Forensic Science International: Digital Investigation 56 (2026) 302054

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2026 - Selected Papers from the 13th Annual Digital Forensics Research Conference Europe

CPR: Corrupted PDF recovery algorithm for digital forensic investigations

Seoyoung Kim [a], Yunji Park [a], Woobeen Park [b], Doowon Jeong [a,*]

[a] *Department of Forensic Sciences, Sungkyunkwan University, Seoul, South Korea*
[b] *Korea Internet & Security Agency, Seoul, South Korea*

ARTICLE INFO

Keywords:
Digital forensics
Portable document format (PDF)
Corrupted document
File recovery
Data extraction

ABSTRACT

As digital documents have become the dominant medium for information exchange, PDF has emerged as a standard format and a crucial source of evidence in digital forensic investigations. However, PDFs are internally organized as reference-based object structures whose interdependencies make recovery from corruption particularly challenging. Moreover, variations in encoding and storage—stemming from different producer tools—further complicate forensic analysis and reconstruction. This paper presents a comprehensive byte-level forensic analysis of the PDF structure and characterizes content-generation patterns across multiple producer types. Focusing on text data, we classify character storage within Content Objects into three categories—Text, XObject, and Path—and systematically analyze structural differences by generation method. Building on these insights, we propose CPR (Corrupted PDF Recovery), an algorithm designed to restore content from partially damaged PDFs. CPR carves objects from raw bytes, reconstructs inter-object relationships, and dynamically adapts its recovery process to the file's generation characteristics. For text restoration, CPR leverages a font mapping database (FontDB) and employs a large language model (LLM) to validate recovered outputs. Evaluation on a multilingual dataset encompassing three languages and multiple corruption scenarios demonstrates CPR's superiority over existing tools, achieving approximately 166 % higher recovery rate and greater forensic completeness, even when only a single content object exists. The CPR implementation, dataset, and FontDB are openly released as open source to support reproducibility and further forensic research.

1. Introduction

Digital documents have become deeply integrated into daily life and business operations, serving as key evidence in various criminal investigations. Among them, the Portable Document Format (PDF) has become one of the most prevalent digital document types due to its high compatibility across diverse operating systems and applications, making it an important target of analysis in digital forensic examinations.

The PDF format is highly extensible and can be generated in diverse ways across different applications. Consequently, the internal structure used to store content varies depending on the generation method. For instance, text may be encoded with proprietary character mappings, converted into images, or represented as vector graphics. Even within a single document, different fonts may employ distinct encoding schemes. Moreover, because PDF relies on a reference-based architecture in which numerous internal objects point to one another, multiple stages of parsing are required to accurately interpret the embedded data.

These characteristics present critical challenges in the recovery of corrupted PDF documents. Because content can be stored in multiple formats, it is essential to determine how each damaged fragment was originally generated and to apply recovery algorithms tailored to each type. Moreover, due to the intricate cross-reference structure among objects, even partial corruption can render an entire document inaccessible. Therefore, precise byte-level analysis of the file format is required to enable effective and reliable recovery.

However, existing PDF recovery tools do not fully account for the diversity of content storage methods, which limits their capability to handle different corruption scenarios. Moreover, when specific objects are damaged, these tools often fail to reconstruct the missing object relationships, resulting in recovery failure even when parsable content remains intact.

To overcome these limitations, we propose a novel recovery algorithm, CPR (Corrupted PDF Recovery), designed to address the major technical challenges in recovering corrupted PDF documents. Specifically, we identify and define three types of unrecoverable damage that existing tools fail to handle.

* Corresponding author.
E-mail addresses: pangkz@skku.edu (S. Kim), yun.jiggle@g.skku.edu (Y. Park), beeny110@kisa.or.kr (W. Park), doowon@skku.edu (D. Jeong).

https://doi.org/10.1016/j.fsidi.2026.302054

2666-2817/© 2026 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Files that are completely unreadable
2. Documents that display as blank pages
3. Text that appears garbled or unreadable due to encoding corruption

For each category, we propose a corresponding recovery strategy. To this end, we systematically analyze the internal content storage structure of PDFs and identify structural variations across different generation methods. Based on these insights, we develop an integrated recovery algorithm that maximizes the utilization of residual data and adapts to producer-specific characteristics, thereby improving both the recovery rate and overall accuracy. In particular, we enhance text restoration performance under font corruption or encoding failure by combining a Font Mapping Database (FontDB) with a Large Language Model (LLM).

The remainder of this paper is organized as follows. Section 2 reviews related research. Section 3 analyzes the internal structure of PDF files and producer-specific characteristics from a forensic perspective. Section 4 describes the design and implementation of the proposed CPR algorithm, while Section 5 presents the experimental results and comparative analysis with existing recovery tools. Finally, Section 6 concludes the paper and outlines directions for future work.

2. Related works

Existing research on PDF forensics has primarily focused on document structure analysis for malware detection and forensic authentication (Maiorca and Biggio, 2019; Zeng and Qiu, 2022; Sakhawat Hossain et al., 2024). These studies employ PDF parsing tools to extract and analyze structural elements such as headers, bodies, and cross-reference tables, along with embedded objects.

Research directly addressing the recovery of corrupted PDF documents remains notably limited. Krutko et al. (Mark et al., 2018) proposed an automated font encoding recovery system that combines CNN-based symbol recognition with language models. The system detects corrupted fonts, performs OCR on individual glyphs, which are graphical representations of characters, to classify them into homoglyph groups, and reconstructs glyph-to-character mappings by integrating positional information with linguistic context. Evaluation on multilingual PDF datasets demonstrated error rates 53–77 % lower than those of Tesseract, with average processing speeds approximately 30 times faster. Furthermore, Stefanovitch (Nicolas, 2022) introduced a text recovery approach for documents written in endangered languages. The method requires users to manually input visually identifiable portions of text from the document, after which the system incrementally constructs a translation map by aligning Unicode sequences with corresponding CID (Character ID) sequences in the PDF. Experiments using the UDHR corpus showed that complete recovery could be achieved by providing only 4–5 % of the total text.

However, these prior studies exhibit several limitations. First, they do not provide a comprehensive definition of PDF corruption. Both studies restrict the notion of a damaged PDF to cases where copy-and-paste operations fail to produce the expected text output or yield garbled characters upon text extraction, implicitly assuming that the document remains visually displayable and printable. As a result, more severe forms of corruption—such as complete file inaccessibility, blank page rendering, or total encoding failure—are not addressed. Second, these studies primarily focus on mapping inconsistencies between CIDs and Unicode. However, text within PDFs can be stored in diverse forms depending on the language and encoding method, including `Text Objects`, `XObjects`, and `Path Objects`. Existing research does not sufficiently account for this diversity in content storage structures.

