatures without a visible component, `/Rect` commonly contains a zero-area rectangle, and `/AP` is absent.

The `/Rect` and `/AP` entries can be abused to turn a signature field into a vehicle for arbitrary visual content. Expanding `/Rect` to cover a larger region, or even the entire page, increases the area in which the field appearance can be rendered. Replacing `/AP` then changes what is drawn. As shown in Figure 4, an attacker can start from a document that already contains a visible signature, preserve the original signature appearance, and add a small malicious modification elsewhere on the page by carefully crafting the replacement appearance stream. When the existing signature is invisible, the same outcome is achieved by creating a new `/AP` and enlarging `/Rect` from its default value.

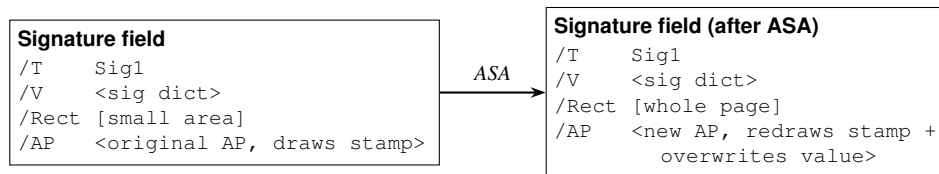
As discussed in Section 2.3, permission levels two and three of `DocMDP` and `SigFieldLock` allow form fields to be filled through an incremental update without invalidating the protected signature. Since a signature field is itself a form field, the specification does not rule out filling such a field after a prior signature has been applied. In fact, the relevant ISO wording leaves room for interpretation on what “filling” means:

[Filling in (signing) the signature field entails updating at least the `/V` entry and usually also the `/AP` entry of the associated widget annotation. (Section 12.7.5.5 of ISO 32000-2 [ISO 2020])]

The phrase “at least” suggests that filling a signature field is not necessarily limited to updating `/V` and `/AP`. Under a permissive interpretation, other field entries, including `/Rect`, could also be treated as part of the same filling operation. This ambiguity enables ASA to be applied to documents protected by modification-permission mechanisms while remaining within the class of changes that affected verifiers accept as non-invalidating.



- (a) Visual outcome. Cyan outlines mark the signature field's `/Rect`: a small region around the original stamp on the left, expanded to cover the whole page on the right. Red outlines on the right highlight the original stamp redrawn at its original location and the malicious value that overwrites \$1,000,000,000.00 with \$1.



- (b) Abstract structural change. `/T` and `/V` are preserved, while `/Rect` is enlarged to span the whole page and a new `/AP` is created that redraws the original stamp at its original location and overwrites the contracted value elsewhere on the page.

Figure 4. ASA applied to a contract signed at gov.br: (a) visual example of the attack; (b) abstract structural view of the modified signature field.

4.2. Signature Field Duplication Attack (SFDA)

A related attack exists when instead of mutating an existing signature field, the attacker introduces a *new* signature field through an incremental update and sets its `/V` entry to point at the signature dictionary of a signature already present in the document. The duplicated field is registered in the `AcroForm` dictionary alongside the original and can be placed on any page, with attacker-controlled `/Rect` and `/AP` entries. Because the bytes covered by the original signature are untouched, the cryptographic verification of that signature continues to succeed, and affected verifiers render the duplicated field as a second valid signature. An example of the resulting document is shown in Figure 5.

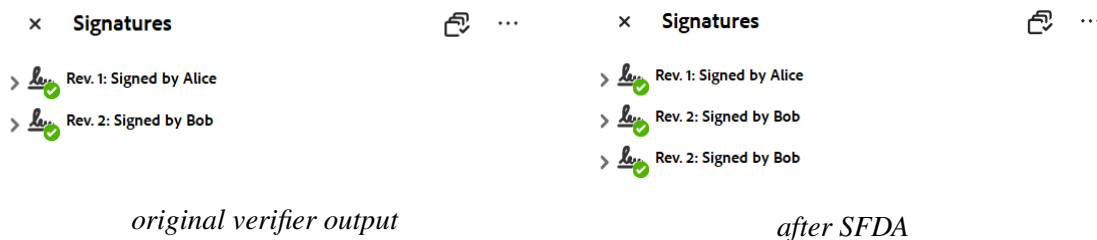
SFDA is closely related to the Sneaky Signature Attack of [Rohlmann et al. 2021] but differs in one fundamental respect: it does not require the attacker to produce a fresh signature trusted by the verifier. Whereas the Sneaky Signature Attack appends an attacker-controlled signature dictionary, SFDA reuses the legitimate one. This makes the attack particularly damaging in scenarios where obtaining a trusted certificate is impractical, such as documents signed under closed PKIs.

