POLYTECHNIC UNIVERSITY OF TURIN
Master's Degree in Cybersecurity Engineering

PDF Forensics and Attack Analysis: Development of a Unified Investigation Tool

Supervisor:
Prof. Andrea Atzeni

Co-Supervisor:
Prof. Paolo Dal Checco

Candidate:
Michele Merico
s332058

Academic Year 2024/2025

Abstract

Nowadays, the number of daily cyberattacks is extremely high. As a mitigation measure and to shed light on such incidents, digital forensics often plays a crucial role. Digital forensics is the process of identifying, collecting, preserving, analyzing and presenting digital evidence to support investigations and legal proceedings. A significant number of the attacks that digital forensics must deal with exploit the Portable Document Format (PDF). For this reason, understanding how to prevent and analyze the misuse of PDF files has become increasingly important.

Despite the existence of several tools for PDF analysis, current solutions present important limitations for forensic usage. In many real-world scenarios, PDFs are not standalone files but are transmitted as email or PEC attachments, making it essential to analyze not only their internal structure but also their associated transmission metadata in a forensically sound manner. Most existing tools either focus on static structure inspection or on malware detection, but they lack integration with email metadata, do not ensure forensic soundness, do not support the analysis of embedded files within the original PDFs and cannot automatically process PDFs embedded in emails or PECs. Moreover, few of them can verify embedded digital signatures or identify objects that have been modified after signing or only partially covered by digital signatures in a forensically reliable way. These shortcomings make it difficult for investigators to safely extract, analyze, and correlate PDF evidence while maintaining data integrity and traceability.

This thesis addresses these issues by developing an integrated tool for PDF forensic analysis called **foredf**. The work first studies the structure of PDFs, emails and PECs, then evaluates existing tools, identifying **peepdf** as the most suitable starting point. Since no existing solution could automatically parse and analyze PDFs from emails or PECs while preserving metadata, a new parsing module was implemented. Furthermore, **peepdf** was extended to support verification of embedded objects and digital signatures. The entire toolset was executed in a containerized environment to ensure forensic soundness, prevent any alteration of evidence, and automatically generate human-readable preliminary reports, serving as a foundation for subsequent full forensic reports.

The tool was evaluated on a variety of PDFs, including those retrieved from emails and PECs containing fake or partial digital signatures and embedded content, analyzed both statically and partially dynamically. The results demonstrate that **foredf** allows users with moderate technical skills to verify PDF integrity, while providing forensic experts with detailed object-level information and metadata correlation. The use of containerization ensures secure handling of potentially malicious PDFs, reducing risks for analysts and systems. Additionally, **foredf** may support investigators in tracing the origin of attacks or harmful attachments without interacting directly with the files, further improving safety and efficiency in forensic investigations.

With further refinements, **foredf** could become a valuable tool for forensic professionals, supporting PDF investigations both as standalone files and when embedded in emails or PECs. It may also help non-expert users make informed decisions about whether to open received PDF documents, making it a potentially powerful tool not

only for forensic analysis but also for proactive cybersecurity.

Contents

List of Tables

List of Figures

1 Introduction

Before diving into the forensic analysis of Portable Document Format (PDF) files, it is essential to understand what digital forensics is and why it plays a critical role in modern investigations. Digital forensics is the process of identifying, collecting, preserving, analyzing and presenting digital evidences to support investigation and legal proceedings. It is mostly used to investigate cybercrimes but it can be also used as support for civil and criminal investigations.

Digital evidence, the core of digital forensics, refers to any information with evidential value that is either stored or transmitted in digital form through digital devices. This can include files, emails, logs, metadata and more. However, managing digital evidence poses unique challenges: unlike physical evidence, digital data is volatile and can be easily altered, overwritten or destroyed even just through normal use of the device. Evidences must be acquired using forensic sound methods that ensure its authenticity and integrity are preserved from the moment of collection to its presentation in a legal context.

In the field of digital forensics, challenges are becoming increasingly complex. Forensic experts are required to be highly skilled and well-trained in the use of specialized tools, even those open-source, and techniques. Open-source tools allow experts to analyze their source code further ensuring their transparency and reliability. This openness not only helps ensure the trustworthiness of the tools but also supports compliance with major legal and forensic standards. Moreover, digital forensic experts must have a deep knowledge of laws and regulations to ensure following rules during investigations. Furthermore, they must be impartial and objective during the whole forensic process since each personal consideration will result in the inadmissibility of corresponding evidence.

However, despite the availability of many tools and methodologies, the field still suffers from a high *failure rate* in practice. Many investigations fail to produce admissible or consistent results due to tool misconfiguration, undocumented steps, or lack of reproducibility in the analysis process. As highlighted in studies [29, 27], a large number of forensic investigations involve manual tool switching or ad-hoc workflows, which significantly increase variability and reduce reliability. Similar evidence from broader computational research [9] indicates that the majority (about 70%) of experiments suffer from reproducibility issues due to non-standardized environments. This situation undermines the credibility of forensic results and emphasizes the need for reproducible, automated, and well-documented workflows that ensure consistent outcomes across different systems and operators.

1.1 Scope and Objectives

PDF files are widely used for document sharing due to their portability and consistency across platforms. However, such versatility introduces opportunities for misuse. From a digital forensics perspective, PDF files can be very important evidences in cybercrimes, fraud investigations and intellectual property cases.

The main objective of this thesis is to investigate the PDF, focusing on its structure and features, in order to understand how it can be exploited to carry out

cybercrimes and how compromised or malicious PDF files can be identified through forensic analysis. PDF files embedded within email communications, especially certified emails (Posta Elettronica Certificata, or PEC), will be also analyzed as they can serve as effective vectors for delivering malicious content.

As part of this research, a prototype of forensic tool based on a well-known forensic tool will be created. The original forensic tool will be examined and extended to enable the automatic parsing of PDF files, including those contained within PEC messages, as well as their subsequent analysis. The proposed framework will also address the reproducibility challenge by running in a containerized environment that guarantees identical outcomes from identical inputs, minimizing dependency on the underlying system configuration. While still allowing manual inspection and analyst control, the system will support integration into automated pipelines to perform predefined checks or batch analyses, enabling scalable and reproducible workflows, thus reducing both human error and failure rates in practical forensic scenarios.

The main objective of this thesis is to investigate the PDF, focusing on its structure and features, in order to understand how it can be exploited to carry out cybercrimes and how compromised or malicious PDF files can be identified through forensic analysis. PDF files embedded within email communications, especially certified emails (Posta Elettronica Certificata, or PEC), will be also analyzed as they can serve as effective vectors for delivering malicious content. As part of this research, a prototype of forensic tool based on a well-known forensic tool will be created. The original forensic tool will be examined and extended to enable the automatic parsing and scanning of PDF files, even those contained in PEC messages.

1.2 Thesis Structure

This thesis is organized as follows:

- Chapter 1 - Introduction: introduces digital forensics, the importance of PDF files in investigations, and defines the objectives and structure of the thesis.
- Chapter 2 - State of the Art: presents the current state of the art in PDF forensics and outlines how the proposed forensic tool fits within this context.
- Chapter 3 - Digital Forensics Investigation: provides a general overview of digital forensics processes, methodologies and the handling of digital evidence.
- Chapter 4 - Portable Document Format: describes the PDF format, its syntax, objects, file and document structures, content streams, incremental updates, encryption, metadata and watermarks.
- Chapter 5 - Digital Signatures in PDFs: explains digital signatures, signing and verification phases and PKCS#7 implementation. It explains also multiple signatures, signature extraction and verification techniques.
- Chapter 6 - Emails and PEC Introduction: presents the structure of email and certified email (PEC) and protocols they use. Their implications in forensic investigations and their relevance for PDF acquisition and analysis are also discussed.

- Chapter 7 - PDF Attack Techniques and Forensic Analysis: different attacks based on the use of PDFs are discussed, alongside forensic analysis techniques.
- Chapter 8 - Acquisition Sources of PDF Files: forensic acquisition principles and potential sources of PDF evidence are discussed and analyzed.
- Chapter 9 - Development of Foredf for Forensic Analysis of PDFs: describes the enhancement of an existing forensic tool and the development of forensically sound email and PEC metadata and files parser, which is then integrated to create a new, more comprehensive forensic tool.
- Chapter 10 - Foredf: Attack Prevention and Analysis : describes potential attacks scenarios and the role that foredf can play.
- Chapter 11 - Conclusions and Future Works.

2 State of the Art

The analysis of PDF files within digital forensics has been the focus of various research efforts and practical tools over the past decade. Investigators have developed multiple approaches to extract evidential information, detect malicious content, and identify structural anomalies. These approaches can be broadly categorized into static analysis, dynamic analysis and hybrid methods that combine both strategies.

Recent literature confirms this tripartite taxonomy. Abdallah et al. [1] provide a comprehensive survey on malicious PDF detection, systematically classifying approaches into static, dynamic and hybrid categories. Their study highlights that hybrid strategies, those that combine structural inspection with runtime observation, are increasingly effective against obfuscated and evasive attacks, achieving higher robustness than traditional static methods alone. This observation strongly supports the evolution of PDF forensic tools toward integrated, multi-phase analysis frameworks.

Static analysis tools parse the internal structure of PDF files, inspecting objects and streams and detecting suspicious patterns without executing the file. On the other hand, dynamic approaches try to monitor the behavior of PDF documents when opened in controlled and safe environments, capturing any malicious actions triggered at runtime. Hybrid methods are built to take advantage of the strengths of both static and dynamic analysis, providing a more complete forensic assessment.

This chapter examines the most widely used tools and frameworks, highlighting their capabilities and limitations, and identifies gaps that motivate the enhancements introduced in this thesis.

2.1 Existing Tools and Frameworks

As stated before, the tools available for PDF forensic analysis can be grouped according to the approach they employ: **static**, **dynamic** or **hybrid**.

2.1.1 Static Analysis Tools

PDF Tools [51] by Didier Stevens are among the first open-source utilities designed to detect potentially dangerous elements such as embedded JavaScript or automatic actions. They are a collection of lightweight, practical utilities widely used in PDF forensics. The suite includes tools such as `pdf-parser.py` for low-level inspection of PDF objects and streams, `PDFiD` for fast keyword-based triage, `make-pdf` utilities for generating test PDFs with embedded JavaScript or files and `pdftool.py` for inspecting incremental updates. These tools emphasize simplicity, robustness, and transparency, allowing analysts to examine potentially malicious PDFs without executing them.

These tools are particularly useful to identify structural anomalies, embedded scripts, automatic actions ,triggered when the files are opened for example, and they are widely referenced in both research and practical forensic workflows. However, they are static analysis tools: they do not perform dynamic analysis, signature validation and often require manual and human interpretation. Despite these limitations, the suite remains a fundamental reference in PDF forensic analysis and

provides a solid baseline for more advanced frameworks.

peepdf [20] represents a more advanced static analysis framework. It is an open-source Python tool designed for the forensic analysis of PDF files. Its main strength lies in the ability to deeply examine the internal structure of PDF documents, identifying potentially malicious components such as embedded JavaScript, shellcode and compressed objects. Through its command-line interface, `peepdf` provides precise and detailed control during analysis, making it particularly useful for digital investigators and security researchers.

Key features of `peepdf` include:

- **Object and stream analysis**: decodes and inspects compressed objects, such as those encoded with `FlateDecode` or `LZWDecode`, and allows detailed examination of streams within the PDF.
- **Embedded JavaScript analysis**: supports decoding, beautification and controlled execution of embedded JavaScript to detect potential exploits or malicious behaviors.
- **Shellcode emulation**: libraries like Pylibemu are integrated into peepdf to emulate the behavior of embedded shellcode or payloads, facilitating the detection of maliciousness.
- **Structural and metadata analysis**: provides detailed information on the physical and logical structure of PDFs, including metadata, versions and incremental updates.
- **PDF creation and modification**: allows creation of new PDF files or modification of existing ones, including the addition of JavaScript, object compression and embedded files.

It can be found in security-related Linux distributions, such as Kali Linux. For this reason, forensic and security experts can easily use it and at the same time, its presence in security-related distributions highlights its reliability and widespread adoption in PDF forensic workflows.

The peepdf is indeed a powerful tool, but it has some shortcomings. It does not offer any kind of sandboxing to keep the runtime activity under observation. Further, it does not provide built-in capabilities to verify digital signatures and analyze embedded files (if any). Additionally, interpreting the results often requires advanced expertise, as the tool provides detailed insights without automating risk assessment or large-scale analysis.

Other frameworks such as **Origami** [16] or **PDF Examiner** [21] provide similar static capabilities, but they are either outdated or limited in extensibility and forensic scope.

2.1.2 Dynamic and Hybrid Tools

Dynamic analysis tools focus on executing PDF files within controlled sandbox environments to observe their runtime behavior. By monitoring system calls, registry

changes, network activity and embedded script execution, these tools can detect obfuscated or previously unknown malware that might evade static inspection. Well-known examples include **Cuckoo Sandbox**[15], an open-source automated malware analysis system that can execute PDFs in virtual machines, **Joe Sandbox**[30], a commercial platform supporting PDF analysis with detailed behavioral reporting, and online services such as **Any.Run**[7], which provide near-real-time execution reports highlighting embedded scripts, network interactions, and system modifications.

On one hand, dynamic analysis is a particularly effective approach to discovering hidden threats; on the other hand, it requires significant resources and it may not be completely reproducible thus making it less suitable for traditional forensic workflows that value evidence preservation and repeatability.

Hybrid analysis combines static inspection with selective dynamic monitoring to achieve more effective malware detection. Abdallah et al. [1] emphasize that such hybrid methodologies significantly reduce false negatives in the presence of obfuscation, as they combine structural analysis with behavioral evidence. Their survey also points out that while static techniques are efficient and transparent, they often fail against encrypted or obfuscated PDFs; conversely, dynamic techniques can detect these cases but lack scalability. Thus, hybrid approaches are the most promising way of resilient and forensic-oriented analysis of PDF. Moreover, Liu et al. [36] show that combining structural analysis, intermediate representations and language models yields more adversarially robust systems. Combined, these studies highlight that merely examining structure is inadequate against advanced malware.

2.1.3 Integration in Forensic Suites

General-purpose forensic suites such as **Autopsy**[8], **FTK**[22] and **X-Ways Forensics**[4] include partial support for PDF documents, focusing mainly on metadata extraction and previewing. However, they do not provide detailed analysis of the internal PDF structure or mechanisms to detect malicious or manipulated content.

This divergence between forensics research and practice is also evident in recent meta-surveys. Javed et al. [29] expose forensic fragmentation, non-standardization and lack of automation as well as limits to the scalability of current tools. In a similar vein, the DFPulse survey [27] indicates that users often make extensive use of manual and semi-automated workflows with more than one tool, which undermines reproducibility and facilities case backlogs. Both studies highlight the need for reproducible, automated and evidence-based pipelines, features that remain scarce in PDF analysis environments.