3. Forensic analysis on PDF structure

The PDF standard specification (Document management, 2020) defines a unified structure for representing document data. However, the internal composition of real-world PDF files often differs in implementation details depending on the software used for generation. Such variations can significantly affect the accuracy of data recovery and therefore warrant analysis from a digital forensic perspective. This section examines the internal structure of PDF files from a forensic viewpoint. Section 3.1 investigates the relationships among key objects that contain information essential for content recovery, while Section 3.2 compares structural differences across various PDF generation methods to identify critical factors relevant to the recovery process.

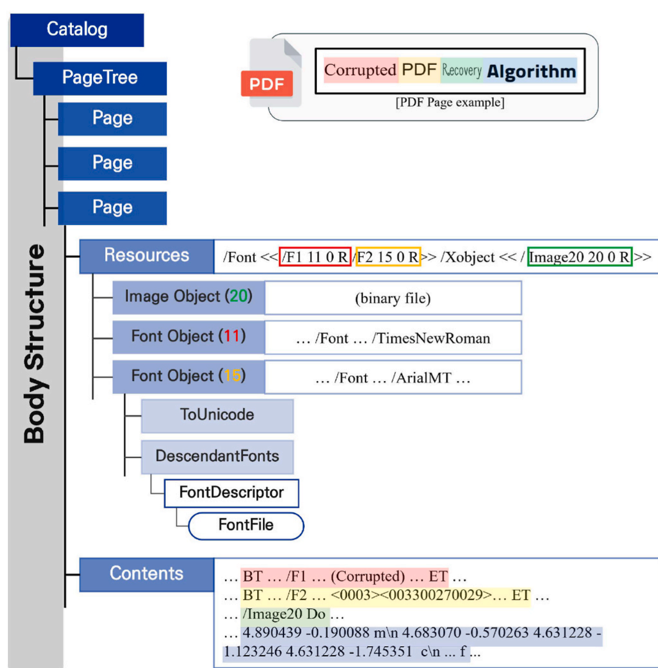

Fig. 1. Example of PDF Body structure with mixed content (Text/XObject/Path).

3.1. PDF content structure and object relationships

A PDF file consists of four structural components: `Header`, `Body`, `XRef` (Cross-Reference Table), and `Trailer`. The `Body` contains the actual content—such as text and images—that composes each page of the document, with data recorded in units of objects. Each object begins with a unique ID and a `obj` header signature and terminates with an `endobj` footer signature. A single object may contain multiple nested objects, and when multiple objects are embedded or when the data within a single object is large, they are typically stored in compressed form.

As illustrated in Fig. 1, the `Body` is hierarchically organized, with a root `Catalog Object` followed by `PageTree Object` that links multiple `Page Objects`. Each `Page Object` represents a single page and references the associated `Resources` and `Content Objects`, which define the page's text, images, and graphics.

Because the `Page Objects`, manage the primary content representation and may encode characters in various forms even within a single page, understanding their internal structure is essential. Fig. 1 shows an example of how the string “Corrupted PDF Recovery Algorithm” is stored when entered in a word processor and converted to PDF.

3.1.1. Resources Object

The `Resources Object` manages external resources referenced by content objects within a page. It contains reference data for reusable objects—such as fonts, images, and graphics—that are independently defined within the PDF. Each resource is linked to the page's content through a unique tag.

Table 1
Characteristics by *Generation Method.*

PDF *Generation Method*	FontTag Scope	CID Format	CID Source	Content Object Tag	Object Num Pattern	Object Data Pattern
macOS Document Editor	Global	()	Internal	q Q q/Cs1 cs 0 0 0	3–1–4	–
MS Save as	Global	(), <>	GlyphID	/Span,/Lang, EMC	1–2–3	Catalog – PageTree
MS Print to PDF	Per Page	<>	GlyphID	0.750000 0 0–0.750000 0	4–5–8	(Identity) – (Adobe)
Adobe Acrobat	Per Page	(), <>	GlyphID Internal	/P <</MCID	–	<</Linearized – <</DecodeParms <</Metadata – <</Length

All information recorded in the `Resources Object` is organized in a dictionary structure. The font-related tag dictionary defines mappings to individual `Font Objects` through font tags such as `/F1` and `/F2`, as illustrated in Fig. 1.

Likewise, `XObjects` that reference images or vector graphics establish connections between tags and their corresponding `XObjects` through an `XObject` tag dictionary. For example, Fig. 1 shows that the `/Image20` tag references object 20, an `Image Object` representing an individual `XObject`.

3.1.2. *Font Object*

A `Font Object` represents the actual typeface referenced by a font tag defined in the `Resources Object`. For example, in Fig. 1, the `/F1` font tag corresponds to a `Font Object` representing the *TimesNewRoman* font. `Font Objects` are critical for text recovery because they define not only glyph appearance but also character encoding and mapping.

Each `Font Object` is associated with several sub-objects, including `ToUnicode`, `DescendantFonts`, `FontDescriptor`, and `FontFile`, which collectively determine character mapping and rendering behavior. PDF text is encoded using CIDs (Character IDs), an Adobe-defined scheme designed to support multilingual and composite fonts. Accordingly, `Content Objects` store text as CIDs rather than Unicode characters.

To render text, PDF viewers rely on CMaps that map CID to Unicode. This mapping is essential for content recovery, as external or embedded fonts cannot be correctly interpreted when CMap information is damaged. CMaps are stored in `ToUnicode` and `FontFile Objects`. The `ToUnicode Object` contains the CMap dictionary used within the document, while the `FontFile Object` includes subsets of the font file, such as glyph data and CMaps. In CID-based fonts, the `DescendantFonts Object` references a `FontDescriptor`, which defines font attributes and links to the corresponding `FontFile`.

Since CMaps are defined per font, the same Unicode character may be assigned different CID values across fonts. Therefore, from a forensic perspective, content recovery requires analyzing each font's CMap to accurately identify the correspondence between CIDs and Unicode.

3.1.3. *Content object*

Although commonly referred to as `Content Streams`, this study uses the term `Content Object` to distinguish it from other PDF objects. The `Content Object` defines the concrete visual output of a PDF page, containing all displayed elements such as text, graphics, and images. It primarily consists of `Text Objects`, `XObjects`, and `Path Objects`, each delimited by specific PDF operators.

3.1.3.1. Text Object. A `Text Object` is enclosed between the “BT” (Begin Text) and “ET” (End Text) operators and stores textual content as font tag-CID pairs. Font tags are defined in the `Resources Object`, while CIDs are mapped to Unicode through the font's CMap. CIDs are expressed either in “( )” or “< >” format: the former uses direct byte-level encoding, whereas the latter represents CIDs as hexadecimal values that must be resolved via CMaps. For example, in Fig. 1, “Corrupted” uses CIDs in “( )” format, stored as “(Corrupted)” with byte-level encoding. In contrast, “PDF” uses hexadecimal CIDs such as “<003300270029>”, representing “0033, 0027, 0029”, which are mapped through the CMap within the `Font Object` corresponding to “ArialMT” referenced by the font tag “/F2”.