Because the duplicated field inherits the certificate and signer name of the original, the forgery is harder for a human reviewer to dismiss than one bearing an unfamiliar identity. The duplication itself is visually detectable, since two signers now appear in the verifier interface, but the verifier displays a known name and a familiar certificate chain, which is precisely what an inattentive reviewer is most likely to accept.

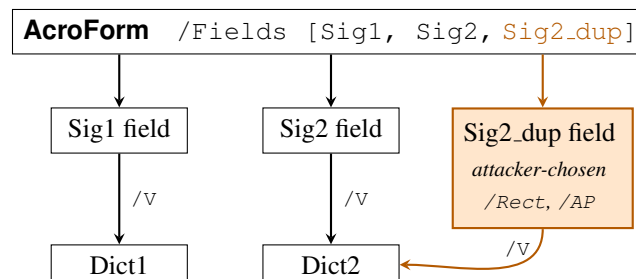
The PDF specification does not prohibit multiple signature fields from referencing the same signature dictionary. ISO 32000-2 acknowledges the configuration but does not specify how such scenario must be handled:

[If more than one location is associated with a signature, the meaning can become ambiguous. (Section 12.7.5.5 of ISO 32000-2 [ISO 2020])]

The text stops short of disallowing the configuration or prescribing how verifiers should handle it. Affected verifiers resolve the ambiguity by rendering every signature field that references the signature dictionary as valid, the attacker-controlled one included, while still reporting the document as correctly signed.



(a) Visual outcome at Layer-2. The verifier on the left correctly reports two valid signatures on a document carrying two signatures. After SFDA (right), the same verifier reports three valid signatures: the second signature has been duplicated, and the verifier accepts the duplicate as a third independent valid signature.



(b) Abstract structural change (new field and edges in orange). A new signature field `Sig2_dup` is appended to the `AcroForm`'s `/Fields` array; its `/V` entry points at `Dict2`, the same signature dictionary already referenced by `Sig2`, while its `/Rect` and `/AP` are chosen by the attacker. The field is registered on an attacker-chosen page via that page's `/Annots` array.

Figure 5. SFDA applied to a document with two valid signatures: (a) visual example of the attack; (b) abstract structural view of the duplicated signature.

4.3. Root Cause

Both passages quoted above overlook how their rules interact with incremental updates. Filling a signature field and referencing a signature dictionary are each defined as a single event, at the moment a form is completed within one static revision. The

specification never asks what should happen if a signature field is filled again in a later revision, after it was already filled once, nor what should happen if a second field is created in a later revision and pointed at a signature dictionary already established in an earlier one. ASA exploits the first gap and SFDA the second.

4.4. Limitations

ASA can only modify an existing signature field and is thus constrained by the page it is presented on. Documents that place signature fields on a dedicated signature page offer a smaller attack surface than documents where signature fields appear alongside important content. SFDA is not bound by this placement constraint, but it introduces a duplicated signature field that remains visible within UI-layer 2, so an attentive user could detect it.

Both attacks also depend on how tolerant a verifier is of post-signing modification, and on the permission configuration chosen when the document was signed. Some verifiers reject both attacks outright by treating any post-signing change, including benign ones, as invalidating, while others only reject them if a protection mechanism is explicitly configured, or if the modified or duplicated field is moved to another page (Section 6).

Because both attacks abuse visual components of interactive fields, their rendering can vary across readers, which may highlight the field, make it clickable, or block text selection. Such artifacts can expose the attack to an attentive user, but prior work shows users still perform security-sensitive actions when inconsistencies are small [Anliker et al. 2025]; in our setting, they may be dismissed as reader quirks, especially when all signatures are reported valid. Prior PDF-signature attacks left similar subtle inconsistencies [Rohlmann et al. 2021].