2.2 Limitations of Current Approaches

Despite the great progress made by research in PDF forensics, there are several limitations that still restrict the effectiveness and reliability of existing tools. These limitations are not only in the field of analytical completeness but also implications on reproducibility and the possibility for mistakes, both important considerations in forensic procedures. These limitations are now discussed in detail:

- **Lack of automation and scalable workflows** - many tools, including widely adopted frameworks such as **PDF Tools** [51] and **peepdf** [20], require manual user interaction to inspect objects or interpret results. This dependency on the operator significantly slows down the process and increases the likelihood of inconsistent outcomes across different investigations. Surveys such as DFPulse [27] and Javed et al. [29] confirm that analysts often combine multiple semi-automated utilities, resulting in fragmented and hard-to-reproduce workflows. In large-scale or repetitive investigations, this manual handling becomes unsustainable and leads to inconsistent findings.

- **Limited support for digital signatures** - several static and dynamic frameworks either ignore or only partially implement signature validation. For example, **peepdf** and **Origami** [16] can parse document objects but do not verify cryptographic integrity of them. This restriction inhibits full forensic analysis of tampering, and even worse the provenance problem, that is crucial for legal or evidentiary environments where authentication should be verified.

- **Incomplete handling of embedded content** - most existing solutions cannot handle the analysis of complex embedded elements, such as compressed files, images, or secondary documents. Static analyzers such as **peepdf** [20] identify structural anomalies and indicate potential suspicious embedded objects, without considering embedded binaries or attachments. As a result, malicious payloads can hide particularly well when they are nested or obfuscated reducing the overall analysis precision and coverage.

- **Insufficient forensic reporting and evidence traceability** - forensic tools tend to produce technical log or raw values based on a command-line output, rather than preparing something that can then be used in the future for redacting a real forensic report. For example, **peepdf** gives a comprehensive analysis, but it does not have an immediate report mechanism that includes metadata, timestamps, and analysis provenance. Lack of standardization not only slows down writing, but also hampers the reproducibility of evidence: experts may have a hard time unwinding how they reached their conclusion without re-running an analysis from scratch.

- **Fragmented and non-reproducible workflows** - this issue, emphasized in both DFPulse [27] and Javed et al. [29], stems from analysts mixing different tools and environments (Windows, Linux, virtualized sandboxes) with inconsistent configurations. When analysis conditions are not controlled or documented, results become non-deterministic. Two analysts may reach different outcomes even when analyzing the same file. The reproducibility literature [9] within digital forensics is also clear that the lack of containerization, version control, and environment consistency are key contributors of this variance.

In summary, the limitations of existing approaches can be traced to the combined effect of manual dependencies, lack of reproducibility mechanisms, and insufficient automation. These shortcomings motivate the development of a reproducible, containerized and transparent system capable of reducing operator error and increasing

the reliability of forensic analyses. The framework proposed in this thesis directly addresses these problems by providing a consistent, automated environment where every analytical step is traceable and repeatable.

2.3 Motivation and Thesis Contribution

To address the gaps identified in Section 2.2, this thesis introduces an extended framework built upon **peepdf**, integrating advanced forensic capabilities and addressing major weaknesses of current tools. The proposed system encapsulates **peepdf** within a containerized environment capable of automatically fetching emails from certified (*PEC*) or standard mailboxes, extracting PDF attachments, and conducting a full forensic analysis.

- **Reproducibility and automation**: in forensic analysis, reproducibility and consistency are critical. However, studies such as DFPulse [27] and Javed et al. [29] indicate that a large number of investigations involve manual tool switching or undocumented configuration steps, leading to inconsistent results. Similar evidence from broader computational research [9] reports that over 70% of experiments suffer from reproducibility issues due to non-standardized environments. The proposed framework solves this by running in a containerized environment that guarantees identical outcomes from identical inputs, reducing dependency on system configuration. Our tool still supports manual interaction for detailed inspection and decision-making, preserving analyst control. However, by running within a containerized environment and offering extensibility, it can also be integrated into automated pipelines to perform predefined checks or batch analyses, enabling scalable and reproducible workflows when desired.

- **Reduction of human error:** manual steps in PDF analysis, such as parameter selection or interpretation of results, are prone to operator mistakes. These may cause discrepancies in object parsing or risk misclassification of content and results. Automating data acquisition, processing and reporting ameliorate the latter risk. Each analysis stage is logged and versioned, guaranteeing traceability and forensic audit.

- **Enhanced transparency**: structured metadata records the environment, tool versions and configurations used for each step. This permits third parties to replicate analyses precisely and confirm their authenticity, in accordance with forensic principles of repeatability and verifiability.

To support these features, the proposed framework integrates the following new or extended functionalities:

- **Fully reproducible workflow**: achieved through containerization, ensuring environmental consistency and deterministic outputs with same inputs;

- **Digital signature verification**: visual differentiation of signed and unsigned objects within the document structure;

- **Embedded file inspection**: entropy and MIME checks, hash computation, VirusTotal integration, and YARA-based rule scanning;
- **Automated forensic reporting**: creation of structured, metadata-heavy reports that increase traceability and reliability of evidence.

Overall, we present a cohesive, automated and reproducible workflow which both improves efficiency and scientific rigor early in the investigation of PDF forensics and directly tackles issues of reproducibility and human error found in precedent work.

2.4 Comparison of Existing Tools

Table 1: Comparison of existing tools for PDF forensic analysis

Tool	Main Features	Strengths	Limitations	Reference
PDF Parser	Object and stream inspection, structure traversal	Flexible and scriptable via Python	No automation, lacks advanced reporting	[51]
peepdf	Interactive shell for PDF structure analysis, object parsing, stream decoding	Advanced static inspection, supports JavaScript emulation	No signature verification, limited embedded file analysis, manual workflow	[20]
Origami	Ruby-based framework for PDF analysis and manipulation	Extensible, supports creation and modification of PDFs	No malware detection, limited forensic use	[16]
PDF Examiner	PHP-based static analyzer, supports PDF object inspection and JavaScript detection	Simple to use, open-source	Limited automation, not designed for forensic pipelines	[21]
Proposed Tool	Containerized, plug-and-play workflow integrating email fetching, peepdf core, digital signature verification, embedded file analysis and report generation	Automated fetching and basic report generation, forensically sound, unified workflow, minimal setup	Requires container environment	[38]

2.5 Summary

In summary, while several tools exist for analyzing PDF files, few provide an integrated and forensically sound framework that combines acquisition, analysis and partial reporting. Recent literature demonstrates that detection accuracy alone is

insufficient; automation, embedded file handling, and signature verification are essential to produce actionable forensic evidence. The proposed solution builds upon these insights to deliver a more comprehensive, automated and verifiable approach to PDF forensic analysis.

3 Digital Forensics Investigation

Digital forensics is a **structured and scientific discipline** dedicated to identifying, preserving, analyzing and presenting **digital evidence** in a manner that ensures its reliability and legal admissibility. It serves as a fundamental component in investigating cybercrime, financial fraud, intellectual property theft, insider misuse, and other offenses where digital information is central to the case.

Digital evidence refers to any information stored or communicated in digital form that holds probative value for an investigation or legal proceeding [52, 47]. This evidence can originate from a broad array of sources, including computers, mobile phones, network devices, cloud servers, IoT systems, and even social media platforms. Digital evidence may take the form of emails, metadata, images, transaction records, log files and other artifacts that shed light on user activities or intent. Due to its fragile and easily alterable nature, digital evidence must be collected using specialized tools and handled according to established procedures to ensure authenticity, integrity and admissibility in court.

Recognized frameworks such as the *NIST Guide to Integrating Forensic Techniques into Incident Response* [31] and *ISO/IEC 27037: Guidelines for Identification, Collection, Acquisition and Preservation of Digital Evidence* [28] categorize the forensic process into five interdependent phases:

1. **Identification**: the process begins with identifying all potential sources of digital evidence. These may include computers, smartphones, external drives, servers, IoT devices and cloud storage services. Overlooking a critical evidence source can jeopardize the integrity and completeness of the investigation.

2. **Preservation**: digital evidences, once identified, must be collected and preserved without any alterations to maintain integrity and admissibility. The first step of the process of correct preservation is to create **forensic images** of the original data. Forensic images are bit-a-bit copies of the original data which ensure reproducibility of analysis and integrity on the original data along the whole forensic process. However, proper documentation of the chain of custody and the use of cryptographic hash values during the whole process ensure integrity throughout the process.

3. **Extraction and Analysis**: in this stage, data are extracted from preserved forensic images and examined using specialized forensic tools. The analysis of these data can serve different purposes: recovering deleted content, decrypting protected data, correlating system logs, reconstructing timelines and any other useful data. In this phase, challenges, such as handling of large data volumes, breaking strong encryption and facing anti-forensic techniques used by adversaries, are commonly encountered by forensic investigators.

4. **Documentation**: each action during the investigation, including tools used, procedures followed and hash values computed, must be meticulously documented. Detailed documentation provides transparency, reproducibility and admissibility. Evidences which lack sufficient procedural documentation are often dismissed by courts.

5. **Presentation**: the final phase translates technical findings into clear, concise, and precise reports for stakeholders such as law enforcement, legal authorities or organizational executives. This phase often involves expert testimony and visualizations that effectively communicate complex technical results.

3.1 Key Challenges in Digital Forensics Investigations

Despite standardized methodologies, digital forensic investigations face several persistent challenges:

- **Data Volume**: the exponential growth of digital data complicates timely analysis. Automated triage systems and AI-assisted analytics are increasingly employed to improve efficiency and accuracy [23, 12].
- **Encryption**: a common challenge while trying to access data is encryption. Investigators may utilize memory captures or seek legal mechanisms to obtain decryption keys in order to freely access, acquire and analyze data.
- **Cloud and Jurisdiction**: data stored across multiple jurisdictions presents complex legal and technical challenges, particularly under cross-border data protection frameworks such as the GDPR. The distributed nature of cloud storage requires specialized tools and clear legal frameworks for evidence acquisition [23].
- **Anti-Forensics**: adversaries may use methods such as steganography, log wiping or multiple layers of encryption to mislead or delay forensic analysis.
- **Mobile and IoT Forensics**: the proliferation of smartphones, IoT and wearable devices demands advanced techniques to extract evidence from a growing variety of platforms [23, 12].

4 Portable Document Format

The **Portable Document Format** (PDF), originally developed in 1993, is a widely used file format designed to reliably represent documents across different devices and platforms, independently of the application software, hardware or operating system used to create them. PDF is commonly used in legal, academic and professional environments where document integrity, authenticity and long-term accessibility are essential.

This chapter is based on the specifications provided in the original Adobe PDF 1.0 reference manual [3], the PDF 1.7 reference manual [2], and the PDF 2.0 reference manual [19], which collectively illustrate how the format has evolved and been standardized over time.

4.1 General Properties of the PDF Format

The PDF format has several key properties that contribute to its portability, efficiency and security. These features have made it a widely adopted standard in legal, forensic and archival contexts:

- **Portability**: PDF files are platform-independent and stored as 8-bit binary data, ensuring consistency across different systems.
- **Compression**: PDFs support multiple compression methods, in order to reduce the size of files while maintaining their integrity. Some of the most commonly used compression methods, supported by PDFs, are JPEG, JBIG2 and Flate.
- **Security**: PDFs offer some security mechanisms, including encryption and digital signatures. Encryption can be used to ensure different access rights to different users who can either view, edit or print the files. On the other hand, digital signatures can be used to verify authenticity and detect unauthorized modifications.
- **Incremental Updates**: incremental updates allow to append modifications to original content of the PDF files while preserving the original content. This allows tracking of document revisions and updates.
- **Random Access**: a cross-reference table enables direct access to objects and pages, improving efficiency in document parsing.
- **Extensibility**: PDFs allow embedding of custom metadata or annotations.

4.2 PDF Syntax

Due to its portable nature, a PDF requires a defined **syntax** to ensure consistency across different platforms and systems. This syntax establishes a set of rules detailing how data is structured, stored and interpreted within a PDF file.

The PDF syntax consists of four fundamental components:

- **Objects**: in the PDF, the representation of data relies on a set of fundamental object types. These objects form the basic building blocks for all content within a PDF document, including text, graphics and metadata.
- **File Structure**: it specifies how objects are organized within a PDF file. It defines the rules for object storage, access and updating, ensuring efficient navigation within the document. This structure is independent of the semantics of the objects themselves. Additionally, it includes security mechanisms such as encryption, which protect the document from unauthorized access.
- **Document Structure**: it defines how objects are utilized to represent key components of the document. These components include pages, fonts, annotations and other elements necessary for document composition. The document structure ensures that the relationships between these objects are maintained, facilitating correct rendering and logical organization.
- **Content Streams**: they contain sequences of instructions that describe the graphical appearance of the document. These streams dictate how text, images, and vector graphics are positioned and rendered on a page. Although content streams are represented as objects, they are conceptually distinct from the document structure, as they focus specifically on visual presentation rather than logical organization.

4.2.1 Objects

An object in PDF is a distinct data entity that can be labeled and referenced, allowing for the construction of complex data structures.

PDF defines eight basic object types, each with a specific purpose:

- **Boolean Objects**: represent logical values, either `true` or `false`. They are used for conditional logic, flags and settings within PDF structures.
- **Numeric Objects**: represent numerical values, including integers and real numbers. Integers are whole numbers (e.g., 123, -5), while reals are floating-point numbers (e.g., 3.14, -0.002). They are used for coordinates, dimensions and numerical data.
- **String Objects**: sequences of bytes representing text or binary data. They can be literal strings (enclosed in parentheses) or hexadecimal strings (enclosed in angle brackets). Used for text content, file names and arbitrary binary data. Example: `(This is a string)`, `<4E6F7620737472696E67>`.
- **Name Objects**: atomic symbols, uniquely identified by a sequence of characters, beginning with a slash (`/`). Used as keys in dictionaries and as symbolic names. Example: `/Name`, `/Font`, `/Page`.
- **Array Objects**: ordered collections of other PDF objects, allowing heterogeneous data structures, enclosed in square brackets (`[` and `]`). Used to represent lists of values, sequences of coordinates and other structured data. Example: `[1 2 3]`, `[true (text) /Name]`.

- **Dictionary Objects**: collections of key-value pairs, where keys are name objects and values can be any PDF object. Used to define attributes and properties. Example: `<< /Type /Page /Contents 10 0 R >>`.
- **Stream Objects**: sequences of bytes, typically used for large data like images or compressed content, associated with a dictionary describing the stream's properties. Used for embedding images, fonts and other binary data.
- **Null Object**: represents the absence of a value, indicating optional or undefined values. Example: `null`.