3.1.3.2. XObject. An `XObject` is invoked using an `XObject` tag and the `Do` operator (e.g.,/Image20 Do) to render images or graphic elements. `XObjects` can be reused across multiple locations within a document. In some cases, textual content may also be stored as image-based `XObjects` and rendered via tag invocation, as illustrated by “Recovery” in Fig. 1.

3.1.3.3. Path Object. `Path Object` defines its beginning with operators such as “m” (moveto) and “re” (rectangle), and its end with path-painting operators such as stroke (*S*, *s*), fill (*f*, *F*, *f**), or fill + stroke (*B*, *B**, *b*, *b**). While it primarily represents graphic elements such as shapes or tables, certain external fonts directly draw character shapes as Paths to represent text, such as “Algorithm” in Fig. 1. A `Path Object` consists of combinations of coordinate values and path operators, as shown in the blue section of `Content Object` in Fig. 1.

3.2. *Structure characteristics by PDF generation method*

PDFs exhibit different internal structures depending on the generation method, including object hierarchy, encoding, and stream structure. This study analyzed the internal structure of six major PDF generation methods that are most widely used: (1) *Microsoft Save as (PDF)*, (2) *MS Print to PDF*, and (3) *Adobe Acrobat PDF Maker* in the Windows environment, and (4) *Adobe Acrobat PDF Maker*, (5) *Document Editor - Save As PDF*, and (6) *Document Editor - Mac Print to PDF* in the macOS environment. The comprehensive analysis results are presented in Table 1. Based on our analysis, the PDF structures produced by the macOS Document Editor–based methods and by Adobe Acrobat were found to be equivalent; therefore, these methods are reported as two unified categories in Table 1.

3.2.1. *macOS - document editor*

PDFs generated using the macOS document editor have font tags that reference the same font across all pages. CIDs are expressed in the “( )” format and are parsed in two ways. First, characters are directly recorded. For example, “(!)” is interpreted as CID 21, corresponding to “!” at Unicode U+0021. Second, hexadecimal escape sequences are used. “(\x21)” corresponds to byte value 0x21, mapping to CID 21. Both methods are converted to Unicode through CMaps. Additionally, CIDs may be sequentially assigned starting from 21 according to their order of appearance in the `Content Object`. As a result, Unicode restoration becomes challenging when the corresponding CMaps are damaged. The `Content Object` begins with the command “q Q q/Cs1 cs 0 0 0”, and object numbers are assigned in the sequence “3–1–4”, which reflects the order in which the objects first appear in the PDF.

3.2.2. *MS save as*

Font tags reference the same font across all pages, and CIDs are expressed in two formats: “( )” and “< >“. The “( )” format directly uses

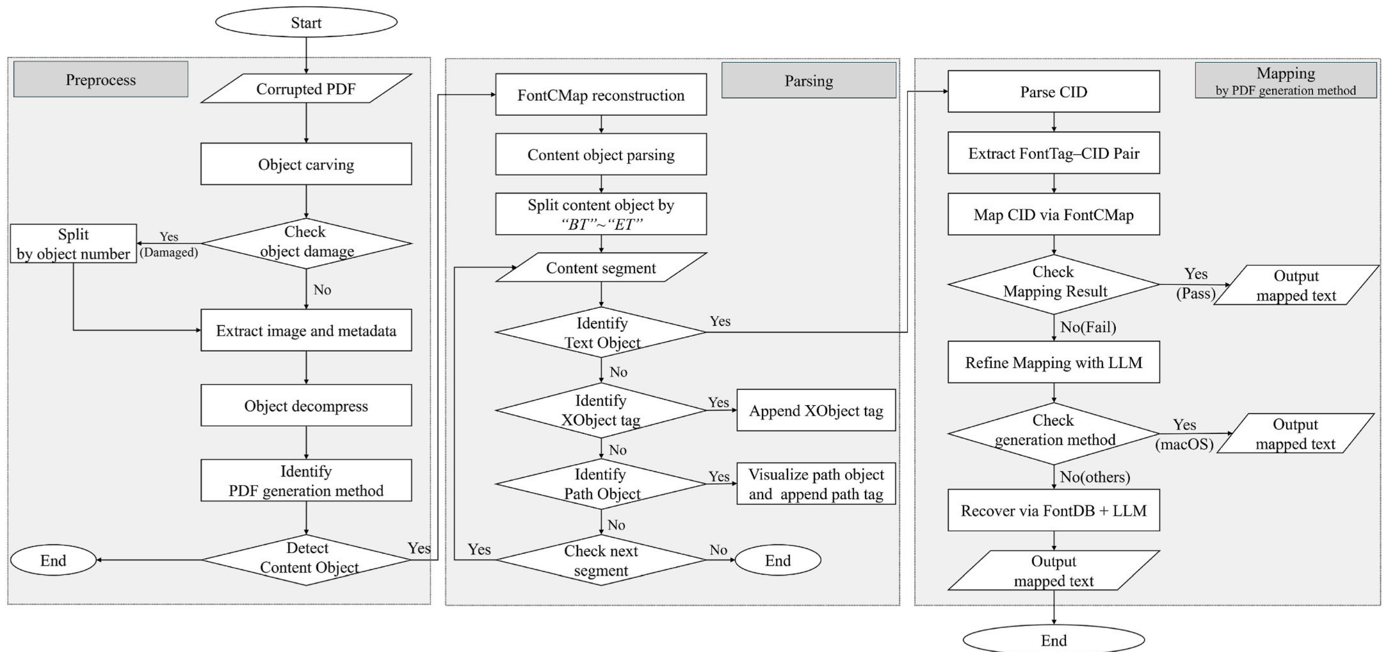

Fig. 2. Cpr algorithm.

byte values of text, while “< >” lists 4-digit hexadecimal CIDs requiring CMap mapping. Each CID is assigned based on the GlyphID in the font file, enabling stable text restoration when the font file is secured. Additionally, when external fonts are used, text is converted to JPEG or PNG images and referenced as XObjects. The `Content Object` includes “/Span”, “Lang”, and “EMC” tags that specify the document's language attributes and syntactic units. Object numbers are assigned in the sequence 1–2–3, with the first and second objects being the `Catalog Object` and `PageTree Object`, respectively.

3.2.3. MS print to PDF

PDFs generated with MS Print to PDF have font tags that reference different fonts on each page. That is, the same `F1` tag may point to “Arial” on page 1 and “Times New Roman” on page 2. CIDs use only the “< >” format and consist of 4-digit hexadecimal values. Each CID corresponds to a GlyphID in the font file, but since the same font tag can reference different fonts on different pages, page-by-page mapping is required. When external fonts are used, `Path Objects` are predominantly employed in the `Content Object`. This causes the Content Object to become lengthy and may result in storage divided across multiple objects. The `Content Object` exhibits a fixed coordinate system pattern beginning with “0.750000 0.000000 0.000000–0.750000 0.000000”. Object numbers are assigned in the sequence 4–5–8, with the first and second objects containing the strings “Identity” and “(Adobe)”, respectively.