5. Methodology

We evaluate 18 verifiers spanning three categories against the Appearance Substitution Attack (ASA) and the Signature Field Duplication Attack (SFDA), classifying each outcome under the criteria of Section 3.3. The four GUI verifiers are Adobe Acrobat [Adobe Inc. 2026b], Foxit PDF Reader [Foxit Software Inc. 2026], Master PDF Editor [Code Industry Ltd. 2026], and Microsoft Edge [Microsoft Corporation 2026]. The two library verifiers are pyHanko [Valvekens 2026] and the European Commission’s Digital Signature Services (DSS) [European Commission 2026a]. The twelve web verifiers are the DSS Demonstration WebApp [European Commission 2026b], the ITI’s signature validator [ITI 2026], Certisign [Certisign 2026], DigitalSign [DigitalSign 2026], CenturySign [CenturySign 2026], the Finnish DVV validator [DPDSA 2026], BRY Verifica [BRY Tecnologia 2026], ZapSign [ZapSign 2026], Clicksign [Clicksign 2026], ON-RCPN’s Verify Signature [ON-RCPN 2026], RNP’s Digital Signature Verifier [RNP 2026], and UFSC’s Digital Signature Verifier [UFSC 2026].

We use two testing strategies. For library verifiers and GUI desktop verifiers, where the trust anchors can be freely chosen, an automated tester (see Section 5.1) evaluates every verifier against many possible attack configurations. For GUI web verifiers, whose trusted certificate lists are fixed by the creator and signatures with arbitrary parameters cannot be easily created, we perform a manual evaluation in which a single document carrying a signature accepted by the verifier is subjected to each attack. In both cases, we record the UI-Layer 1 and UI-Layer 2 outcomes for GUI verifiers, or the API-Layer

outcome for library verifiers, and label each as *vulnerable*, *partially vulnerable*, or *secure*. Claude and Gemini were used for aiding on the implementation of the automated testing tool, the detection tool and for the production of figures within this paper.

5.1. Automated Tester

The automated tester is composed of two execution flows: test-case generation and test execution. Test-case generation enumerates configurations of the trusted signature and, for each configuration, produces one PDF carrying the corresponding ASA attack and one PDF carrying the corresponding SFDA attack. Each generated file is persisted, in a local database for use by later test execution runs¹. Each signature uses a self-issued certificate provided by the automated tester.

For library verifiers, each test case is submitted to the relevant verification function and the returned verdict and modification warnings are stored, populating the API-Layer. For GUI desktop verifiers, the tester follows the screenshot-based approach of previous works [Rohlmann et al. 2021]. First, it opens the document in the verifier, then captures one screenshot per UI layer being evaluated, and lastly classifies each screenshot by template matching against a small reference-image set per verifier and layer, one set per outcome class (*valid*, *warn*, *invalid*). The need for a reference-set exists because some verifiers use multiple icons when reporting whether a signature is valid or invalid; for example, Adobe Reader[Adobe Inc. 2026b] and Foxit[Foxit Software Inc. 2026] use different icons to distinguish between valid certification signatures and valid signatures. The *warn* class captures the partial-vulnerability case described on Section 3.3, in which the verifier accepts the signature but raises a warning that the document was modified after signing.

To ensure reproducibility, all operations performed by the automated tester run inside a Docker environment. While library verifiers use a simple Docker image, GUI desktop verifiers rely on a virtual machine that runs the desktop session and automatically adds the self-issued certificates to the OS trust store. Although the script cannot download specific versions of every GUI desktop verifier due to licensing restrictions, a warning is displayed whenever a version mismatch occurs, making it explicit that the results may differ from those reported in the paper. The template-matching classifier can be error-prone, since the templates were generated on a different computer; to mitigate this, the tester ensures that the virtual machine uses the same resolution as the one used when the templates were created, and applies a high similarity threshold to reduce the chance of false positives. If no template is matched, the tester considers the test case invalid and asks the user to regenerate the templates. The self-issued certificate used to produce trusted signatures for the automated tester uses a 2048-bit RSA key and a SHA-256 signature.

6. Evaluation

All results in Table 1 are tied to the specific versions listed and reflect the state of each verifier at the time of testing. Vendors may patch the underlying issue in later releases, following the disclosure notifications described in Section 9. We therefore treat these results as a snapshot rather than a permanent characterization of each product.