Within a PDF document, objects can be labeled and thus becoming **indirect objects** to facilitate referencing and sharing objects. This mechanism is essential for efficient storage and manipulation of complex data structures. An indirect object is uniquely identified by an **object number** and a **generation number**. The object number uniquely identifies the object within the PDF file and it is typically assigned consecutively by PDF itself. On the other hand, the generation number is used to distinguish between different versions of the same object and it is particularly relevant when objects are modified through incremental updates. The other objects can use this unique identifier to refer to it, avoiding nesting duplicate objects while enhancing the reuse of the same original object.

For instance, consider a scenario where a PDF document contains an image that is used multiple times across different pages. Instead of embedding the image data repeatedly, it can be stored as an indirect object, allowing each page to reference it. The definition of the image object could appear as:

```
10 0 obj
    <image data>
endobj
```

Here, `10 0` identifies the image object. Any other object in the PDF can now refer to this image using the syntax `10 0 R`. For example, a page's content stream might include:

```
/Image1 10 0 R Do
```

This indicates that the image labeled `Image1` is represented by the indirect object `10 0` while `Do` indicates to PDF renderer to draw or render the image referenced.

Another example involves a font definition that is used across various text elements in a document. The font definition can be stored as an indirect object, allowing multiple text elements to reference it.

```
20 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont /Helvetica
```

```
>>
endobj
```

And then, a text object can refer to it (indicated by the `R` operator):

```
/Font 20 0 R
```

Indirect objects are essential for reducing redundancy and improving the efficiency of PDF documents, especially when dealing with large or frequently used objects.

4.2.2 File Structure

The **File Structure** of a PDF file is a crucial element. It dictates how objects are organized within the file, ensuring efficient random access and support for incremental updates.

A PDF file consists of four main components (see Fig. 1):

- **File Header**: the first line of a PDF file contains the header, which specifies the PDF version the file conforms to. It begins with the string "%PDF-" followed by the version number, such as 1.0, 1.1, 1.2, and so on, with 2.0 being the latest. Starting from PDF 1.4, the version declared in the header can be overridden by the *Version* entry within the document's catalog dictionary located by means of the *Root* entry in the file's trailer, enabling updates via incremental changes (see Section 4.2.3). This feature ensures backward compatibility, allowing newer PDF features to be ignored by older applications that do not support them. Additionally, some PDF files may include a comment line (marked by a "%") immediately following the header. This line contains at least four binary characters, which help file transfer applications determine whether the file should be treated as text or binary.

- **File Body**: the body contains a series of indirect objects that define the document's content. These objects represent various components, including fonts, pages and embedded images.

- **Cross-Reference Table**: the **cross-reference table** allows fast access to indirect objects within a PDF without requiring a full file scan. It consists of sections, each starting with the keyword `xref`, followed by subsections containing entries for a range of object numbers.

  Each entry is exactly **20 bytes** long and follows this format:

  ```
  nnnnnnnnnn ggggg t eol
  ```

  where `nnnnnnnnnn` is a 10-digit byte offset representing the position in the file where the corresponding object definition begins, `ggggg` is a 5-digit generation number, `t` represents the **entry type** (`n` for in-use objects and `f` for free objects) and `eol` is a 2-character end-of-line sequence.

Each **generation number** starts at `00000` when an object is first created. If an object is later deleted, it is marked as free (`f`), and its generation number is **incremented by 1**. The maximum value is **65535**, beyond which the object can no longer be reused.

Free objects form a **linked list**, allowing efficient management of deleted objects. The **first entry (object 0) is always free** with a generation number of `65535`, serving as the head of the list. The last free entry links back to object 0. Some free entries may also link back to object 0 without being part of the list.

For instance, the entry `0000000123 00001 f` indicates that the object's definition begins at byte offset 123 in the file. The object has a generation number of 00001, which means it was initially created (generation 00000), then deleted once, and is now marked as free (i.e. available for reuse). When an object is later recreated, its new instance will have its generation number incremented accordingly.

Each **subsection** in the cross-reference table begins with:

```
<starting_object_number> <number_of_entries>
```

where `starting_object_number` is the first object number covered in the subsection and `number_of_entries` is the count of consecutive objects described in the subsection. These two numbers define the **range** of object numbers described in the lines that follow.

Indirect objects in a PDF are defined with a unique object number (and an associated generation number), typically assigned sequentially as they appear in the file. For example, if the PDF contains:

```
1 0 obj
    ...
endobj
2 0 obj
    ...
endobj
```

then objects 1 and 2 are referenced in the cross-reference table with their corresponding byte offsets, enabling quick access to each object.

Example of a Cross-Reference Table

```
xref
0 6   % 6 entries starting from object 0
0000000003 65535 f  % Object 0 (always free)
0000000017 00000 n  % Object 1 (in use)
0000000081 00000 n  % Object 2 (in use)
```

```
0000000000 00007 f  % Object 3 (free)
0000000331 00000 n  % Object 4 (in use)
0000000409 00000 n  % Object 5 (in use)
```

This example shows a single subsection with six entries: four in-use objects (1, 2, 4, 5) and two free objects (0, 3). Object 3 has been deleted and will be assigned 7 as generation number when reused.

Example with Multiple Subsections

```
xref
0 1   % 1 entry starting from object 0
0000000000 65535 f  % Object 0 (always free)
3 1   % 1 entry starting from object 3
0000025325 00000 n  % Object 3 (in use)
23 2   % 2 entries starting from object 23
0000025518 00002 n  % Object 23 (in use)
0000025635 00000 n  % Object 24 (in use)
30 1   % 1 entry starting from object 30
0000025777 00000 n  % Object 30 (in use)
```

Here, the table consists of multiple subsections, each containing entries representing a range of objects.

Starting from PDF 1.5, cross-reference tables may be replaced with **cross-reference streams**, which store this information in a compact binary format.

- **Trailer**: the trailer of a PDF file is essential for applications to quickly locate the cross-reference table and other critical objects. The trailer's structure is as follows:
  - `trailer` keyword, followed by the trailer dictionary.
  - Trailer Dictionary (`<< ... >>`):
    - `/Size`: (Required) total entries in the cross-reference table.
    - `/Prev`: (Conditional) byte offset to previous cross-reference section.
    - `/Root`: (Required) indirect reference to the document's catalog.
    - `/Encrypt`: (Conditional) encryption dictionary.
    - `/Info`: (Optional) indirect reference to the document's information dictionary.
  - `startxref` keyword, followed by the byte offset to the last cross-reference section's `xref` keyword.
  - `%%EOF`: End-Of-File marker.

  The trailer acts as a directory at the end of the PDF file, providing pointers for navigation and interpretation.

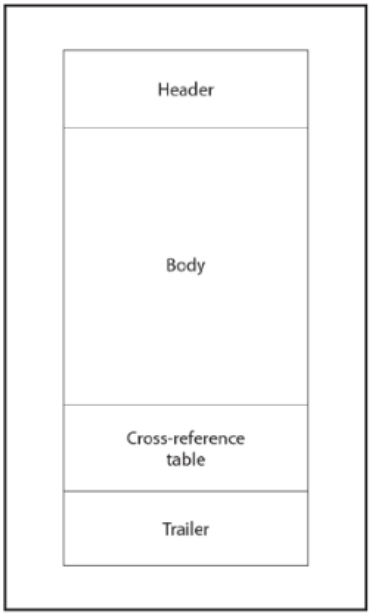

Figure 1: Initial structure of a PDF file

4.2.3 Document Structure

A PDF document can be described as a hierarchy of objects contained in the body section (see Section 4.2.2) of a PDF file.

The structure is built upon a root object, which is the **document catalog** and acts as the central control point. The catalog in a PDF file serves as a guide for the PDF reader. It directs the reader to the key elements like, for example, page tree, which is essential for rendering the document's pages. The **page tree** defines the sequence of pages within the document through a nested structure of page tree nodes and page objects. **Page objects**, the leaf nodes of this tree, contain attributes specific to each page, such as its dimensions, content and associated resources necessary for rendering the pages.

Additionally, the document catalog may reference important dictionaries, such as those related to logical structure (MarkInfo, StructTreeRoot) and security permissions (Perms). It can also include an `OpenAction` entry that specifies an action to be executed when the document is opened, which can point to a JavaScript script via an action dictionary with the `/S /JavaScript` type. Similarly, entries like `AA` (Additional Actions), `AcroForm` (for interactive forms) and annotation objects may also trigger JavaScript, enabling interactive behavior within the document.

These objects typically use an action dictionary with a `/JS` entry that holds the script content, allowing dynamic behaviors based on user interaction or document events.

Example: A minimal action dictionary embedded in the catalog to run JavaScript on document open:

```
<< /OpenAction
   << /S /JavaScript
```

```
      /JS (app.alert("Hello from PDF!");)
   >>
>>
```

Another important entry that may appear in the document catalog is the **/Metadata** key. This entry points to a metadata stream object that contains XML-encoded information following the Extensible Metadata Platform (XMP) specification. This stream allows for the inclusion of structured and standardized metadata describing the document, such as its title, author, language, creation date, modification dates and more.

The referenced stream object typically includes the `/Type /Metadata` and the `/Subtype /XML` entries, identifying it as a metadata container.

Example: A simplified metadata reference from the document catalog:

```
<<
  /Type /Catalog
  /Pages 2 0 R
  /Metadata 12 0 R
>>
```

And the corresponding metadata stream:

```
12 0 obj
<< /Type /Metadata
   /Subtype /XML
   /Length 512
>>
stream
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <!-- XMP metadata here -->
</x:xmpmeta>
endstream
endobj
```

This mechanism offers a flexible way to embed descriptive information within the PDF, supporting features like searchability, indexing and document management.

4.2.4 Content Streams

A **content stream** in a PDF is a sequence of instructions that describe how graphical elements should be painted on a page. These instructions follow the standard PDF object syntax and must be interpreted sequentially. Each page in a PDF has at least one content stream, and content streams can also be used to define reusable graphical elements like forms, patterns, certain fonts and annotation appearances. The instructions within a content stream consist of **operands** and **operators**, where operands provide data, and operators specify actions, such as painting shapes or rendering text. Operators must have their required operands immediately before them.

Since content streams cannot contain indirect object references, any external objects they need, such as fonts or images, must be referenced through a *resource*

dictionary. A resource dictionary maps names to these external resources, ensuring that a content stream can access fonts, images, color spaces, shading patterns, and other graphical elements. In a resource dictionary each resource type is stored in a subdictionary where specific resource names are associated with corresponding PDF objects. This ensures that all resources required by a content stream are clearly defined and accessible within its scope.

4.3 Incremental Updates

Incremental updates allow for efficient modifications to PDF files by appending new data to the end, rather than rewriting the entire document. This method significantly speeds up saving, especially for minor changes. For instance, editing a page or adding an annotation results in new content being appended, with only the modified portions updated (see Figure 2). This is particularly advantageous in scenarios where overwriting the original file is not feasible, such as HTTP transfers.

The process involves updating the cross-reference section, which saves changes to objects, including marking deleted objects without physically removing them. Additionally, the trailer section is updated, containing a 'Prev' entry that links to previous cross-reference versions, enabling PDF readers to navigate through multiple updates. Consequently, a PDF might contain multiple versions of the same object, with the cross-reference section directing the reader to the most recent one. Beginning with PDF 1.4, the document catalog's 'Version' entry can also be updated, facilitating version tracking. In essence, incremental updates provide a fast and efficient way to modify PDFs without altering the original content, ideal for environments requiring frequent and small edits.

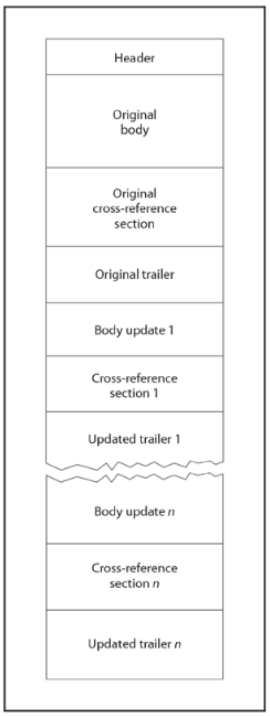

Figure 2: Structure of an updated PDF file

4.4 Encryption in PDFs

Encryption in the PDF format is implemented through the use of a dedicated dictionary structure, enabling document authors to restrict access and define usage permissions. Central to this mechanism is the **Encrypt** dictionary, which is referenced in the PDF file's trailer (see Section 4.2.2). Its presence indicates that the document contents are encrypted and specifies the necessary parameters for interpreting or decrypting the file. The dictionary includes several key entries, such as `Filter`, which defines the **security handler** used for the encryption process (commonly `Standard`), and `V` and `R`, which denote the version and revision of the encryption algorithm, respectively. Other important fields include `Length`, which indicates the encryption key length in bits, and the entries `O` and `U`, which are 32-byte strings representing the owner and user passwords. These are used in combination with the permissions flag `P` to control what a user can or cannot do with the document, such as printing, copying or modifying its content.

The encryption process typically applies to all strings and streams within the file, ensuring that textual content, images and even embedded files are protected. Depending on the document's settings, metadata may also be encrypted. To manage this, PDF encryption uses object-level encryption, where the encryption key is derived by combining a document-level key with the object and generation numbers. This design allows for consistent and secure decryption of each individual object within the file.

The most widely used encryption method is the **Standard Security Handler**, which relies on passwords to differentiate between user access (with limited permissions) and owner access (with full control). When a PDF viewer opens an encrypted document, it reads the `Encrypt` dictionary and attempts to derive the decryption key using the supplied password. The specific decryption algorithm employed depends on the version and revision values, which correspond to different levels of security, from RC4 to AES encryption.

PDF encryption also supports **Public-Key Security Handlers**, offering an alternative to traditional password-based methods. These handlers use different algorithms to compute the encryption key and manage access permissions.

Furthermore, PDF 1.5 introduced `crypt` filters, which allow for finer control over encryption. Crypt filters enable specific streams or parts of a document to be encrypted differently or even excluded from encryption, providing more granular security management.

In general, PDF encryption offers a flexible mechanism for securing content, built directly into the document's internal structure through dictionaries. Using various security handlers, password protection, granular permission settings and crypt filters, it supports both confidentiality and controlled access.

4.5 Metadata in PDFs

PDF documents can include **metadata**, which is information that describes the document itself rather than its visible content. These metadata help identify, categorize and manage PDF files and can be divided into two main categories: **document-level metadata** and **object-level metadata**.