3.2.4. Adobe Acrobat

PDFs generated with Adobe Acrobat have font tags that reference different fonts on each page. CIDs use both “( )” and “< >” formats. The “( )” format directly stores byte values, with special characters recorded as octal escapes such as \222, which are converted to hexadecimal and then mapped to Unicode. The “< >” format consists of 4-digit hexadecimal CIDs mapped through CMaps. In Windows, CIDs are assigned based on the font file's Glyph ID, whereas in macOS they may be sequentially assigned starting from 0001. The `Content Object` of the first page is stored divided across multiple objects, with the/P <</MCID tag appearing repeatedly within the `Content Object`, and object numbers assigned irregularly. The first two objects contain strings such as “<</Linearized”, “<</DecodeParms” or “<</Metadata”, “<</Length”.

CID	ArialMT	Brilliant	Calibri	Frenchpress
...	...	...	...	...
0x0006	#	Á	G	C
0x0007	$	Â	H	D
0x0008	%	Ã	I	E
0x0009	&	Ä	J	F
...	...	...	...	...

Fig. 3. Example structure of FontDB Showing CID–Glyph mapping across fonts.

4. Corrupted PDF recovery algorithm

Building upon the analysis presented in Section 3, we propose CPR (Corrupted PDF Recovery), an algorithm that recovers damaged PDF documents by comprehensively utilizing residual data and adapting to producer-specific characteristics. CPR consists of the FontDB construction stage and three recovery phases—Preprocess, Parsing, and Mapping—as shown in Fig. 2.

4.1. FontDB construction

Prior to the recovery phase, a FontDB must be constructed to compensate for damaged or missing CMap information. As described in Sections 3.1.2 and 3.1.3, `Text Objects` rely on the CMap of their associated `Font Objects` to map CIDs to Unicode. When the CMap is corrupted, text recovery becomes infeasible. To address this limitation, this study employs a FontDB-based mapping approach, in which CMap information extracted from font files is used to supplement damaged PDFs, particularly when CID source is GlyphID (see Table 1).

FontDB is constructed by collecting widely used font files (e.g., TTF, OTF, and TTC formats) and converting them into XML representations using font-processing tools such as fonttools (Rossum, 2017). From each font, the *glyphorder* and *name* tables are extracted. The glyphorder table provides mappings between GlyphIDs and glyph names, while the name

table stores font name–related metadata; among its records, $nameID = 6$ (PostScript Name) is used to identify fonts as referenced in PDFs.

The extracted data are inserted into the database after converting glyph names to Unicode using the Adobe Glyph List (Adobe, 2002). Fig. 3 shows an example of FontDB, where each row corresponds to a CID and columns represent Unicode mappings across different fonts (e.g., ArialMT, Calibri, Frenchpress). This structure enables comparison and verification of font-specific CID–Unicode mappings. FontDB can be continuously expanded by collecting additional fonts via web crawling from font distribution platforms, allowing the approach to scale across diverse languages and font types.

4.2. Preprocess

The preprocessing phase performs fundamental operations required for PDF recovery. First, each object is carved based on the “obj” and “endobj” signatures. For object data, the regular expression “(\d+)\s+(\d+)\s+obj([\s\S]*?)endobj” is used, while object numbers are extracted using “(\d+)\s+(\d+)\s+obj([\s\S]*?)”.

Next, the carved objects are examined to determine whether they are corrupted. A damaged object may be missing the “endobj” signature, causing the end offset of the object to be misidentified. To address this issue, corruption is detected based on Equation (1), which compares the sum of the previous object's start offset and data length with the start offset of the next object. In the equation, *offset* denotes the offset of the object number, and *list* refers to the list of object data. If the two values are inconsistent ($\Delta_n \neq 0$), the object is considered corrupted. Such objects are separated using the position of the next valid object number signature as a boundary. The data of corrupted objects is preserved until the start of the next valid object. To maximize recoverability, an “endstream” marker, which is the standard termination signature of an object stream, is appended to each separated corrupted object so that it can be utilized in subsequent decompression.

$$\Delta_n = offset_n - (offset_{n-1} + Length(list_{n-1})), \quad \text{corrupted if } \Delta_n \neq 0 \tag{1}$$

`Image Object` and `Metadata Object` are stored independently within the file and can therefore be extracted regardless of the PDF producer. When an `Image Object` is detected, it is immediately extracted using a PDF image extraction tools such as PyMuPDF (Inc. Artifex software). Each extracted image file is named according to its corresponding object number. If a `Metadata Object` is found, it is also extracted. Subsequently, the PDF generation method is identified by analyzing the object number and data patterns defined in Table 1.

Since multiple objects can be compressed within a single stream, the identified compressed objects are decompressed. Afterward, the essential objects, as described in Section 3.1, are identified based on their unique tags. Next, the PDF generation method is identified by analyzing the object numbers, data patterns, and `Content Object` tags defined in Table 1. If no `Content Objects` remain, further recovery is determined to be unrecoverable, and the process is terminated. Finally, the `Resources Object` is analyzed to identify the mapping between `XObject` tags and object numbers, and the corresponding image file names are renamed according to their `XObject` tags.

4.3. Parsing

The parsing phase is composed of two parts: FontCMap construction, which builds the font mapping from preprocessed objects, and Content parsing, which analyzes the structure of `Content Objects` and extracts three types of objects. The FontCMap is a dictionary that stores pairs of CID and Unicode values corresponding to the font tags used within `Content Objects`. When the objects involved in FontCMap construction are intact, parsing is performed using this information as extensively as possible.

FontCMap - Global	FontCMap[*pidx*]	Supplementary Info

‘F1’: #font tag
{
#CID : Unicode
‘0024’ : ‘A’
‘0025’ : ‘B’
},
‘F2’:#font tag
{
‘0006’ : ‘A’
‘0007’ : ‘B’
},
‘Random1’:
{
‘0024’ : ‘A’
‘0025’ : ‘B’
}
	‘1’ : #pidx
	{
	‘F1’:
	{
	‘0024’ : ‘A’
	‘0025’ : ‘B’
	},
	...
	}
	‘2’: #pidx
	{
	‘F1’:
	{
	‘0006’ : ‘A’
	‘0007’ : ‘B’
	}
	}
		‘FontTag-FontName’ :
		{
		‘F1’: ‘ArialMT’,
		‘F2’: ‘Calibri’
		...
		},
		‘FontName List’ :
		[
		‘ArialMT’,
		‘Calibri’,
		...
		]

Fig. 4. FontCMap structure and its supplementary information.

4.3.1. FontCMap construction

FontCMap construction begins by analyzing the `Resources Objects` to identify font tag dictionaries and determine whether FontCMap is applied globally or defined per page. Each font tag is traced to its corresponding `Font Object`, and the associated CMap is parsed to construct the FontCMap.