¹The source code of both tools can be accessed at: <https://doi.org/10.5281/zenodo.21782014>

Table 1. Per-verifier results of how each verifier handles ASA and SFDA. ● means a verifier is fully *vulnerable* and does not warn that modifications were made despite the signature being valid. ◐ means a verifier is *partially vulnerable* and warns that a modification occurred. ○ means a verifier is *secure* and correctly invalidates the signature. Empty cells are layers that do not apply to the verifier’s class or versions not exposed by the verifier.

Application ²	Version	Category	UI-Layer 1		UI-Layer 2		API-Layer	
			ASA	SFDA	ASA	SFDA	ASA	SFDA
Adobe Acrobat	26.1.21662.0	desktop	◐ ●		◐ ●			
Foxit PDF Reader	2026.1.2.36540	desktop	● ●		◐ ○			
Master PDF Editor	5.9.9.8	desktop	● ●		● ●			
Microsoft Edge	150.0.4078.105	desktop	● ●		● ●			
pyHanko	0.32.1	library					○ ●	
DSS	6.4	library					● ○	
DSS Demo WebApp	6.4	web	◐ ○		◐ ○			
ITI	6aec769	web	● ○		● ○			
Certisign		web	● ●		● ●			
DigitalSign		web	◐ ○		◐ ○			
DVV		web	◐ ○		◐ ○			
BRY Verifica		web	● ◐		● ◐			
ZapSign		web	○ ○		○ ○			
ON-RCPN		web	○ ○		○ ○			
CenturySign		web	○ ○		○ ○			
Clicksign		web	● ●		● ●			
RNP		web	● ○		● ○			
UFSC		web	● ●		● ●			
ASA, vulnerable or partial			13/16		13/16		1/2	
SFDA, vulnerable or partial			8/16		7/16		1/2	

Our evaluation covers 12 web-based, 4 desktop, and 2 library verifiers. Desktop and library verifiers, which support automation, were each evaluated against 112 crafted PDF documents; web verifiers were evaluated manually with a single document carrying a signature trusted by each verifier. Desktop tests ran on Windows 11, web tests on Windows 11 with Microsoft Edge, and library tests on NixOS 26. Signatures for the automated tests were produced by a custom pyHanko script using a self-issued certificate.

As Table 1 shows, the majority of verifiers are vulnerable to ASA at **UI-Layer 1** or **API-Layer**, the layers users rely on to judge whether a signature is valid. Except for Foxit, every GUI verifier vulnerable to ASA at **UI-Layer 1** is also vulnerable at **UI-Layer 2**, so the detail panel rarely catches the attack even when the user opens it. By the criteria of Section 3.3, this yields 8 *successful* attacks and 5 *partially successful* ones.

SFDA reaches slightly fewer verifiers than ASA, with almost half of the GUI and library verifiers reporting the signature as valid in some form at **UI-Layer 1** or the **API-Layer**. As with ASA, Foxit is the only GUI verifier whose verdict downgrades between UI layers, going from fully vulnerable at **UI-Layer 1** to *secure* at **UI-Layer 2**. Overall, SFDA yields 6 *successful* attacks and 2 *partially successful* ones.

²Application, product or company name. All verifiers were used as is, with no configuration besides the bare minimum needed for ensuring the self-issued certificates are considered trusted.

Foxit’s downgrade between UI layers reflects how it renders **UI-Layer 1**. Inspecting the automated test outputs, we observed that even when the underlying checks detect post-signing modifications, the top bar shows only a green “certified document” icon and no further detail. The warnings driving the partial **UI-Layer 2** verdict for ASA, and the *secure* verdict for SFDA, only appear once the user opens the per-signature detail panel.

The page-change variants of the two attacks behave asymmetrically. Every verifier vulnerable to SFDA at **UI-Layer 1** remains vulnerable when the duplicated signature field is placed on a different page. This is arguably the stronger variant, since it lets the attacker stage the forged appearance over arbitrary content on any page. The same does not hold for ASA, where DSS, ITI, BRY, and DigitalSign all reject the attack once the modified signature field is relocated to a different page.