Document-level metadata refers to information that describes the entire PDF file. This may include the title of the document, the author's name, subject, keywords, creation date, modification date and the software used to generate the PDF. Historically, this kind of metadata was stored in a structure known as the *document information dictionary*. However, starting with PDF 1.4, Adobe introduced a more flexible and extensible method for storing metadata: the *metadata stream*, which uses XML formatted according to the Extensible Metadata Platform (XMP) standard. From a structural point of view, the XMP metadata stream is stored as a separate stream object within the PDF file and is referenced by the document catalog (see Section 4.2.3) through the `/Metadata` key.

Object-level metadata, on the other hand, refers to individual components within a PDF, such as images, fonts or embedded multimedia. This allows specific elements to carry their own descriptive information, which can be particularly useful in complex or composite documents.

4.5.1 Example of dual Metadata representation

Consider a PDF document that was created on March 15, 2024, modified on April 1, 2024, authored by "Jane Doe," titled "Climate Data Report," and generated using Adobe Acrobat Pro.

- The **document information dictionary** might contain:

```
<<
  /Title ()
  /Author ()
  /CreationDate (D:20240315120000Z)
  /ModDate (D:20240401113000Z)
>>
```

- In contrast, the **metadata stream** would include a richer XMP packet like the following (simplified for illustration):

```
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
      xmlns:dc="http://purl.org/dc/elements/1.1/"
      xmlns:xmp="http://ns.adobe.com/xap/1.0/"
      xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
      <dc:title>
        <rdf:Alt>
          <rdf:li xml:lang="x-default">Climate Data Report</rdf:li>
        </rdf:Alt>
      </dc:title>
      <dc:creator>
        <rdf:Seq>
```

```
            <rdf:li>Jane Doe</rdf:li>
          </rdf:Seq>
        </dc:creator>
        <xmp:CreateDate>2024-03-15T12:00:00Z</xmp:CreateDate>
        <xmp:ModifyDate>2024-04-01T11:30:00Z</xmp:ModifyDate>
        <pdf:Producer>Adobe Acrobat Pro</pdf:Producer>
      </rdf:Description>
    </rdf:RDF>
  </x:xmpmeta>
```

This example shows how both traditional and modern metadata mechanisms can coexist within a single PDF. While the document information dictionary provides minimal date-related metadata, the metadata stream delivers a more comprehensive and structured description.

4.6 Identification of Watermarks

In the context of the PDF, a **watermark** is a visible graphic or textual element layered over or behind the page content. Watermarks are commonly used to provide additional information about the status of the document, such as 'Confidential,' 'Draft,' or company logos, and are meant to be visible to the reader.

Unlike some other formats, PDF does not define a dedicated object type named `watermark`. Instead, watermarks can be implemented in various ways using standard PDF features:

- *Content stream watermarks*: drawn directly onto the page within the content stream.
- *Annotation-based watermarks*: defined using page annotations, which are layered over the page content. These may include transparency or interactive features.
- *Optional Content Groups (OCGs)*: introduced in PDF 1.5 and standardized in PDF 2.0, OCGs allow certain content, such as a watermark layer, to be dynamically shown or hidden in the viewer.

Even without opening a PDF in a viewer, the presence of a watermark can often be identified by inspecting internal structures such as page content streams, annotations or OCGs. These elements may contain visible text or images labeled as watermarks, or refer to layers selectively rendered by the viewer. Automated tools or custom scripts can scan these object structures for indicators of watermark use.

5 Digital Signatures in PDFs

With the widespread use of digital documents in both personal and professional contexts, ensuring the authenticity and integrity of files has become increasingly important. **Digital signatures** provide a cryptographic solution that allows users to verify the origin of a document and confirm that it has not been tampered after signing. Among various formats, PDFs are widely adopted for contracts, reports and official communications, making them a common target for applying digital signatures. This section introduces the fundamental concepts of digital signatures and explores how they are specifically implemented within PDF documents.

5.1 Understanding Digital Signatures

Digital signatures are a core cryptographic tool used to ensure secure communication, particularly in the contexts of authentication, authorization and non-repudiation [37].

They serve to confirm the identity of the signer and provide proof that the signer cannot deny having signed a specific document or message [6].

Public-key cryptography forms the basis for digital signatures. In order to generate a verifiable signature, the signer must have a **signing certificate** which corresponds to an association between their identity and a particular public key. The certificate is issued by a trusted party, known as **Certificate Authority (CA)**and can be expired, renewed or revoked following the CA verification policy.

Digital signatures are used in two primary operations: (i) **signing** a message using the private key, and (ii) **verification** of that message by using the corresponding public key to verify the authenticity as well as integrity of the signature.

Digital signatures offer three main types of guarantees:

- **Authentication**: ascertains the source of the message, will affirm who sent it.
- **Integrity**: verifies that the content has not been changed since it was signed.
- **Non-repudiation**: guarantees that the signer cannot deny that it had pushed a signature into document.

In addition, digital signatures may optionally contain a **trusted timestamp** from an Time Stamping Authority(TSA). This timestamp evidences the time at which the document was signed, and permits verification of the timestamp even after the certificate of the signer has expired. This is particularly critical for validation over long timescales (as in an archival or legal context).

The particular hash functions, encryption methods and other cryptographic techniques used vary from one digital signature implementation to the other and may depend on an underlying application or system. The maind igital signature schemes are:

- **ECDSA (Elliptic Curve Digital Signature Algorithm)**: a digital signature algorithm that provides good security with relatively short keys, which is desirable for modern application.

- **PKCS#1 (Public-Key Cryptography Standards #1)**: specifies the RSA-based cryptographic signature scheme widely used in PDF digital signatures and other document formats, and thereby lending both compatibility and solid cryptographic assurance.
- **DSA (Digital Signature Algorithm)**: a US federal standard proposed by the NIST relying on discrete logarithms. It is not widely used today, but still available in most cryptographic libraries.
- **EdDSA (Edwards-Curve Digital Signature Algorithm)**: a fast, modern digital signature scheme that provides high security and it is being widely used in secure communication and cryptographic schemes.

5.1.1 The Signing Phase

The **signing phase** is required to create and attach a valid digital signature to the original document.

As shown in Figure 3, this phase consists of the following steps:

1. The document to be signed is retrieved.
2. A hash function is used to create a unique fingerprint of the document.
3. The resulting hash value is encrypted using the **signer's private key**.
4. The result of the encryption is the **signature**, which can be attached to the original document.
5. At this point, the document, composed of the original content and the digital signature, is considered signed and can be sent to the receiver.

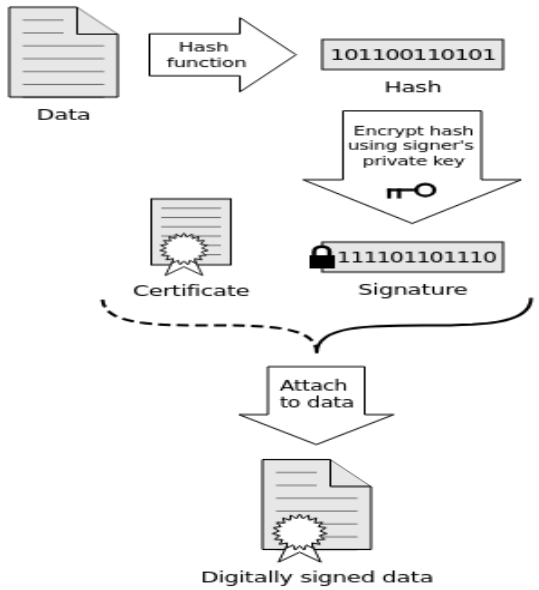

Figure 3: Signing Phase [53]

5.1.2 The Verification Phase

The **verification phase** is necessary to validate the authenticity of the digital signature.

This phase includes the following steps, as shown in Figure 4:

1. The digitally signed data, the original document along with the digital signature, is received.

2. The original document (excluding the digital signature) is hashed, and the resulting value is stored.

3. The digital signature is extracted and decrypted using the sender's public key, which is retrieved through the associated certificate. The result of this decryption is stored.

4. Finally, if the result of the hashing and the result of the decryption match, the verification is successful and confirms authenticity, integrity and non-repudiation.

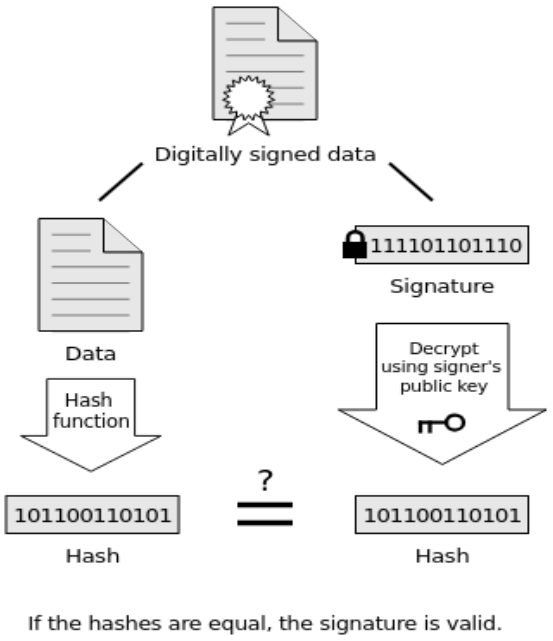

Figure 4: Verification Phase [53]

5.2 Implementation of Digital Signatures in PDFs

Digital signatures in PDF files are implemented using standardized cryptographic mechanisms. These mechanisms rely on well-established standards, particularly the **Cryptographic Message Syntax (CMS)**, also known as **PKCS#7**, as defined

in RFC 2315 [48]. CMS provides a general syntax for data that can be cryptographically signed or encrypted, and it is the core format used to encapsulate digital signature data in PDF documents.

5.2.1 Structure of Public- Key Cryptography Standards #7

PKCS#7 is structured using **Abstract Syntax Notation One** (ASN.1), a formal language that describes data structures for representing, encoding and decoding data independently of programming languages. The encoding of these structures is done using either the Basic Encoding Rules (BER) or the Distinguished Encoding Rules (DER). BER provides flexibility in encoding, while DER imposes stricter rules that guarantee a unique encoding, this determinism is critical for ensuring the integrity of digital signatures during validation.

In PKCS#7,the **ContentInfo** structure is the core element acting as a container to connect a content type with the corresponding value(s). In the context of digital signatures, the content type itself is set to **SignedData**, meaning whatever data is in there has been signed.

```
ContentInfo ::= SEQUENCE {
    contentType ContentType,
    content [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL
}
ContentType ::= OBJECT IDENTIFIER
```

The `contentType` field includes a so-called object identifier (OID) that defines which format the content has, including a `signedData` or an `envelopedData` for instance, whereas the optional content contains the payload itself in some agreed fashion referring to some data or textual representation. The format of this payload depends on the content type, (e.g. in signedData it is the data to be signed and in envelopedData it is the encrypted data). This modularity permits PKCS#7 to support different kinds of content, while preserving a uniform top-level syntax.

When the content type is `signedData`, the associated structure includes all the elements necessary for verifying a digital signature. It encapsulates not only the signed content but also the cryptographic metadata and information about the signers.

```
SignedData ::= SEQUENCE {
    version Version,
    digestAlgorithms DigestAlgorithmIdentifiers,
    contentInfo ContentInfo,
    certificates [0] IMPLICIT ExtendedCertificatesAndCertificates OPTIONAL,
    crls [1] IMPLICIT CertificateRevocationLists OPTIONAL,
    signerInfos SignerInfos
}
```

The `SignedData` structure includes a version number, a list of digest algorithms used by the signers, and a nested `ContentInfo` structure that contains or references the data being signed. Additionally, it may include the signer's certificate

chain (`certificates`), any associated certificate revocation lists (`crls`), and a set of `signerInfos` elements that detail the specific attributes of each signer.

The `signerInfos` field contains a collection of **SignerInfo** structures. Each individual signer is represented using a `SignerInfo` structure, which specifies the identification of the signer, the hash algorithm used and the actual digital signature (the encrypted digest). It may also include optional signed and unsigned attributes that provide additional metadata or functionality.

```
SignerInfo ::= SEQUENCE {
    version Version,
    issuerAndSerialNumber IssuerAndSerialNumber,
    digestAlgorithm DigestAlgorithmIdentifier,
    authenticatedAttributes [0] IMPLICIT Attributes OPTIONAL,
    digestEncryptionAlgorithm DigestEncryptionAlgorithmIdentifier,
    encryptedDigest EncryptedDigest,
    unauthenticatedAttributes [1] IMPLICIT Attributes OPTIONAL
}
```

In this structure, the `issuerAndSerialNumber` field is used to identify the signer by making an indirect reference to the certificate. The `digestAlgorithm` identifies the hash function applied to the content of the document, and optionally is included with `authenticatedAttributes` (e.g. signing time or document hash) before it is encrypted with the private key of a signer. The Encrypt operation is performed, and the result of it is placed in `encryptedDigest`, which is a digital signature, itself. The `unauthenticatedAttributes` field might include additional information, outside the scope of the signature.

5.2.2 Embedding PKCS#7 Signatures in PDF Files

For security in PDF documents, the digital signatures are encapsulated using PKCS#7. This encapsulated structure is included into the PDF by using a special signature mechanism.

When a PDF is signed, a **Signature Dictionary** is included in the document, usually as an interactive form field (whose type is `/Sig`). This dictionary is a bucket of metadata and cryptographic info to hold the signature, and points to a PKCS#7 `SignedData` object.

The **Signature Dictionary** include:

- `/Type`: specifies the type of object that, for digital signatures, usually is `/Sig`.
- `/Filter`: identifies the signature handler that was used to create the signature, such as `Adobe.PPKLite`. A **signature handler** is a software module responsible for managing the signing process, including creating the signature, interfacing with cryptographic libraries and embedding the PKCS#7 data into the PDF structure.
- `/SubFilter`: indicates the format of the signature. Common values include:

  - `adbe.pkcs7.detached` – the signature is stored separately from the signed content. This is the most commonly used method in PDF signatures.
  - `adbe.pkcs7.sha1` – the signed content is hashed using SHA-1 and included within the PKCS#7 container. This method is deprecated.
  - `ETSI.CAdES.detached` – used for qualified electronic signatures, especially under EU eIDAS regulations.

- `/ByteRange`: an array defining the exact byte offsets of the signed content in the file. This ensures that the cryptographic hash, as shown in Figure 5, covers the intended parts of the document and explicitly excludes the signature contents themselves.

- `/Contents`: this field, as shown in Figure 5, stores the DER-encoded and hex-encoded PKCS#7 `SignedData` object, which includes the digital signature, certificate chain and optional attributes.

- `/Reason`: a string indicating the reason for signing the document.

- `/M`: the signing time in PDF date format.

- `/ContactInfo`: optional field to provide the signer's contact information.

- `/Location`: indicates where the signing took place, if applicable.