When FontCMap is globally applied, a single FontCMap mapping is used across all `Content Objects`. If defined per page, an index is assigned to each `Resources Object` to maintain separate FontCMaps. Correspondingly, a page-level identifier, ***pidx***, is assigned to each `Content Object` via the `Page Object` to align content with the appropriate FontCMap. This *pidx*-based scheme is applied to PDF *generation methods* with page-scoped font tags, such as Microsoft Print to PDF and Adobe Acrobat (Table 1).

Because some objects may be corrupted, supplementary information is collected during construction. Font names are first extracted from `Font Objects` referenced in `Resources Objects` to build a font tag–to–font name dictionary. Font names found across font-related objects, such as `DecendantFonts` and `FontDescriptor`, are then aggregated. In addition, any CMaps recovered from individual `ToUnicode` or `FontFile Objects` are stored as auxiliary entries (labeled Random#) within the FontCMap. Fig. 4 illustrates an example of the constructed FontCMap and its supplementary information.

4.3.2. Content object parsing

After constructing the FontCMap, `Content Objects` are parsed to identify `Text Objects`, `XObjects`, `Path Objects` based on each operator, and to extract `XObject` tags and visualize `Path Objects`. Each `Content Object` is segmented using the “BT” and “ET” operators, and each segment is classified according to the operators it contains. Because operator-like noise may appear within text, classification is performed in a prioritized order: `Text Objects` are identified first, followed by `XObjects` and `Path Objects`.

A content segment is classified as a `Text Object` if it contains the “BT” and “ET” operators. However, when large `Content Objects` are fragmented and these markers are missing, text patterns in CID form, such as “( )” or “< >”, are additionally detected using regular expressions. Identified `Text Objects` are forwarded to the mapping phase, where text is reconstructed using PDF *generation method*–specific mapping process described in Section 4.4.

For content segments not identified as `Text Objects`, `XObjects` are detected by identifying the “Do” operator using the regular expression “/(\w+)\sDo”, and the immediately preceding XObject tag is extracted and stored.

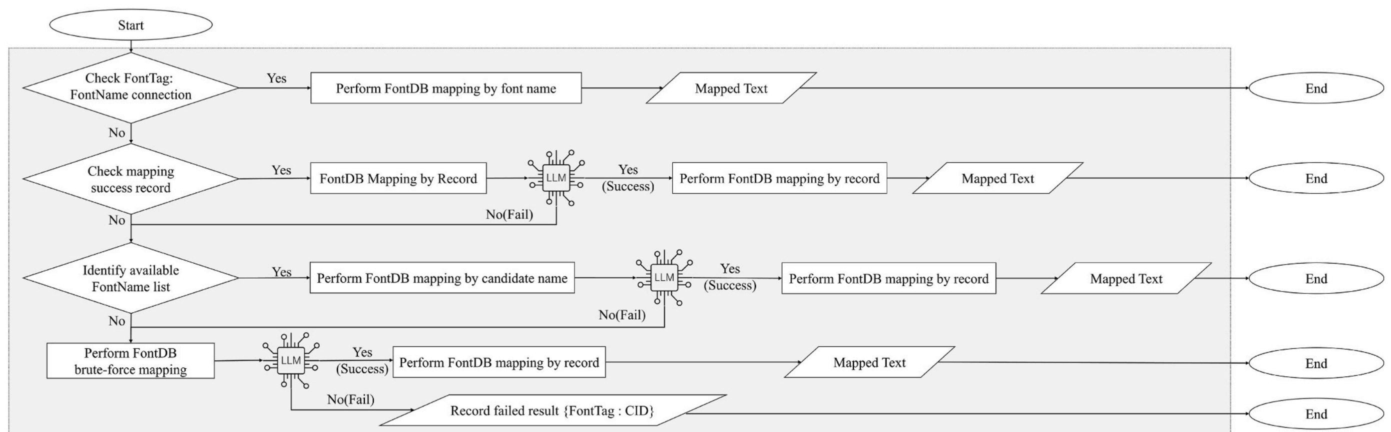

Fig. 5. FontDB-based recovery algorithm in case of CMap damage.

Remaining segments are examined for `Path Objects` by detecting path construction operators ("m" or "re") together with path painting operators (e.g., "f"). Detected `Path Objects` are tokenized and visualized following Appendix B (Algorithm 1), which collects path data until a path-painting operator finalizes the path. Valid paths are rendered as images, and the corresponding output filenames are stored as path tags in the recovery results.

4.4. Mapping

The Mapping phase operates on `Text Objects` identified during parsing, and proceeds according to the characteristics of each PDF *generation method*. When the generation method cannot be determined by Table 1, mapping procedures for all methods are applied sequentially to maximize recovery.

CID values are first parsed based on the formats and interpretation rules described in Section 3.2, producing a list of `FontTag:CID` pairs. These pairs are matched against the FontCMap dictionary to reconstruct text.

For pairs that remain unmapped due to missing or damaged CMap information, an additional recovery step is applied. Consecutive unmapped pairs sharing the same font tag are grouped into `FontTag:CID list` to improve efficiency. During recovery, LLM-based verification is used to validate consistency with candidate CMaps and reduce mismapping. Because single-CID validation is unreliable, multiple CIDs associated with the same font tag are jointly verified to improve mapping accuracy.

4.4.1. macOS - document editor

For the macOS Document Editor, which employs internal CIDs, FontDB cannot be utilized. Mapping is therefore performed solely using the parsed global FontCMap, following the procedure in Appendix B (Algorithm 2). Candidate dictionaries within the FontCMap—including auxiliary entries such as "Random#" and other font tags—are sequentially tested, and each mapping result is validated via LLM-based verification.

Successful mappings are recorded as success records and prioritized when the same font tag reappears to ensure consistency. For instance, if the font tag "F1" is repeatedly mapped to "Random1", a success record is accumulated for that association. When a font achieves more than K successful mappings for a given font tag, it is designated as a confirmed font, and the corresponding CMap in the global FontCMap is updated accordingly. This mechanism reduces redundant mapping attempts and improves efficiency.

Unmapped `FontTag:CID` pairs are retained for later processing. In the final stage, if a confirmed font exists for a font tag, its CMap is applied to the remaining pairs. For single-character CIDs (e.g., spaces or special symbols), where LLM-based validation is unreliable, the CMap of the confirmed font is applied when available to compensate for this limitation.

4.4.2. MS save as

The MS Save as method encodes CIDs based on GlyphID, allowing FontDB to complement mapping when CMap information is damaged. Accordingly, both the global FontCMap and FontDB are used during mapping. Mapping first follows the procedure in Section 4.4.1; if unsuccessful, FontDB-based recovery is applied.

FontDB-based mapping considers two damage cases: partially damaged and totally damaged CMaps. A partially damaged CMap occurs when a font tag's CMap exists in the FontCMap but contains missing CID–Unicode pairs. In this case, if the font tag has already been confirmed through at least K successful mappings, it is treated as incorrectly assigned, its CMap is reset, and mapping is restarted using the FontCMap to reduce mismapping. Otherwise, two fallback strategies are applied: (1) if a FontName linked to the font tag exists, its corresponding CMap is retrieved from FontDB and supplemented; or (2) if no FontName is available, residual CMap clues are used to search FontDB and map the remaining CIDs.