The pyHanko library is vulnerable to SFDA only when neither `DocMDP` nor `SigFieldLock` is present, and rejects both attacks otherwise. DSS exhibits a similar behavior for ASA, accepting the attack only when the modified signature field originally defined an invisible signature. Other verifiers built on DSS, such as the DSS Demo WebApp and DVV, are plausibly vulnerable in the same configuration, but we could not confirm it, as no documents carrying an invisible signature trusted by these verifiers were available. ZapSign rejects both attacks, but does so by treating any post-signing modification as invalidating, including benign ones. In our tests, even appending a comment or a new revision to a signed document was enough to invalidate the signature.

7. Countermeasures

We propose countermeasures for both attacks at three levels: the specification, individual verifiers, and external detection tooling. As outlined in Section 4, both attacks exploit ambiguities in the PDF specification that allow a document’s visual content to be modified without invalidating the signature. The specification can close both gaps. To prevent ASA, ISO 32000-2 should require that any modification to the `/AP` or `/Rect` of a filled signature field invalidates the protecting signature, regardless of the prevailing `DocMDP` or `SigFieldLock` permission level. To prevent SFDA, require that no two signature fields reference the same signature dictionary, or, as a weaker but sufficient alternative, all fields referencing a signature dictionary already exist when the dictionary is created.

Individual verifier implementations can adopt both of the above rules without waiting for the specification to change. This precedent already exists since many of the attacks reported in prior work [Rohlmann et al. 2021, Mainka et al. 2021, Müller et al. 2021, Mladenov et al. 2019, Müller et al. 2019] were addressed by individual verifier vendors rather than by amendments to ISO 32000. Verifier-level mitigation, however, does not scale; every new verifier implementation can reintroduce the same class of vulnerability present in old versions of more mature verifiers.

We provide a Python tool that takes a PDF as input and produces a textual report indicating whether either attack is present. When an attack is detected, the report identifies the affected signature fields and the offending PDF objects. ASA is detected by comparing each signature field as it appears in the revision that signed it against its appearance in the latest revision; if `/AP` or `/Rect` differ, the document is flagged. SFDA is detected by scanning the latest revision for pairs of signature fields that reference the same signature dictionary and verifying, for each pair, whether at least one of the fields was introduced

in an incremental update later than the dictionary it references.

The script is, however, prone to false positives. If an user genuinely modified a signature field after it was signed, such modification would wrongly be flagged as an instance of ASA even if no visual modification to the document was made. Furthermore, some signature producers might intentionally attach multiple signature fields to the same signature dictionary as a way of introducing visual components - say, a rectangle with text "Signed By Bob" - on multiple pages of the document simultaneously; such modification would be flagged as an instance of SFDA even if their origin was legitimate. Given both kinds of false positives use the same mechanisms that our attacks use, fixing our attacks implies considering such modifications invalid; which, albeit less than ideal, is still a better scenario than the one observed today.

8. Related Works

One Trillion Dollar Refund [Mladenov et al. 2019] presented three attacks against approval-signature verifiers. The Universal Signature Forgery (USF) changes the signature dictionary so the verifier cannot find what it needs but still reports the signature as valid. The Incremental Saving Attack (ISA) appends a broken incremental update that adds visual modification. The Signature Wrapping Attack (SWA) moves the signed bytes elsewhere and puts new content in place.

Subsequent work introduced shadow attacks [Mainka et al. 2021], building on top of [Mladenov et al. 2019], which use spec-compliant incremental updates instead of broken ones. Their attack plants hidden content in the document before it is signed and reveals it afterwards through normal incremental updates.

In contrast, later work examined certification signatures and identified two attacks [Rohlmann et al. 2021]. The Evil Annotation Attack (EAA) adds annotations that hide or replace certified content. The Sneaky Signature Attack (SSA) adds a new signature whose appearance covers attacker-controlled content. Both attacks abuse changes that DocMDP allows. SSA also requires the attacker to get a fresh certificate trusted by the verifier, since the new signature has to be considered valid.

Unlike USF, ISA, and SWA, ASA uses solely incremental updates fully within the spec. Unlike the shadow attacks, ASA needs no access to the document before signing and can be applied to any already-signed PDF. The closest match is EAA, which also covers the page with attacker-chosen content, but through a new annotation governed by DocMDP/SigFieldLock annotation permissions, ASA instead changes the signature field's own /AP and /Rect, which affected verifiers treat as part of filling the field, letting it work at lower permission levels than EAA.