- `/Name`: the name of the signer or signing authority.

In the case of `adbe.pkcs7.detached`, the digital signature (the PKCS#7 **SignedData** structure) is cryptographically associated with the document contents but does not include a copy of it in the signature container itself. Instead, only a hash (digest) of portions of the document as identified by `/ByteRange` is signed using the private key of the signer.

Which means that the `SignedData` structure holds only the digest and metadata (certificate, signing time, etc.), but not the document content itself at least inside of the signature block. During verification, the signed byte ranges of the document are rehashed and compared with the decrypted digest from the signature.

This approach allows the original content to remain external and unchanged while ensuring integrity and authenticity. The advantage is that it reduces redundancy and avoids embedding large document data directly into the signature structure, which improves efficiency, especially for documents with large size.

PDF signatures can also include a **trusted timestamp** issued by a **Time Stamping Authority (TSA)**. This timestamp is typically included as an *authenticatedAttribute* in the `SignerInfo` structure of PKCS#7. It proves that the document was signed at a specific time and is especially useful for long-term validation (LTV), where the certificate may later expire or be revoked.

By incorporating PKCS#7 signatures in a standardized way, PDF documents ensure better interoperability, security, and alignment with legal standards for digital signatures.

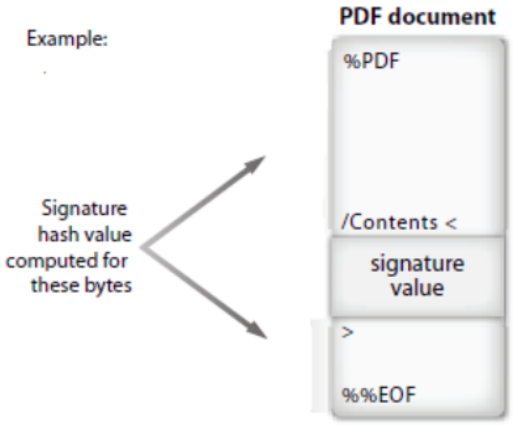

Figure 5: Example of Signature

5.3 Multiple Signatures and Incremental Updates

PDF documents support multiple digital signatures through a mechanism called **incremental updates**, which appends changes to the end of the file rather than modifying the existing content. This feature is fundamental to the secure addition of multiple signatures as shown in Figure 6: each new signature is added as an incremental update, leaving previous content and its associated signatures, untouched and fully verifiable.

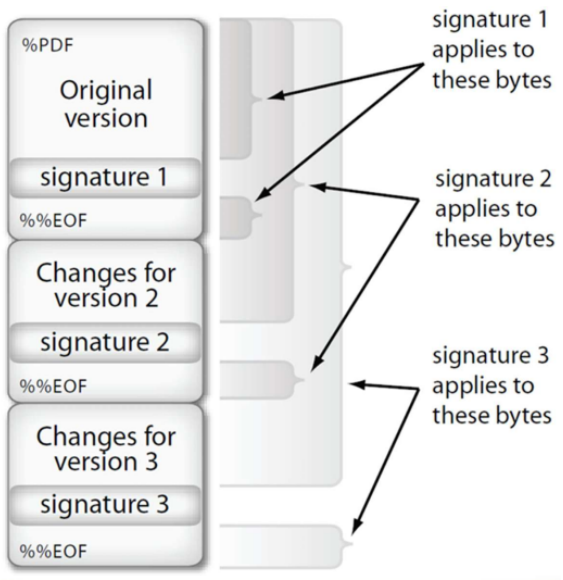

Figure 6: Multiple Signatures and Incremental Updates

Each PDF digital signature only covers the particular byte ranges specified at the time it was signed. This ensures that:

- Previous signatures remain valid even when new ones are added.

- The full history of modifications and signatures can be reconstructed and audited.
- Each signer only takes responsibility for the state of the document as it existed at their time of signing.

With the addition of a second or subsequent signature, a new `/Sig` form field is created and the new structure is added to the end of the file (see Figure 6). The `/ByteRange` of the new signature refers to the document up to this point, i.e. including previous signatures.

This append-only behavior provides the following strong guarantees:

- Signed content is tamper-evident, as any change in the signed data will fail to verify the signature.
- Signature changes are saved directly into the file, tracking and auditing the signing history have never been easier.
- PDF viewers can also display a signature panel displaying the validity of each signature and the details about the signer.

This implementation with PKCS#7 encapsulation and incremental update allows PDFs to be able to support workflows with multiple signers yet still preserve the strong cryptographic guarantees that had been achieved.

5.3.1 Certifying Signatures and Approval Signatures

PDF supports a signature type often referred to as a *certifying signature*, which is typically the first signature applied to a document. This signature designates the document as final or sets restrictions on the types of modifications allowed afterward (e.g. filling out form fields or adding comments).

Although only one certifying signature is permitted, additional signatures, commonly known as *approval signatures* can be inserted, provided that they do not violate the constraints established by the original certifying signer. Approval signatures are commonly used by other participants to indicate that they have read, approved of or explicitly agreed to the document in its current state.

5.3.2 Example of Multiple Signatures

Consider a PDF contract that is signed by two parties, Alice and Bob, sequentially.

1. Alice signs the document:
    - Her signature field, `Sig1`, is added.
    - A `ByteRange` is created to cover the entire document except for the `/Contents` of `Sig1`.
    - A PKCS#7 `SignedData` object is embedded, referencing Alice's certificate and her signature.

- All changes are saved as an incremental update, preserving the original document state.

2. Bob signs afterward:
   - A new signature field, `Sig2`, is created.
   - His `ByteRange` now spans the original document plus the incremental update added by Alice.
   - Bob's PKCS#7 signature covers all content up to the point just before his `/Contents` field.
   - His signature is appended in a second incremental update, preserving Alice's signature untouched.

Both signatures can now be independently verified:

- Alice's signature validates against the original state of the document.
- Bob's signature validates against the document as it existed after Alice signed.

5.4 Extracting and Verifying Signatures

Verifying a digital signature in a PDF involves both structural analysis of the document and cryptographic validation of the signature. If a PDF contains multiple signatures, each one corresponds to a different revision of the document and must be independently extracted and validated.

5.4.1 Signature Extraction

PDF digital signatures are stored in `/Sig` fields, which are embedded within interactive form fields defined in the document's `/AcroForm` dictionary. The extraction process typically involves:

- Locating all form fields of type `/Sig` via the `/AcroForm` entry in the document catalog. The `/AcroForm` is a dictionary that defines interactive form elements in the PDF, including signature fields, text fields and checkboxes.
- For each signature field:
  - Extracting the `/ByteRange`, which specifies the exact byte offsets of the signed portions.
  - Reading the `/Contents`, which contains the DER-encoded PKCS#7 signature.
  - Parsing additional metadata such as `/Name`, `/Reason`, `/Location`, and `/M` (modification time).

Multiple signatures are supported through incremental updates: each new signature appends data to the end of the PDF without modifying previously signed content. Each signature thus references the state of the document at the time it was created, enabling independent validation of each revision.

5.4.2 Signature Verification

The process of verifying a PDF signature includes:

1. Recomputing the hash of the signed byte ranges using the specified digest algorithm (e.g., SHA-256). The hash algorithm used is typically indicated by the `digestAlgorithm` field within the `SignerInfo` (see Section 5.2.1).
2. Decoding and validating the PKCS#7 signature from the `/Contents` field.
3. Comparing the computed hash with the one embedded in the signature, which is decrypted from the `encryptedDigest` field in the `SignerInfo` structure, to confirm the document's integrity.
4. Validating the signer's certificate embedded in the `SignerInfo` structure:
   - Ensuring a valid certificate chain up to a trusted root authority.
   - Checking expiration and revocation status using **OCSP (Online Certificate Status Protocol)** or **CRL (Certificate Revocation List)**.
5. Validating timestamps, if a trusted timestamp is included via a Time Stamping Authority (**TSA**), to confirm the signing time and support Long-Term Validation(**LTV**).

If there are multiple signatures in the document, then each must be validated **independently** with respect to the revision they reference.

Several tools and libraries facilitate the extraction and validation of PDF signatures:

- **Adobe Acrobat**: offers a graphical interface that displays trust status, certificate chains and document revisions associated with each signature.
- **PyHanko**: a modern Python library for digital signatures verification, timestamp validation and even for multiple signatures and long-term validation.
- **iText** and **Apache PDFBox**: two Java libraries used to interact programmatically with PDF signatures and verification.

5.4.3 Example of an Embedded PDF Signature

Below is an example of a signature object embedded in a PDF:

```
/Sig <<
  /Type /Sig
  /Filter /Adobe.PPKLite
  /SubFilter /adbe.pkcs7.detached
  /ByteRange [0 12345 67890 13579]
  /Contents <308206a830820590a0030201...>
  /Reason (Approved for release)
  /M (D:20240406143000+01'00')
```

```
  /Name (John Doe)
  /Location (Berlin, Germany)
>>
```

In this structure:

- The `/ByteRange` indicates which parts of the document are covered by the signature.
- The `/Contents` field holds the PKCS#7 signature, which contains the encrypted digest and certificate chain.
- Signature objects allow for additional human-readable fields (like an `/Reason`, or a `/Name` and `/Location`) to provide context about a signature.

A document can include any number of these structures, which represent individual signing events and document versions. PDF viewers and verification tools use this information to determine the validity and integrity of each individual signature.

6 Emails and Posta Elettronica Certificata (PEC)

Electronic communication is fundamental to modern society. Among its many forms, **emails** represent a cornerstone of both personal and professional interaction. In Italy, the **Posta Elettronica Certificata (PEC)** provides a legally recognized, certified email system that supplements traditional email with enhanced legal guarantees. Both emails and PEC are critical sources and subjects of digital forensic investigations, relevant in fraud, phishing, data breaches and legal disputes. A thorough understanding of their structures, mechanisms, and vulnerabilities is essential for forensic experts.

6.1 Emails

Email, or Electronic Mail, is a fundamental technology of digital communication, enabling the exchange of messages across distributed networks. Its architecture follows a client–server model combined with a store-and-forward mechanism, ensuring reliable message delivery even when recipients are temporarily unavailable.

6.1.1 Core Architectural Components

The email system is composed of several interacting components that collectively manage message creation, transmission and retrieval.

- **Mail User Agent (MUA)**: the application used by end users to compose, send, receive and organize email messages. Common examples include Gmail, Microsoft Outlook and Mozilla Thunderbird. The MUA provides the user interface and formats messages with appropriate headers and body content ensuring consistency.
- **Mail Submission Agent (MSA)**: acts as an intermediary between the MUA and the Mail Transfer Agent (MTA). It verifies and queues messages before they are sent, ensuring compliance with transmission standards and error-free delivery preparation.
- **Mail Transfer Agent (MTA)**: responsible for routing and transmitting emails between mail servers across domains using the Simple Mail Transfer Protocol (SMTP) [33]. Each transfer between MTAs is recorded in the message header, allowing traceability of the email path.
- **Mail Delivery Agent (MDA)**: receives messages from the MTA and stores them in the recipient's mailbox. It ensures that messages are correctly placed in user-specific storage and made accessible through retrieval protocols such as IMAP [13] or POP3 [40].
- **Message Store**: the persistent storage system that maintains emails until they are accessed by the recipient. It supports functions such as folder management, search and synchronization across multiple devices.

The transmission, storage and retrieval of email messages rely on a set of well-defined protocols that operate across different stages of communication:

- **Simple Mail Transfer Protocol (SMTP)**: the primary protocol for sending messages between clients and mail servers or between servers themselves [33]. It typically operates over ports 25, 465, 587 or 2525 and supports secure transmission through TLS/SSL encryption.
- **Internet Message Access Protocol (IMAP)**: enables users to access and manage emails directly on the mail server, allowing synchronization across multiple devices and clients [13]. IMAP operates on port 143 for unencrypted connections and port 993 for encrypted communications.
- **Post Office Protocol Version 3 (POP3)**: used to fetch email from a mail server onto a user's client computer [40]. Unlike IMAP, POP3 generally downloads and removes messages from the server after retrieval. It uses port 110 for unencrypted traffic and port 995 for encrypted (SSL/TLS) communications.

These components and protocols together comprise a resilient infrastructure that guarantees the transmission, delivery and retrieval of messages over distributed networks.

6.1.2 Email Message Structure

Every email message is divided into two major components: the header and the body. These components are defined by Internet standards such as RFC 5322 [46] and RFC 2045–2049 [24, 25].

The **header** is an organized chunk of metadata at the top of each message. This is used to carry crucial routing and authentication information. Common header fields include:

- **From**: the sender's email address.
- **To, Cc, Bcc**: the primary, carbon copy, and blind carbon copy recipients.
- **Date**: the timestamp of sending the message.
- **Subject**: the title or subject of the message.
- **Message-ID**: a globally unique identifier assigned by the sender's system.
- **Received**: a list of entries recording the mail servers that the message traversed.
- **Return-Path**: the address to which delivery errors or bounces are sent to.
- **Authentication Fields**: fields such as *DKIM-Signature*, *SPF*, and *Authentication-Results*, which indicate domain authentication and message integrity status [34, 32, 35].

The header serves as a critical element for message delivery and for subsequent forensic or security analysis.

The **body** of the email contains the actual content of the message, which may include text, HTML-formatted content and attachments. Modern email messages utilize the Multipurpose Internet Mail Extensions (MIME) standard to handle different content types and encode file attachments [24, 25]. The different types of body content are:

- **Plain Text**: unformatted text suitable for simple communication.
- **HTML**: richly formatted content that may include images, links and styling.
- **Attachments**: files of any type encoded and included as distinct MIME parts.

A **blank line** separates the header and body, and their proper structure ensures compatibility between mail clients as well.

6.2 Forensic Considerations in Email Analysis

Email forensics involves the examination and interpretation of email artifacts to verify authenticity, trace message origins and detect tampering or malicious activity. A systematic forensic investigation typically focuses on the following aspects:

- **Header Analysis**: by examining the sequence of `Received` fields, investigators can reconstruct the transmission path of the message, identify the originating IP address and detect anomalies such as forged or manipulated headers.
- **Authentication Verification**: analysis of authentication mechanisms such as SPF, DKIM and DMARC helps confirm whether the sender's domain legitimately authorized the message, thereby identifying spoofing or phishing attempts [34, 32, 35].
- **Timestamp and Routing Consistency**: discrepancies between message timestamps on relay servers can indicate a manipulation of the message.
- **Detection of Tampering**: missing or malformed headers, irregular message formatting and inconsistencies between header data and message content may indicate tampering or the use of nonstandard email tools.
- **MIME Structure Inspection**: examination of the MIME structure ensures that message bodies and attachments correspond accurately to header information and have not been altered post-transmission [24, 25].