A totally damaged CMap refers to the absence of any CMap for a given font tag in the FontCMap, and the mapping procedure is shown in Fig. 5. If a corresponding FontName exists, its CMap is retrieved from FontDB, registered in the global FontCMap, and applied. Otherwise, candidate FontNames from the pre-collected list are queried in FontDB, and the results are validated via LLM-based verification. If no valid mapping is found, brute-force mapping over the entire FontDB is performed, and only LLM-validated mappings are accepted. Successfully verified mappings are recorded as success records, while unmapped `FontTag:CID` pairs are retained and resolved in post-processing using confirmed fonts when available.

4.4.3. MS print to PDF, Adobe Acrobat

For PDFs generated by MS Print to PDF and Adobe Acrobat, font tag dictionaries are defined per page. Therefore, each page must accurately reference its corresponding FontCMap, which is managed using the *pidx* described in Section 4.3.1.

When both the `Page Object` and the `Resources Object` are intact, *pidx* is correctly mapped between the FontCMap and the corresponding Content Object. In this case, each page is processed by treating FontCMap[*pidx*] as a local dictionary, and the mapping procedure follows the same process as described in Section 4.4.2.

If the `Resources Object` is damaged and FontCMap[*pidx*] is missing, a new FontCMap[*pidx*] is initialized. The recovered CMap is shared across all `Content Objects` with the same *pidx*, enabling consistent reuse and improving efficiency and accuracy. When a

Table 2
Evaluation result by corrupted object type.

Corrupted Object Type	PDF24			PDF4me			iLovePDF pdfonline			CPR		
	Recovery	**CER**	**F1**	**Recovery**	**CER**	**F1**	**Recovery**	**CER**	**F1**	**Recovery**	**CER**	**F1**
Catalog	0.000	–	–	1.000	0.242	0.986	1.000	0.239	**1.000**	1.000	**0.026**	0.988
PageTree	0.000	–	–	1.000	0.000	1.000	1.000	**0.000**	**1.000**	1.000	0.022	0.988
Page	0.000	–	–	0.000	–	–	0.000	–	–	**1.000**	**0.043**	**0.949**
DescendantFonts	0.000	–	–	0.290	0.022	0.891	0.290	0.022	0.891	**1.000**	**0.020**	**0.995**
Font	0.000	–	–	0.000	–	–	0.000	–	–	**1.000**	**0.085**	**0.847**
ToUnicode & FontFile	0.000	–	–	0.000	–	–	0.000	–	–	**0.333**	**0.048**	**0.939**
Resources	0.000	–	–	0.000	–	–	0.000	–	–	**1.000**	**0.092**	**0.947**
Content*	0.000	–	–	0.000	–	–	0.000	–	–	**0.875**	**0.069**	**0.828**
Average	0.000	–	–	0.347	0.243	0.983	0.347	0.240	0.997	**0.924**	0.077	0.934

`Content Object` lacks *pidx* information, independent mapping is first attempted to minimize incorrect assignments, while leveraging reusable CMap data from other FontCMap[*pidx*] entries or global auxiliary entries (e.g., "Random#"). Font name–based searching, success record accumulation, and FontDB brute-force mapping are then iteratively applied.

If the `Page Object` is damaged and *pidx* cannot be directly assigned to `Content Object`, all existing FontCMap[*pidx*] entries are sequentially applied to the `Content Object`, and the results are validated via LLM-based verification. If these attempts fail, a new *pidx* is generated, and recovery proceeds using the same strategy when the `Resources Object` is damaged.

This page-level *pidx*-based mapping reflects the structural characteristic that identical font tags may correspond to different fonts across pages, thereby minimizing mapping errors and improving recovery accuracy even in severely damaged PDFs.

5. Evaluation

5.1. Overview

This section evaluates the performance of the proposed CPR using a multilingual dataset composed of diverse fonts and PDF *generation methods*. A comparative analysis with existing PDF recovery tools was conducted to assess the effectiveness of the proposed algorithm. The implemented CPR tool, along with the dataset and font database used for evaluation, has been released as open source to contribute to the digital forensic community.[1]

5.1.1. Evaluation objective and environment

The evaluation assesses the accuracy and efficiency of CPR in recovering content from damaged PDF documents. During mapping, validation was performed using Llama 3.1:8B, configured to output binary judgments (O/X) based on text naturalness. FontDB contained 40 fonts covering all dataset fonts, and the success record threshold (K) for confirmed fonts was set to five.

5.1.2. Test datasets

The test datasets were created using 10 fonts across three languages including English, French, and Korean, under six PDF *generation methods*, resulting in a total of 22 ground-truth datasets. The datasets included six OS embedded fonts and four external fonts, with each PDF file containing three to five pages.

To simulate damage, eight content-related object types were selectively deleted from ground-truth PDFs, including complete CMap loss cases where both `ToUnicode` and `FontFile` objects were removed. All instances of duplicated objects (e.g., `Page Objects`) were deleted, and PDFs containing only the `Content Object` were also included to emulate realistic corruption. In total, 655 corrupted PDF samples were generated.

5.1.3. Evaluation metrics

The performance of CPR was evaluated using two complementary metrics. First, the recovery success rate is calculated as the ratio of successfully recovered PDFs to the total number of damaged PDFs. A recovery is considered successful when at least part of the content is recovered, even if the document does not match the original perfectly. Conversely, files that remain in a damaged or unrenderable state after recovery are regarded as failures.

Second, the content recovery rate evaluates character-level and object-level accuracy for successfully recovered PDFs, covering `Text Object`, `Path Object`, and `XObject`. Character-level accuracy is measured using the Character Error Rate (CER), computed from the Levenshtein Distance between the ground truth (GT) and the recovered text. At the word level, Precision, Recall, and F1-score are additionally used to assess lexical consistency between GT and recovered results.

5.1.4. Baselines

To evaluate the performance of CPR, four widely used and publicly accessible PDF repair tools were selected as baselines: PDF24 (PDF24), pdf4me (Geek Software GmbH), pdfonline (PDF Tools AG), and iLovePDF (iLovePDF). Each tool was tested on the same dataset, and their recovery performance was compared against CPR.

5.2. Evaluation results

5.2.1. Evaluation by corrupted object type

The evaluation results by corrupted object type are summarized in Table 2. Among the baseline tools, PDF24 (PDF24) failed to recover any of the 655 corrupted PDF samples, while pdf4me (Geek Software GmbH), pdfonline (PDF Tools AG), and iLovePDF (iLovePDF) were unable to restore most files when objects other than the `Catalog` or `PageTree` were damaged. In contrast, CPR consistently achieved higher recovery rates across all object types, with an average recovery success rate of 0.924—approximately a 166 % improvement over the average recovery rate of the best-performing baseline—demonstrating robustness regardless of the corrupted object.