SFDA also works only on already-signed PDFs and follows the spec. The closest match is SSA, since both add a new signature field the verifier shows as valid, but SSA brings its own signature dictionary, forcing the attacker to get a trusted certificate, while SFDA points the new field at a signature dictionary already present in the document. This widens the threat to settings where a trusted certificate is hard to get, such as documents signed under closed PKIs, and the duplicated field inherits the original signer's name and certificate chain, making the forgery harder to dismiss than SSA's unfamiliar identity. Table 2 summarizes the attacks discussed above along the dimensions that separate them.

Table 2. Comparison of PDF signature-forgery attacks.

Attack	Spec-comp.	Pre-sign	Own cert.	MDP 2	Targets
USF [Mladenov et al. 2019]	No	No	No	No	Signature dictionary
ISA [Mladenov et al. 2019]	No	No	No	No	Xref table / trailer
SWA [Mladenov et al. 2019]	No	No	No	No	Signed byte range
Shadow [Mainka et al. 2021]	Yes	Yes	No	No	Hidden content
EAA [Rohlmann et al. 2021]	Yes	No	No	No	Annotations
SSA [Rohlmann et al. 2021]	Yes	No	Yes	Yes	New signature field
ASA (ours)	Yes	No	No	Yes	Sig. field (/AP, /Rect)
SFDA (ours)	Yes	No	No	Yes	Sig. field duplication

9. Ethical Considerations and Responsible Disclosure

This paper demonstrates practical attacks against 17 PDF signature verifiers, several of which are commercial products in wide use. We followed a coordinated disclosure process to notify affected vendors before these findings were made public.

We notified the vendor or maintainer of every verifier found vulnerable or partially vulnerable to ASA or SFDA under the criteria presented in Section 3.3, and worked with each toward a fix. We followed a 30-day disclosure window per vendor, counted from the date of notification. Not every vendor has shipped a fix as of the writing of this manuscript, and thus we are unable to publish a complete table of fixed versions; we will publish a list of the affected verifiers and their fixed versions³. The list will be updated when a vendor publishes a new version in which the vulnerability is fixed.

We released the detection tool described in Section 7 in full⁴, including both the code for analyzing verifiers’ verdicts and the scripts used for generating the exploits.

10. Conclusion

PDF signatures provide a standardized and consistent way for someone to sign a document at a specific instant of time. However, visual modifications can be made by incrementally updating the document in a way that keeps digital signatures valid and triggers no modification warning. In this paper, we introduced two novel attacks that abuse flaws in the specification, the Appearance Substitution Attack (ASA) and the Signature Field Duplication Attack (SFDA). Although neither modifies the page directly, both abuse the appearance mechanism of interactive forms to overlay visual changes on top of the page, and even when detectable, the result is subtle enough that most users would not hesitate to trust the signature, since affected verifiers still report it as valid and do not flag the modifications. We then evaluated these attacks against many verifiers and found that most were vulnerable in one way or another.

Although we proposed countermeasures to mitigate the damage such attacks can cause, both this work and previous attacks can be seen as proof that the large complexity of the PDF specification, coupled with the decentralized ecosystem where each verifier

³The list of fixed verifiers can be accessed at: <https://doi.org/10.5281/zenodo.21782139>

⁴The source code of the detection tool can be accessed at: <https://doi.org/10.5281/zenodo.21782014>

implements its own algorithm and interprets the ISO in its own way, creates fertile ground for attacks of this kind. In fact, one could argue that incremental updates, which greatly complicate signature validation, are dangerous features, given that both the attacks introduced here and more than half of those in Section 8 abuse incremental updates in one way or another. As future work, it would be valuable to define a standardized verification algorithm on which all verifiers could base their implementations.

Acknowledgements

This research was partially financed by the Operador Nacional do Registro Civil de Pessoas Naturais (ON-RCPN).