A full-featured forensic analysis of email headers and bodies offers insight into the veracity of a message, facilitating technically involved and legal investigations.

6.3 Posta Elettronica Certificata (PEC)

The **Posta Elettronica Certificata (PEC)** is a legally recognized certified email system used in Italy to ensure that electronic communications have the same legal validity as registered postal mail with proof of delivery [50, 44]. It was introduced by the *Decreto del Presidente del Consiglio dei Ministri (DPCM) 2 novembre 2005* and is currently governed and supervised by **AgID** (Agenzia per l'Italia Digitale).

The PEC system extends the traditional email model by integrating cryptographic mechanisms, certified delivery receipts and mandatory logging, thereby guaranteeing **integrity, authenticity and non-repudiation** of messages.

6.3.1 PEC Architecture and Transmission Flow

The PEC infrastructure is composed of certified providers (*Gestori PEC*), users' clients and legally binding receipt mechanisms. It operates through the following entities:

- **Sender's PEC Provider**: it receives the message from the sender's client, checks its structure and creates a *"ricevuta di accettazione"* (acceptance receipt) digitally signed with the provider's certificate.
- **Transport Network**: uses standard SMTP transmission [33] within a closed, certified domain environment. Each message is encapsulated in a secure container ensuring cryptographic integrity.
- **Recipient's PEC Provider**: verifies the digital signature and integrity of the message, then issues a *"ricevuta di avvenuta consegna"* (delivery receipt), also digitally signed.
- **Recipient's PEC Client**: retrieves the message and receipts via IMAP or POP3 protocols [13, 40], with all metadata and digital envelopes preserved.

During transmission, PEC providers log every transaction (including timestamps, message IDs and cryptographic hashes) in their internal systems, maintaining these logs for a minimum of 30 months [5]. Each stage is cryptographically verifiable, ensuring end-to-end traceability.

6.3.2 Message Composition and Legal Receipts

A PEC message is built upon a standard email structure [46] and extended with legal metadata. It consists of:

- **Original Message**: the content created by the sender, formatted as a MIME email [24, 25].
- **Transport Envelope**: a cover digitally signed by the sending provider with the original message as an attachment called `postacert.eml`.
- **Receipts**:

- *Ricevuta di accettazione*: indicates that the sender's provider has accepted the message.
- *Ricevuta di avvenuta consegna*: acknowledges that the recipient's provider has successfully delivered it.
- *Ricevuta di mancata consegna*: issued in the event of failure specifying the cause for non-delivery.

Each receipt is an email that has been digitally signed using X.509 certificates and sent as a separate PEC message. Both acceptance and delivery receipts are definitive legal evidence that the message was indeed dispatched as well as received.

6.3.3 Cryptographic Structure and Verification

PEC relies on asymmetric cryptography from **Public Key Infrastructure (PKI)** for authenticity and integrity. Certified providers hold digital certificates issued by accredited Certification Authorities (CA), and every receipt or envelope is signed with these credentials. For forensic verification:

- The digital signature of each envelope (`.p7m`) requires verifying with its related CA chain.
- The internal message (`postacert.eml`) should be extracted and analyzed separately.
- Timestamps in the signature metadata need to be consistent with those present in provider logs and message headers.

All these artifacts collectively prove message integrity, sender authenticity and the sequence of transmission events.

6.3.4 Forensic Workflow and Evidentiary Value

From a forensic perspective, PEC communications are very reliable evidence sources, because there is a recorded and cryptographically signed step of each transaction. A typical forensic procedure includes:

- **Acquisition**: acquiring the whole PEC message (`.eml` or `.p7m`) keeping original signature and metadata.
- **Verification**: checking certificate and signatures via trusted CAs; comparison of message hashes.
- **Timeline Reconstruction**: reconstructing a provable chain of communications by cross-referencing message headers, service receipts and log entries.

As PEC having legal certified time stamps and provider signed evidence, It is acceptable as digital proof of communication. But the messages have to be treated very carefully by investigators while maintaining cryptographic integrity and for not destroying legal validity.

6.3.5 Limitations and Interoperability

Despite its robustness, PEC has some constraints:

- It is **limited to Italy** and a few compatible European systems (e.g. eIDAS-compliant certified delivery services).
- PEC addresses can be handled only by accredited suppliers of AgID.
- Interoperability with non-PEC systems (e.g. Gmail, Outlook.com) may result in loss of legal certification.

To summarize, PEC is a reliable legal communication infrastructure built upon widely used email standards with added cryptographic protection, and it has been conceived as the new channel for administration-digital ready services in Italy.

7 PDF Attack Techniques and Forensic Analysis

7.1 PDF Metadata

PDF documents often contain embedded metadata that can reveal critical information about their origin, structure and usage history. From a digital forensic perspective, metadata can serve as a valuable source for establishing timelines, attributing authorship, identifying suspicious activities and detecting potential cyber threats.

Several types of PDF metadata exist:

- **Document Information Dictionary**: that contains standard fields, such as `Title`, `Author`, `Subject`, `Keywords`, `CreationDate` and `ModDate`. This information can often be seen in PDF readers, or extracted with forensic tools.
- **XMP Metadata Streams**: PDFs may include XML-based metadata using the eXtensible Metadata Platform (XMP), providing more detailed and structured information, including custom schemas and XML namespaces (unique identifiers, usually a URL, that qualify the names of elements and attributes in an XML document to avoid naming collisions) added by editing software.
- **File System Metadata**: timestamps from the file system (creation, modification, access), associated with each file, provide context about the file on a host machine and help correlate activity on that system.
- **Embedded Content Metadata**: metadata embedded in files (e.g., images, documents, multimedia) could contain application-specific information (EXIF in images, office document properties), which can also help attribution or detection.
- **Object Structure Metadata**: inside a PDF document object structure, we may find entries like `/OpenAction`, `/AA` (used to specify event-driven actions other than those in `/OpenAction`) or `/JS` which allows JavaScript execution on opening, each indicating potentially malicious actions.

Each of the above types of metadata can be useful for forensic investigators in different ways:

- **Establishing Timelines**: comparing timestamps from various artifacts, such as the Document Information Dictionary (e.g., `/CreationDate`), XMP metadata, and file system timestamps show discrepancies that might indicate tampering, repackaging or staged file creation.

  XMP metadata is often considered the most reliable, as it is typically generated and updated automatically by editing software and is harder to modify casually. In contrast, the Document Info Dictionary can be easily edited using PDF manipulation tools and file system timestamps may change simply due to copying or moving the file.

  For example, a PDF may have a `/CreationDate` of `2024-12-10T15:30:00Z`, an XMP creation date of `2023-11-01T11:00:00Z` and a file system creation

timestamp of `2025-01-05T08:20:00Z`. This discrepancy could indicate that the document was modified or backdated to mislead investigators about its origin or activity timeline.

- **Attribution**: metadata fields such as author name, last editor, language settings and originating software can provide leads to identify the document's creator or point of origin.
- **Malicious Intent Detection**: signs can come in the form of documents generated by suspect or anonymizing software, the presence of embedded script or metadata which does not match system logs.
- **Reconstructing Document History**: changes to metadata fields between different versions of the same file (that may have been versioned through incremental updates) or suspicious addition of a new objects (e.g. embedded executable) can be exploited in order to reconstruct the file history and identify unauthorized alterations.

  For instance, an investigator may determine that a prior version of the PDF (which is stored in an incremental update) does not include JavaScript; however, a subsequent appended object contains a `/JS` entry with obfuscated code. Furthermore, a modification in the `Producer` field from version to version (from "Microsoft Word" to "PDF Editor Pro") could make us suspect that the document was modified by external tools not originally used in the creation of the document, meaning it has been manipulated.
- **Localization Clues**: metadata may contain information such as language settings, time zone offsets and system locale, which can help profile the source environment or infer the regional origin of the document or threat actor.

  For instance, working on a document in an application in `ru-RU` locale with a time zone of Moscow can suggest the document was made somewhere where Russian is spoken.

Malicious actors may attempt to manipulate or erase metadata to evade detection. Common techniques include:

- Manual editing or removal using specialized on purpose tools.
- Forging timestamps to match file creation with legitimate activity.
- Metadata hiding either through obfuscation or definition in an encoded form to evade analysis tools.

Analysts are forced to cross-reference multiple levels of metadata and file attributes to determine if a piece of content was authentic or tampered with.

7.1.1 Real-World Example

PDF-based attacks often exploit vulnerabilities via embedded scripts or malicious payloads. For instance, the `CVE-2013-3346` [42] exploit used embedded JavaScript to trigger a vulnerability in Adobe Reader, allowing remote code execution when the file was opened. In such cases, forensic analysis of metadata and structure, such as the presence of suspicious `/OpenAction` or `/JS` entries, was key to detecting and understanding the attack vector and tracing its origin.

7.2 Embedded JavaScript

One of the most common techniques used in malicious PDFs involves embedding JavaScript code directly into documents. Attackers utilize the PDF specification's support for scripting to automatically execute code when the document is opened or interacted with. This is typically achieved via entries such as `/OpenAction`, `/AA`, and `/JS` (see Section 4.2.3).

Embedded JavaScript can be abused in several ways:

- **Redirection to Malicious Websites**: scripts can automatically open a browser window or silently send HTTP requests to a malicious server, leading the victim into phishing pages or drive-by download websites, which are malicious or compromised websites that automatically download and often execute harmful software onto a visitor's device without the user explicitly agreeing or even realizing it.

- **Exploitation of PDF Viewer Vulnerabilities**: by generating JavaScript payloads, an attacker can take advantage of memory corruption vulnerabilities (such as buffer overflows or use-after free bugs) in the PDF reader's JavaScript implementation. These exploits can lead to the execution of arbitrary code on the victim's system.

  Some examples are:

  - **Adobe Reader - CVE-2008-2992**[41]: a vulnerability in the JavaScript function `util.printf()` allows an attacker to trigger a stack buffer overflow using a format string like `"%5000f"`. This can overwrite exception handlers and redirect execution flow to attacker-supplied code.
  - **PDFium (Chrome) – CVE-2015-1282**[43]: use-after-free issues in functions such as `Document::delay` enable potential denial of service or unspecified impact via maliciously crafted PDF files.

- **Automatic File Downloads & Execution**: an malicious Javascript code could download additional payloads (e.g. trojans, ransomware) from remote servers. In certain instances, the attackers use vulnerabilities to escape from any sand box in order to save and run the malware locally.

- **Data Exfiltration**: scripts can be used to capture system information, enumerate installed software or read cached credentials and then send this data to an attacker-controlled server.

- **Social Engineering via Interactive Forms**: some PDFs can have form fields or pop-up dialog boxes enable using JavaScript which makes the user enter sensitive information (such as, for example, passwords and bank details), which is then sent to the attacker.

Because JavaScript execution in PDFs is often hidden from the user and triggered by document events (e.g. opening, scrolling, clicking), these attacks can be highly effective and difficult to detect without proper security controls. Analysts must examine suspicious object entries and extract JavaScript code using specialized tools. Carefully analyzing the JavaScript and observing its behavior are crucial for understanding the PDF's true intent. Comparing it with known vulnerabilities (CVEs) and looking at when the JavaScript was added through incremental updates can also help with attribution and detection.

7.3 Embedded Files and Launch Actions

The PDF format allows to embed files of different types (e.g. executables, Office documents, scripts or other PDFs) with the `/EmbeddedFiles` dictionary. Attackers frequently took advantage of this feature to hide their malicious payloads in what otherwise appeared to be safe documents, essentially transforming an innocent-looking PDF into a toolkit for malware delivery. The `/Launch` action allows to open the embedded file when opening the pdf or clicking on specific elements (in a button or a link, for example).

In terms of standards, if PDF/A-compliant PDFs would only be allowed to embed other PDF/A files for archival purposes, with PDFA-3 the possibility is given to include any sort of file type into a document. While this has legitimate uses when attaching items like invoices or spreadsheets, it's also quite dangerous: if he wants, an attacker can insert an executable, a macro-enabled Office document or a trojanized archive filled with viruses and wrap the PDF around all of them as delivery medium.

In the real world, malicious PDF files often pack a few techniques together, such as JavaScript opening a `/Launch` action automatically or unpacking the embedded payload at runtime and running it. So it is important to examine the `/EmbeddedFiles` dictionary and any launch actions of them can be helpful in finding hidden flaws. Relating these discoveries to known news such as CVEs or possible exploit techniques improves attribution of threat and helps preventing attacks. Researches can extract any embedded objects and filter their hashes for malware scanning. Looking for `/Launch` entries, uncommon MIME types, or excessively large/obfuscated objects can help to uncover delivered hidden payloads. Analysts can sometimes figure out when an object was added by comparing timestamps and versions.

7.4 Exploitation of Reader Vulnerabilities

Attackers often craft PDFs to exploit vulnerabilities in specific reader software versions. These vulnerabilities can be triggered by various malformed or specially

crafted objects within the document, potentially leading to memory corruption or arbitrary code execution. Key exploitation techniques include:

- **Malformed Objects**: PDF readers parse complex object structures such as dictionaries, arrays or streams. Attackers may craft objects with unexpected types, sizes or references to trigger heap or stack memory corruption. For example, the Adobe Reader and Acrobat vulnerability (CVE-2010-2862), caused by an integer overflow in the "CoolType.dll" module when processing a PDF containing a TrueType Font (TTF) with a malformed "maxComponentContours" field in the "maxp" table, could allow an attacker to crash the application or execute arbitrary code if a user opens a specially crafted PDF.

- **Corrupted Image Streams**: PDFs often embed images (JPEG, JPEG2000, JBIG2) that are decompressed by the reader. Vulnerabilities in image decoding libraries can be exploited by embedding malformed image data. For example, CVE-2016-0936 is a memory corruption vulnerability in Adobe Reader and Acrobat (versions before 11.0.14) on Windows and OS X. Attackers can exploit this by embedding specially crafted JPEG 2000 data within a PDF. When the document is opened, the reader's JPEG 2000 decoder mishandles the data, leading to memory corruption. This can result in arbitrary code execution or a denial of service (application crash).

- **Malformed Annotations**: PDF annotations (for example, text, links and buttons) are interactive objects which may also include appearance streams and JavaScript actions. However, the vulnerabilities come in when the reader is unable to properly validate these fields. A use-after-free vulnerability in the way that Adobe Acrobat/Reader handled annotation objects was used by CVE-2020-9715, which could result in remote code execution, for example.