CPR reconstructed text using embedded CID information, with minor differences in line breaks and spacing relative to ground truth, resulting in a slightly higher average CER of approximately 0.02. Despite this variance, CPR showed the most stable performance under font-damage conditions.

When the `Page Object` was damaged, all baseline tools failed to recover content due to their inability to reconstruct inter-object relationships. In contrast, CPR successfully restored all page contents, achieving a 100 % recovery success rate by reconnecting these relationships during mapping. CPR also achieved recovery rates of 100 % for damaged `Resources Objects` and 87.5 % when only the `Content Object` remained, substantially outperforming all baselines.

[1] https://github.com/BeenyHail/CPR.

Table 3
Evaluation Result by PDF *Generation Method* (1) macOS–Document Editor, (2) MS Save as, (3) MS Print to PDF, (4) Adobe Acrobat.

Generation method	iLovePDF pdfonline			CPR		
	Recovery	CER	F1	Recovery	CER	F1
(1)	0.376	0.000	1.000	**0.854**	0.066	0.890
(2)	0.346	0.004	1.000	**0.993**	0.029	0.974
(3)	0.323	0.000	1.000	**1.000**	0.030	0.876
(4)	0.333	0.692	0.981	**0.924**	0.134	0.986

Table 4
Evaluation result by three types of content object.

Content Type	iLovePDF pdfonline		CPR	
	Recovery	CER	Recovery	CER
Text Object	0.346	0.089	0.931	0.050
XObject	0.400	0.00	0.800	0.000
Path Object	0.500	0.00	1.000	0.045

Even in cases of complete CMap loss—where both `ToUnicode` and `FontFile` Objects were removed—CPR achieved a 33.3 % recovery rate with a CER of 0.048, indicating that meaningful text recovery is possible without font-mapping data. The higher error rate in this scenario primarily stems from misclassifications during LLM-based verification.

5.2.2. Evaluation by PDF generation method

Table 3 summarizes the recovery results categorized by PDF *generation methods*, comparing CPR with the best-performing baseline tools identified in Table 2. CPR outperformed baseline tools across all methods, demonstrating that generation-specific structural and procedural characteristics were effectively incorporated into the algorithm. For (1) the macOS Document Editor, recovery was impossible when CMap information was entirely missing; however, when CMaps were present, CPR achieved an 85.4 % recovery rate by fully leveraging the available mapping data. CPR achieved a 100 % recovery rate for (3) MS Print to PDF and near-complete recovery for (2) MS Save as, with only a single failure case.

For (4) Adobe Acrobat, recovery was infeasible when the CIDs were internally assigned by macOS; however, when mixed with GlyphID-based CIDs, text associated with those fonts was successfully reconstructed. The resulting CER of 0.134 reflects unrecovered segments encoded with internally defined CIDs. In contrast, pdfonline (PDF Tools AG), and iLovePDF (iLovePDF) exhibited significantly higher error rates for Adobe Acrobat datasets due to incorrect page-order reconstruction, as objects were restored based on object numbers rather than storage sequence. For other *generation methods*, these tools achieved zero CER whenever recovery was possible, indicating that the overall error rates in Table 2 are primarily driven by failures in the Adobe Acrobat case.

5.2.3. Evaluation by content type

CPR also demonstrated strong recovery performance across the three components of the `Content Object`, as summarized in Table 4. For `Text Objects`, CPR achieved a recovery success rate of 93 % and a low error rate of 0.05, outperforming other tools and confirming its effectiveness in text reconstruction. For `Path Objects`, CPR achieved a 100 % recovery rate across all damaged files; the few observed errors occurred when path construction operators were split across adjacent objects, resulting in incomplete path reconstruction. For `XObjects`, CPR successfully recovered objects even when dictionary information in the `Resources Object` was damaged, whereas other tools failed. In cases where only the `Content Object` remained and `XObject` data were entirely lost, CPR preserved `XObject` tags in the recovered output (`Content.txt`), enabling investigators to infer the prior existence of images or external objects despite irreversible data loss.

Fig. 6 presents an example of a dataset using the OS embedded font “Arial” and the external font “Frenchpress”, along with the corresponding recovery results produced by CPR. When both `Text Objects` and `XObjects` are present within a `Content Object`, the recovered `Content.txt` file contains both the mapped text and the associated `XObject` tags. In addition, the multimedia folder stores the image files corresponding to each `XObject`, named according to their respective tags. When `Text Objects` and `Path Objects` coexist, the visualized `Path Objects` are saved as PNG files in the graphics folder, with each assigned a unique path tag for mapping. These results demonstrate that CPR accurately preserves the storage order within the `Content Object`, enabling correct reconstruction even when multiple object types are interleaved.

6. Conclusion

This paper proposed CPR (Corrupted PDF Recovery), an adaptive recovery algorithm that addresses three major types of unrecoverable PDF damage not supported by existing tools. By analyzing content storage characteristics across different PDF *generation methods*, CPR decomposes the recovery process into generation-specific procedures and maximizes recoverable content by exploiting residual inter-object references and reconstructing disrupted object relationships, even when only a single Content Object remains. The integration of FontDB

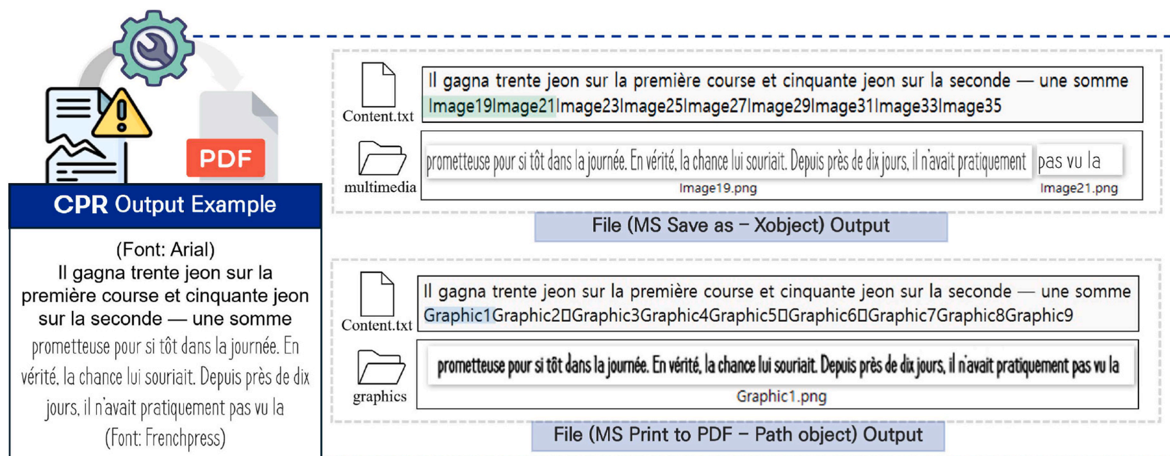

Fig. 6. CPR-recovered output examples by pdf *Generation Method* and object type.

with LLM-based validation further improves text restoration accuracy under damaged or missing CMap conditions. Experimental evaluation demonstrated CPR's language independence and robust recovery performance across diverse corruption types, outperforming existing tools by approximately 166 % in recovery success rate. CPR also successfully recovered all content types, confirming its robustness under severe structural damage and its suitability as a reliable forensic approach for recovering evidential content from damaged PDFs.