References

- Adobe Inc. (2026a). Acrobat Sign: Electronic and digital signatures for business. <https://www.adobe.com/acrobat/business/sign.html>.
- Adobe Inc. (2026b). Adobe Acrobat Reader DC. <https://www.adobe.com/acrobat/pdf-reader.html>. Accessed on 2026-05-10.
- Anliker, C., Lain, D., and Čapkun, S. (2025). Phishing attacks against password manager browser extensions. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA. USENIX Association.
- Brazil (2020). Lei nº 14.063, de 23 de setembro de 2020.
- BRY Tecnologia (2026). BRY Verifica. <https://verifica.bry.com.br/>. Accessed on 2026-05-24.
- CenturySign (2026). CenturySign. Unreleased product.
- Certisign (2026). Certisign Verificador de Assinaturas. <https://assinaturas.certisign.com.br/VerificadorAssinaturas/Verificador>. Accessed on 2026-05-24.
- Clicksign (2026). Clicksign Validador. <https://www.clicksign.com/validador>. Accessed on 2026-05-24.
- Code Industry Ltd. (2026). Master PDF editor. <https://code-industry.net/masterpdfeditor/>. Accessed on 2026-05-24.
- DigitalSign (2026). DigitalSign Signatures Validator. <https://verify.digitalsign.pt/signaturesValidator/validator/validation>. Accessed on 2026-05-24.
- DPDSA (2026). Validation of Electronically Signed PDF Documents. <https://dvv.fineid.fi/en/validation>. Accessed on 2026-05-24.
- EU (2014). Regulation (EU) No 910/2014. Official Journal of the European Union.
- European Commission (2026a). Digital Signature Services (DSS). <https://github.com/esig/dss>. Accessed on 2026-05-24.
- European Commission (2026b). DSS Demonstration WebApp: Signature Validation. <https://ec.europa.eu/digital-building-blocks/DSS/webapp-demo/validation>. Accessed on 2026-05-24.

- Foxit Software Inc. (2026). Foxit PDF reader. <https://www.foxit.com/pdf-reader/>. Accessed on 2026-05-10.
- ISO (2020). Document management — Portable document format — Part 2: PDF 2.0 . Standard ISO 32000-2:2020, International Organization for Standardization, Geneva, Switzerland.
- ITI (2026). Validador de assinaturas ITI. <https://validar.iti.gov.br/>. Accessed on 2026-05-24.
- Mainka, C., Mladenov, V., and Rohlmann, S. (2021). Shadow attacks: Hiding and replacing content in signed pdfs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
- Microsoft Corporation (2026). Microsoft Edge. <https://www.microsoft.com/edge>. Accessed on 2026-05-24.
- Mladenov, V., Mainka, C., Meyer zu Selhausen, K., Grothe, M., and Schwenk, J. (2019). 1 trillion dollar refund: How to spoof pdf signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 1–14, New York, NY, USA. Association for Computing Machinery.
- Müller, J., Ising, F., Mladenov, V., Mainka, C., Schinzel, S., and Schwenk, J. (2019). Practical decryption exfiltration: Breaking pdf encryption. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 15–29, New York, NY, USA. Association for Computing Machinery.
- Müller, J., Noss, D., Mainka, C., Mladenov, V., and Schwenk, J. (2021). Processing dangerous paths: On security and privacy of the portable document format. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
- ON-RCPN (2026). ON-RCPN. <https://assinatura.registrocivil.org.br/verificar>. Accessed on 2026-08-03.
- RNP (2026). RNP verificador de assinaturas. <https://verificador.rnp.br/home>. Accessed on 2026-08-03.
- Rohlmann, S., Mladenov, V., Mainka, C., and Schwenk, J. (2021). Breaking the specification: Pdf certification. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1485–1501.
- The Document Foundation (2026). LibreOffice. <https://www.libreoffice.org/>. Accessed on 2026-05-10.
- UFSC (2026). UFSC verificador de assinaturas. <https://assina.ufsc.br/assinatura/index.xhtml>. Accessed on 2026-08-04.
- U.S. Congress (2000). Electronic Signatures in Global and National Commerce Act. Public Law 106–229, 114 Stat. 464, 15 U.S.C. ch. 96.
- Valvekens, M. (2026). pyHanko: PDF signing and validation library for Python. <https://github.com/MatthiasValvekens/pyHanko>. Accessed on 2026-05-24.
- ZapSign (2026). ZapSign Validador. <https://validador.zapsign.com.br/validate-pdf>. Accessed on 2026-05-24.