- **Stream Parsing Bugs**: as seen before, PDF files are composed of objects, some of which are stored in streams and use special markers such as `startxref` (indicating the byte offset of the cross-reference table) and `%%EOF` (marking the end of the file). PDF readers rely on these structures to locate and parse objects correctly. Vulnerabilities can occur when a reader fails to properly handle malformed or maliciously crafted streams or markers. This can lead to issues such as buffer overflows, invalid memory dereferences or crashes. A concrete example is CVE-2019-14267, which affected PDFResurrect 0.15. In this case, specially crafted PDF files containing malformed `startxref` and `%%EOF` markers triggered a buffer overflow during parsing. Exploiting this flaw could allow an attacker to crash the application or potentially execute arbitrary code.

It's important to be able to reverse-engineering the structure of a PDF and then compare it against known signatures of vulnerabilities (such as from CVE databases) so that investigator can identify the attack vector utilized. Analysts frequently use controlled environments (e.g., sandboxes) to observe crash behavior and examine object streams for irregularities. Also it makes sense to need for validation and checking the PDF substructure such as byteocde of objects, cross-reference tables,

length of stream and operators sequence. Proper validation ensures that malformed structures or malicious instructions are detected before execution, reducing the risk of triggering vulnerabilities during analysis and improving the accuracy of identifying exploitation mechanisms.

7.5 Phishing and Social Engineering

Some PDF documents are designed to deceive users by mimicking legitimate websites, brands or services. They may include fake login forms, fraudulent account notifications or malicious hyperlinks and QR codes intended to steal credentials or redirect users to phishing sites. These documents often exploit trust and familiarity, making it easier for attackers to manipulate users into revealing sensitive information.

Analysts investigate such PDFs by extracting and examining embedded links, form fields and interactive elements. By cross-referencing URLs and domains with threat intelligence databases, they can determine whether a link or resource is associated with known malicious activity. Metadata within the PDF, such as author information, templates or localized content, may also provide clues about phishing campaigns, including reused designs or targeted geographic regions. Understanding these patterns helps defenders anticipate social engineering tactics and improve detection of malicious PDFs before they reach users.

7.6 Obfuscation and Evasion Techniques

Malicious PDFs often employ a variety of techniques to evade detection and hinder forensic analysis. Attackers may use obfuscation methods such as stream compression, hexadecimal or Base64 encoding, object splitting, and deep object nesting to hide malicious code. Incremental updates can also be abused to conceal changes, allowing attackers to modify a PDF without altering its apparent structure. In addition, encryption or password protection may be used not only for legitimate purposes but also as a way to prevent antivirus scanners or analysts from inspecting the document's contents. These techniques make it difficult to detect or analyze malicious behavior. Obfuscation hides scripts or payloads across multiple objects, while encryption can prevent automated tools from inspecting the file, enabling attackers to deliver malicious content safely.

Analysts normalize the PDF structure with specialized tools (such as `qpdf`, `pdfwalker`) to deeper inspect PDF. High entropy objects, encoded streams or unusually nested objects are marked as suspicious. The decompression and decoding of streams is necessary to expose obfuscated scripts or payload contained within. Document version comparison, particularly when the updates are incremental, can expose hidden manipulations. Decryption efforts are also performed for encrypted or password-protected files, if a key is known or can be estimated. Examining the encryption dictionary and flags helps determine the level of protection. Contextual information, such as how the PDF was delivered (e.g. via email), assists in assessing the intent behind the protection.

7.7 Attacks on Digitally Signed PDFs

While digital signatures are intended to ensure authenticity and integrity of PDF documents, attackers have developed techniques to bypass or abuse these mechanisms.

7.7.1 Signature Wrapping and Incremental Update Abuse

PDFs support incremental updates, allowing new content to be appended without modifying the original signed content. An attacker may leverage this functionality to add malicious objects or scripts after signing, while still apparently having a valid signature in certain viewers.

For example, an attacker may append a malicious page with altered payment instructions to a signed invoice. If the viewer fails to properly verify the signature against the full file, the signature may still appear valid while the user sees modified content.

A real-world example of this type of attack is CVE-2018-18689[49], which affected multiple PDF viewers including Foxit Reader and PhantomPDF. In this vulnerability, attackers could manipulate the '/ByteRange' and cross-reference (xref) tables to insert malicious content into a signed PDF without invalidating its signature. This allowed unauthorized modifications to go undetected if the PDF viewer did not properly verify that the signature covered the entire document.

Investigators should analyze all incremental updates using tools to detect appended objects and verify which byte ranges are covered by the signature to ensure that the signed portions of the document have not been tampered with.

7.7.2 Exploiting Certification Signatures

PDFs can have two primary kinds of digital signatures: *approval* and *certification*. Certification signatures are intended to verify that the document is indeed authentic and intact, as well as to describe what types of alterations are permitted post-signing. That makes them more flexible than approval signatures, but also subject to abuse.

The function of certification signatures is controlled by two main fields:

- **/DocMDP (Document Modification Detection and Prevention)**: it specifies the modification permissions for the certified PDF. There are three common permission levels:
  1. Level 1 – Do not make any changes; if any modify is made, the signature becomes invalid.
  2. Level 2 – Only form filling is allowed.
  3. Level 3 – Form filling and adding annotations are allowed.
- **/Perms (Permissions Dictionary)**: Specifies which signatures exist in the document, what they cover and what changes are considered valid under the certification rules.

A certified PDF, for instance, may allow annotations (Level 3). An attacker could insert an invisible note with Javascript callback. As this change is permitted by the certification settings, in the viewer, the signature would still be valid and it wouldn't matter if harmful code has now been added to that file.

Investigators also should consult the `/DocMDP` and `/Perms` entries to learn what modifications are allowed. All new objects and annotations are to be carefully inspected; hidden components or scripts are not uncommon and may be indicative of compromise.

7.7.3 Viewer Parsing Bugs and Shadow Attacks

Even when a PDF is fully signed, vulnerabilities in PDF viewers can allow attackers to display or execute content not covered by the signature. This is often related to how PDF signatures define their coverage and how viewers interpret it.

As a reminder, when a PDF is signed, the signature covers a specific *byte range* defined in the `/ByteRange` entry. This range specifies the exact segments of the file that are protected. Any content outside of this range is not validated by the signature. Attackers can exploit this in two main ways:

- **Incremental updates after signing** (see Section 7.7.1).
- **Pre-existing unused or hidden objects**: a PDF may include objects in the signed byte range that are not yet referred to anywhere in the visible page tree at signing time. An attacker can then use references from such (e.g. with an allowed modification) to force the viewer to show these previously unused objects instead of the original designed content. As these objects were essentially still part of the signed data, a dumb validation would still return 'valid' for the original signature, despite the visual content having changed.

This approach is called the *Shadow Attack*, as recorded in PDF Insecurity Project [39] in 2020. In these attacks, the PDF document is generated to include both a benign version (which will be used during the process of signing) and a malicious version (kept silent until triggered). By swapping references after signing, the attacker shows malicious version to the user as signed version but signature looks valid.

To compare the rendered content with which objects or byte ranges where actually signed, inspectors have to work with several PDF viewers and forensics tools. They should also check incremental updates of newly appended objects and examine the structure of the document for unused or hidden objects that could be activated after signing.

7.7.4 Universal Signature Forgery (USF)

Universal Signature Forgery (USF) is an attack which enables the attacker to create a PDF for which the digital signature looks valid in certain PDF viewers, but is invalid in reality. This abuses differences in the way various PDF viewers implement validation rules for signatures. The attack is based on carefully mixing the PDF structure so that every platforms can choose to see or not see an embedded

signature as a valid signature, and thus bypassing trust policies or legal validation checks.

7.7.5 Certification Attacks

Certification attacks exploit the permissions granted by a PDF's certified signature to perform unauthorized changes while still appearing valid. Two well-known variants are:

- **Evil Annotation Attack (EAA)**: the attacker adds hidden or malicious annotations in a certified document. These annotations should not change anything that is displayed on the page, but can affect how signatures will be interpreted by PDF viewers.
- **Sneaky Signature Attack (SSA):** an attacker adds a second signature in such a manner that it does not go through or avoids being validated. This way it is possible to manipulate the content of the document without invalidating the primary certification signature.

Both attacks rely on subtle differences in how PDF viewers enforce certification permissions, enabling a signed PDF to appear unchanged while actually being modified.

7.7.6 Pre-Signing Compromise

An attacker can insert malicious content or payloads into a PDF prior to digital signing. Upon matching, a signature is effectively masking the malicious content so that they appear to be genuine. An organization may digitally sign an insecure document template that contains embedded JavaScript. When spread, this triggers the malicious payload despite it being signed legitimate.

Analysts have to track down the document's origin, review incremental versions if they're available, examine for odd embedded objects or scripts which could predate the signing.

7.7.7 Format Confusion and Polyglot Attacks

PDFs can be crafted to contain multiple interpretations simultaneously. A "polyglot" file is a single file that can be understood in more than one way by different programs. In other words, the same sequence of bytes in the file can be interpreted differently depending on the program reading it.

As an illustrative example, one can think of a page which visually appears to be normal plaintext if it is opened in a word processor, but dirty hidden data are executed unknowingly when the same is opened with another reader. The file contains both sets of instructions at the same time, but each program only "sees" the part it understands.

The same bytes are interpreted differently in various programs, according to their expected rules:

- A PDF viewer may display harmless content, such as text or images.

- A signature verification tool may check the PDF and see only the signed, benign portion.
- Another program might read hidden or overlapping data streams, potentially triggering malicious behavior.

In this situation, the signature on the polyglot PDF could validate as safe; however, when displayed in a regular viewer, obfuscated content gets executed or otherwise processed maliciously.

Investigators should analyze the raw file structure, check for overlapping or hidden data streams and compare what is signed versus what is actually rendered or executed. Looking at the file with multiple tools and examining it byte by byte can reveal these hidden interpretations.

7.7.8 External Resource Abuse

Signed PDFs may reference external resources such as images, scripts or media. These resources can change over time, allowing attackers to influence the document's behavior without altering the signed content.

For instance, a signed PDF references an external JavaScript file hosted online. The script is updated to deliver malware, bypassing signature validation.

Investigators are required to trace every external link in the PDF and evaluate their integrity. Capture external dependencies at the time of analysis as a means of detecting post-signing changes. One way to possibly do this is store a hash of each external resource (e.g. SHA-256) in the signed document. When you later check the PDF, recompute the hash and compare that to your original: if it differs, something's been changed about the resource even though the PDF hasn't.

8 Acquisition Sources of PDF Files

Digital forensics begins with the correct identification and handling of acquisition sources. In the context of PDF forensics, the acquisition source refers to the original medium or platform from which a PDF file is obtained. This initial step is decisive, since any mishandling, alteration or incomplete collection of the source may compromise the reliability and admissibility of the evidence.

8.1 The Critical Importance of Acquisition Sources

The **acquisition source** can never be just merely a starting point for the investigation but is actually right where bottom lies in all forensic activity. One mistake in collection can render otherwise useful proof inadmissible in court. To reduce these risks, best practices for digital evidence handling typically focus on a number of key principles:

- Maintaining a flawless **chain of custody**, so that everyone who comes into contact with the evidence is accounted for.
- Employing **cryptographic hashing** throughout to validate the integrity of collected files.
- Following recognized **international standards**, such as ISO/IEC 27037, which provide guidelines for identification, collection, acquisition, and preservation of digital evidence.
- Make use of modern solutions, such as **blockchain-based tracking**, that can give an unabridged audit trail for high-value or high-risk investigative work.

In practical terms, this deals with retention of sources in unaltered state, good documentation and reproducible processing. Without that assurance, the legitimacy of forensic examination is undermined.

8.2 Common Sources for PDFs

PDF files can originate from a variety of sources, both legitimate and malicious. In forensic practice, identifying the acquisition source provides context, such as whether a file was distributed intentionally through official communication channels or introduced covertly as part of an attack. The most relevant categories of acquisition sources for PDFs are listed below.

8.2.1 Emails and Certified Email (PEC)

Emails are the most common medium for distributing PDF documents, both in legitimate communication and as vectors of attack. A large proportion of phishing campaigns and malware intrusions exploit PDF attachments, as users are accustomed to receiving invoices, contracts or official letters in this format. Attackers may embed malicious JavaScript, exploit vulnerabilities in PDF readers or disguise executable payloads as documents with double extensions.

In forensic analysis, the email headers, timestamps and transmission path are often as important as the PDF itself. Investigators may need to establish not only what the PDF contained, but also *who sent it*, *when it was received* and whether the attachment was altered in transit.

An Italian and European special case is **Posta Elettronica Certificata (PEC)**. Considering that PEC is legally binding as registered mail, the PDFs sent through PEC generally carry huge probative value in disputes, for instance, administrative and judicial origilling. But this also turns PEC into a sweet candy for attackers: a rogue PDF sent through PEC might be automatically seen as trustworthy by the recipients. Experts in forensic that handle PEC messages must take into account the certified delivery receipts and also the content of PDF attachments, to avoid considering authenticity as an obstacle against malintent.

8.2.2 Websites and Online Databases

Another major source of PDF evidence is the web. PDFs are often hosted on corporate websites, academic repositories or governmental portals. Attackers may compromise these platforms to distribute infected documents, for example by replacing a legitimate policy document with a trojanized version. From a forensic perspective, acquisition in this context involves preserving the webpage, the server headers and the original hosting environment in addition to the PDF itself. This ensures that the context of distribution is documented and verifiable.

8.2.3 Repositories for Testing and Research

For controlled testing and training purposes, forensic practitioners and researchers often may rely on repositories of sample PDF files. One such example is the *format-corpus* collection available on GitHub (see format-corpus), which contains a wide variety of file formats, including PDFs, suitable for validating forensic tools and methods. While these repositories are not direct forensic acquisition sources, they are valuable resources for benchmarking, tool verification and experimentation in academic and professional contexts.

8.2.4 Local Filesystems and Removable Devices

Last but not least, one of the most direct sources of acquisition is the local device of a suspect or target machine. PDFs can be found that are in a user folder, a document folder or in cloud synchronized folders. USB sticks and other removable mediums are also very commonly (if not ideally) used to take PDFs from place to place or even as a vector for spreading infected files. In these cases, the acquisition is done by full disk imaging or targeted forensic copy of digital evidence files so that no metadata is discarded.

9 Development of foredf

In this chapter, which forms the core of the thesis, we introduce **foredf**, our tool whose name is derived as an acronym from **FOR**ensic **E**mail fetcher and PDF analyzer using peep**DF**.

foredf is a forensic-oriented tool designed for the analysis of PDF documents within a fully dockerized environment. By default, all operations run under a non-root user to ensure safer and more controlled execution. It allows fetching PDFs contained in emails and PEC communications, and analyzing them in a safe and reproducible environment. Finally, it supports the generation of information that can later be included in a formal forensic report.

9.1 Docker and Containerization

Docker[17] is a lightweight virtualization platform that allows applications to run inside *containers*, which are isolated environments sharing the host system's kernel. Unlike traditional virtual machines, containers are more efficient and faster to start, making them ideal for deploying and distributing applications consistently across different systems.