Despite these strengths, CPR has limitations. For macOS-generated PDFs that internally assign CIDs, recovery remains challenging when CMap information is absent. In addition, LLM-based verification may yield false positives or false negatives for single-character CIDs or when contextual information is limited.

In future research, we plan to broaden producer analysis to cover additional creation pipelines and emerging PDF generation methods, and to improve mapping verification through task-specific LLM training or fine-tuning to reduce validation errors. Through these improvements, CPR will deliver more accurate and robust recovery across a wide range of producers and languages, and enhance the reliability and consistency of digital evidence analysis.

Acknowledgements

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00398745, Proofs and responses against evidence tampering in the new digital environment).

Appendix. A Source Code and Tool Implementation

All source code developed for CPR, including parsing modules, recovery algorithms, and visualization scripts, is publicly available at https://github.com/BeenyHail/CPR.

B Detailed Algorithms Implemented in CPR

Algorithm 1. Visualization from Path Object.

```
Input: Path Object (stream)
drawing ← False ;                                  // path active flag
c ← [(0,0)] ;                        // list of coordinates for current path
paths ← [] ;              // list to store valid paths for visualization
lines ← SPLIT(stream,"\n") ;              // divide stream by newline
for line in lines do
    tokens ← TOKENIZE(line) ;            // tokenize by whitespace
    for token in tokens do
        if token =="m" or token =="re";          // start of new path
        then
            if drawing and c ≠ ∅ then
                paths.append(c) ;                // save previous path
            drawing ← True
            c ← [(0,0)] ;                               // start new path
            if token =="m" then
                c.append(MOVETO(x, y))
            else if token =="re" then
                c.append(RECTANGLE(x, y, w, h))
        else if drawing ;            // draw: path construction op
        then
            if token =="l" then
                c.append(LINE(x, y)) ;                 // l: draw line
            if token =="c" then
                c.append(CURVE(x1, y1, x2, y2, x3, y3)) ;   // c:
                  curve using control points
            if token =="h" then
                c.append(CLOSE) ;        // h: close current path
            if token ∈ {S, s, f, f*, B, B*, b, b*} then
                paths.append(c) ;                  // end current path
                drawing ← False ;          // reset for next path
                  segment
if paths ≠ ∅ then
    visualize(paths) ; // visualize paths & save as Graphic#.png
    return VISUALIZED PATH OBJECT(GRAPHIC#.PNG)
```

Algorithm 2. FontCMap Mapping with Validation.

```
Input: f: Font tag, cids: CID list, fontcmap: FontCMap dictionary
Output: Mapped text string
// Initialize state variables
success_record ← {} ;  // tracks successful mappings per font
confirmed_font ← [] ;  // list of validated fonts
mapped_text ← None  K ← 5 ;  // threshold for font confirmation
// Early termination if no CMap available
if fontcmap = None or f ∉ fontcmap then
    return {f : cids} ;  // return unmapped data
// Use pre-confirmed font if available
if f ∈ confirmed_font then
    mapped_text ← MAPPING(fontcmap[f], cids)  return mapped_text
// Try each candidate CMap
foreach candidate ∈ fontcmap do
    result ← MAPPING(fontcmap[candidate], cids)  if result ≠ None then
        validation ← LLM-VALIDATE(result)  if validation = "O" then
            // Initialize nested dict if needed
            if f ∉ success_record then
                success_record[f] ← {}
            if candidate ∉ success_record[f] then
                success_record[f][candidate] ← 0
            // Update success count
            success_record[f][candidate] ←
              success_record[f][candidate] + 1
              mapped_text ← result  // Confirm font if threshold reached
            if success_record[f][candidate] ≥ K then
                confirmed_font.APPEND(f)  fontcmap[f] ←
                  fontcmap[candidate] ;  // cache mapping
            break ;  // exit loop after successful mapping
// Return result or original CIDs
if mapped_text = None then
    return {f : cids} ;  // fallback to unmapped
return mapped_text
```

References

Adobe, 2002. Adobe glyph list. https://github.com/adobe-type-tools/agl-aglfn/blob/master/glyphlist.txt. (Accessed 26 September 2025).

Document Management — Portable Document Format — Part 2: Pdf 2.0. Standard ISO 32000-2:2020, 2020.

Geek Software GmbH. pdf4me. https://pdf4me.com/repair-pdf/. (Accessed 26 September 2025).

iLovePDF. Ilovepdf. https://www.ilovepdf.com/repair-pdf. (Accessed 26 September 2025).

Inc. Artifex software. Pymupdf. https://github.com/pymupdf/PyMuPDF. (Accessed 26 September 2025).

Maiorca, Davide, Biggio, Battista, 2019. Digital investigation of pdf files: unveiling traces of embedded malware. IEEE Sec. Priv. 17 (1), 63–71.

Mark, Vol, Krutko, Andrey, Stefanovitch, Nicolas, Postanogov, Denis, 2018. Automatic recovery of corrupted font encoding in pdf documents using cnn-based symbol recognition with language model. In: 2018 13th IAPR International Workshop on Document Analysis Systems (DAS), pp. 121–126.

Nicolas, Stefanovitch, 2022. Recovering text from endangered languages corrupted pdf documents. In: Moeller, Sarah, Anastasopoulos, Antonios, Arppe, Antti, Chaudhary, Aditi, Harrigan, Atticus, Holden, Josh, Lachler, Jordan, Palmer, Alexis, Rijhwani, Shruti, Schwartz, Lane (Eds.), Proceedings of the Fifth Workshop on the Use of Computational Methods in the Study of Endangered Languages. Association for Computational Linguistics, Dublin, Ireland, pp. 78–82.

PDF Tools AG. Pdfonline. https://www.pdf-online.com/osa/repair.aspx. (Accessed 26 September 2025).

PDF24. Pdf24. https://tools.pdf24.org/en/repair-pdf. (Accessed 26 September 2025).

Rossum, J.v., 2017. Fonttools. https://github.com/fonttools/fonttools. (Accessed 26 September 2025).

Sakhawat Hossain, G.M., Deb, Kaushik, Janicke, Helge, Sarker, Iqbal H., 2024. Pdf malware detection: toward machine learning modeling with explainability analysis. IEEE Access 12, 13833–13859.

Zeng, Jinhua, Qiu, Xiulian, 2022. Forensic authenticity examination of pdf documents. In: Third International Conference on Computer Science and Communication Technology (ICCSCT 2022), 12506. SPIE, pp. 1228–1233.