Containerization[45] ensures that all the dependencies and configurations required by an application are packaged together, reducing compatibility issues and making the environment reproducible.

Docker Compose[18] is a complementary tool that allows the definition and management of multi-container applications through a single configuration file, typically named `docker-compose.yml`. With Docker Compose, multiple services, such as databases, web servers and analysis tools, can be orchestrated easily, enabling the creation of complex and fully reproducible environments with minimal effort.

In **foredf**, Docker and Docker Compose are used to provide a secure and fully reproducible environment for PDF analysis. By default, all operations run under a **non-root** user to enhance safety and control in case of malicious PDFs or erroneous operations, while allowing users to fetch and analyze PDFs from emails and PEC communications seamlessly.

9.2 Analysis of Peepdf: Strengths and Limitations

Peepdf[20] is a Python-based forensic and security analysis tool designed for the detailed inspection of PDF files, primarily to detect potential malicious content such as embedded scripts or shellcodes. Its key strength lies in its comprehensive analytical capabilities since it can parse multiple PDF versions, decode various encodings (hexadecimal, octal, names), analyze object structures and identify suspicious elements using filters and reference tracing. The integration with PyV8 [26] and Pylibemu [10] extends its capabilities to dynamic JavaScript and shellcode analysis, allowing researchers to deobfuscate code and simulate execution behavior within the same tool. Furthermore, its interoperability with VirusTotal enables quick hash-based malware cross-referencing, enhancing its usefulness in threat intelligence workflows.

Peepdf also supports the creation and modification of PDF documents, including the generation of files containing executable JavaScript or obfuscated strings, which

makes it not only an analysis tool but also a testing framework for exploit development and research. Its flexible operation modes, command-line, interactive shell and batch, make it adaptable to both automated pipelines and manual inspection.

However, despite its robustness, peepdf has some limitations. The project remains relatively static, with limited ongoing development and only partial automation in its JavaScript analysis modules. The absence of a graphical user interface (GUI) restricts accessibility for less experienced users, and certain features, such as embedded PDF analysis or validation of embedded digital signatures, remain unimplemented. Moreover, since it relies on legacy dependencies like PyV8 and Pylibemu, installation and compatibility on modern systems can be challenging. For these reasons, several forks of peepdf have emerged to address some of these issues, including a fork that ports peepdf from Python 2 to Python 3, and another that replaces the legacy PyV8 with StPyV8 [11].

In conclusion, peepdf is a great tool to manually analyse PDF malware and deep structure exploration, not including some of the fresh really cool features you would see in such advanced security frameworks like real-time behavior emulation or full automation.

9.3 Overview: How foredf Works

As illustrated in Fig. 7, from a high-level perspective, **foredf** operates entirely within the host device after being downloaded and initialized. Once set up, the tool can be used in two main ways, which can also be combined for more advanced workflows:

1. Users can manually insert PDF files into the *files* folder, under the *pdfs* one, and analyze them using the enhanced version of *peepdf* integrated into the tool.

2. The application lets the user set up their email or PEC (Posta Elettronica Certificata) inboxes to be connected with the tool and download PDFs and related metadata automatically from them. The unpacked files can then be inspected with the improved *peepdf*.

In both cases, the analysis produces a comprehensive basic report. When the email fetcher is used, the report includes also the related email or PEC metadata; otherwise, it contains only the interaction logs from *peepdf* together with cryptographic hashes of the analyzed files.

Prior to *peepdf* being invoked, **foredf** calculates and retains the hash value for each file and provides a transient forensic copy to the analysis tool. After the analysis, a fresh hash is calculated and compared with it to verify that input file is untouched. This ensures that the original object is not altered during analysis and therefore preserved from a forensic standpoint.

As an additional security measure, the container runs in a non-privileged mode and write all the files from the use of foredf in read-only mode. This ensures that the original data cannot be altered from inside the container, even when files are automatically fetched by the email modules.

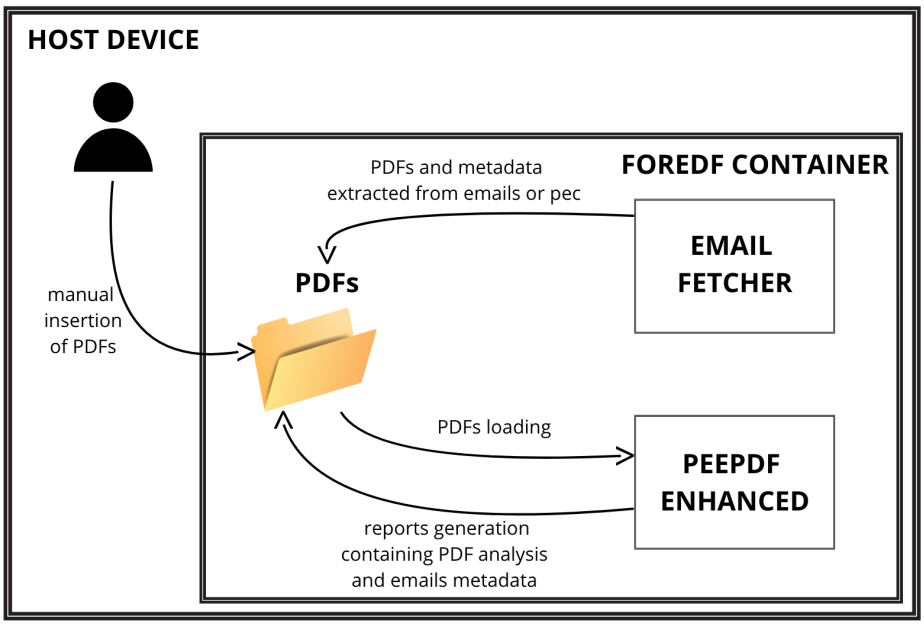

Figure 7: Overview of Foredf

9.4 Internal Structure of foredf

The **foredf** tool is composed of different parts that work closely together. The final goal is to create a forensic tool that aims to minimize possibilities of file alterations while maintaining a clean workflow and supporting forensic experts in doing their work.

9.4.1 Dockerfile and Docker-Compose file

For creating a container with non-root permission, the Dockerfile used is the following:

```
# Base image
FROM python:3.12-slim

ENV DEBIAN_FRONTEND=noninteractive
ENV PYTHONUNBUFFERED=1

# Install system dependencies
RUN apt-get update && apt-get install -y \
    libmagic1 gcc make nano jq \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# -------------------- Copy only files needed for pip
   --------------------
COPY mail-pdf-extractor/requirements.txt
   /app/requirements.txt
```

```
COPY peepdf-3 /app/peepdf-3

# -------------------- Install Python dependencies
    --------------------
RUN pip install --upgrade pip
RUN pip install /app/peepdf-3
RUN pip install -r /app/requirements.txt

# -------------------- Copy the rest of the project
    --------------------
COPY mail-pdf-extractor /app/mail-pdf-extractor
COPY fetch_email /app/fetch_email
COPY peepdf /app/peepdf
RUN chmod +x /app/fetch_email /app/peepdf
ENV PATH="/app:${PATH}"

# -------------------- Non-root user --------------------
ARG UID=1000
ARG GID=1000
RUN groupadd -g ${GID} user && \
    useradd -m -u ${UID} -g ${GID} user && \
    chown -R user:user /app
USER user

CMD ["bash"]
```

This Dockerfile defines a lightweight Python 3.12–based container and installs the required system dependencies (`libmagic1`, `gcc`, `make`, `nano` and `jq`) to support file analysis and script execution. The project files are copied in stages to optimize the build: first the dependencies needed for Python installation, and then the rest of the project scripts.

Python dependencies are installed with `pip`, including `peepdf-3` and any requirements listed in `mail-pdf-extractor/requirements.txt`. Execution permissions are set for scripts like `fetch_email` and `peepdf` to ensure they can run inside the container.

For security and forensics reasons, the container executes as a **non-root user**, no system files are modified and input data is not changed as well. The group IDs (UID and GID) can be set to match the host system's values to prevent permission issues.

After that, a docker-compose file has been used for simplicity in creation and removal. The file is the following:

```
services:
  foredf:
    build:
      context: .
      args:
        UID: ${UID:-1000}
        GID: ${GID:-1000}
```

```
    container_name: foredf
    volumes:
      - ./pdfs:/app/pdfs
      - ./mail-pdf-extractor:/app/mail-pdf-extractor
      - ./peepdf-3:/app/peepdf-3
    env_file:
      - .env
    tty: true
```

In this docker-compose configuration, project directories are mounted inside the container. Moreover, also an `.env` file is included since it is needed for the Email Fetcher (see Sec. 9.4.2). The use of docker-compose also simplifies container management, including building, starting and removing the environment in a reproducible way.

9.4.2 Email Fetcher

The **Email Fetcher** is a Python script named `fetch_email` designed to extract PDFs from IMAP inboxes while ensuring forensic integrity. It verifies PEC signatures, if present, stores metadata, and maintains a read-only environment for all evidence. This section provides a detailed explanation of the script, component by component.

To facilitate its usage inside the container, a wrapper shell script is provided. This allows users to execute `fetch_email.py` from the container's entry point without worrying about the current working directory. By running `fetch_email -h`, the user can view the main help description. Essentially, this script acts as a convenient launcher for the Python program:

Listing 1: Wrapper script to run fetch_email.py from anywhere

```
#!/bin/bash
# Wrapper to run fetch_email.py from anywhere
cd /app/mail-pdf-extractor
python3 fetch_email.py "$@"
```

The main script begins with environment setup and configuration. It loads the configuration from a `.env` file, ensuring that sensitive credentials are not hardcoded:

Listing 2: Environment and Configuration Setup

```
dotenv_path = os.path.join(os.path.dirname(__file__), "..",
    ".env")
load_dotenv(dotenv_path)
# Extract environmental variables from loaded .env
HOST = os.getenv("IMAP_HOST")
USERNAME = os.getenv("IMAP_USER")
PASSWORD = os.getenv("IMAP_PASS")
# Use a trusted CA certificate for validating PEC signatures
PEC_CA_CERT = "AgIDCA1_20210921.pem"
# Find or create folders
# The folder in which PDFs can be manually inserted or
    fetched
```

```
OUTPUT_DIR = "../pdfs/files"
# The folder which receives forensic copies of the files and
    information generated
FORENSIC_FOLDER = "../pdfs/forensic_copy"
# Create folders if not yet created
os.makedirs(OUTPUT_DIR, exist_ok=True)
os.makedirs(FORENSIC_FOLDER, exist_ok=True)
```

First, the code loads the `.env` file to retrieve its variables. This approach allows sensitive credentials to be loaded securely, enhancing both security and usability, as nothing is hardcoded in the script. The `PEC_CA_CERT` variable points to the trusted root certificate used to verify PEC signatures whenever they are present.

Next, the script ensures the existence of two directories:

- `OUTPUT_DIR`, located under the `pdfs` root folder, is intended to store the original PDFs. These PDFs can either be manually inserted by the user or automatically fetched by the Email Fetcher itself. If fetched, these files are written in read-only mode to preserve their integrity.
- `FORENSIC_FOLDER` includes forensic copies of the PDFs and all the generated information (metadata extracted from emails and analytical reports).

By dividing original and forensic data into separate folders, raw evidence is also kept unmodified once processed.

The Email Fetcher are leaning on a couple of small **helper functions** to ensure the code becomes clear and easy maintainable.

The first such function, `compute_hashes`, computes cryptographic hashes for each file. These hashes are then saved in a metadata file (`.json`) located in the forensic folder:

Listing 3: Compute hashes of files

```
def compute_hashes(data):
    return {
        "sha256": hashlib.sha256(data).hexdigest(),
        "md5": hashlib.md5(data).hexdigest()
    }
```

Hashes, as well as metadata files, are generated for each fetched file to ensure the integrity of the entire process, from download to subsequent analysis. They serve as a cryptographic guarantee that the files have not been altered, supporting the forensic chain of custody.

Another vital function for this script is the `save_pdf` function

Listing 4: Save PDFs and emails metadata

```
def save_pdf(filename, data, mail=None, uid=None, prefix=""):
    """Save PDF attachment and create forensic JSON
       metadata."""
    path = os.path.join(OUTPUT_DIR, prefix + filename)
    forensic_json_path = os.path.join(FORENSIC_FOLDER,
       f"{prefix}{filename}.json")
```

```
5
6       # --- Check if file already exists ---
7       if os.path.exists(path) or
           os.path.exists(forensic_json_path):
8           print(f"[*] File or metadata already exists,
               skipping download: {path}")
9           print("[!] If you want to force a fresh download,
               remove the existing file(s) first.")
10          print("[!] Be careful in a forensic context:
               altering or deleting existing evidence may
               compromise integrity.")
11          return
12
13      # Decode base64 if needed
14      if isinstance(data, str):
15          try:
16              data = base64.b64decode(data)
17          except Exception as e:
18              print(f"[!] Could not decode {filename}: {e}")
19              return
20
21      # --- Save PDF ---
22      with open(path, "wb") as f:
23          f.write(data)
24      print(f"[+] Saved PDF: {path}")
25      os.chmod(path, 0o444)  # read-only for everyone
26
27      # Forensic metadata
28      if mail:
29          meta = {
30              "uid": uid,
31              "original_filename": filename,
32              "saved_filename": os.path.basename(path),
33              "content_type":
                   mail.attachments[0].get("mail_content_type",
                   ""),
34              "subject": mail.subject,
35              "from_header": mail.from_,
36              "headers": dict(mail.headers),
37              "hashes": compute_hashes(data)
38          }
39
40          # Detect sender mismatch
41          env_sender = mail.headers.get("Return-Path")
42          hdr_sender = mail.from_[0][1] if mail.from_ else None
43          if env_sender and hdr_sender and
               env_sender.strip("<>") != hdr_sender:
44              meta["sender_mismatch"] = {
45                  "from_header": hdr_sender,
```

```
                "envelope_sender": env_sender
            }

        # --- Save JSON ---
        with open(forensic_json_path, "w", encoding="utf-8")
            as jf:
            json.dump(meta, jf, indent=2, ensure_ascii=False)
        print(f"[+] Forensic metadata saved:
            {forensic_json_path}")
        os.chmod(forensic_json_path, 0o444)
```

Its primary responsibility is to save PDF attachments from emails while simultaneously generating corresponding forensic metadata. The function begins by defining the paths for both the PDF and its associated metadata file. The PDF is saved under the `OUTPUT_DIR`, while the JSON metadata file is stored under `FORENSIC_FOLDER`. An optional prefix can be added to handle nested attachments, ensuring that each file remains uniquely identifiable. This separation of original PDFs and forensic metadata is critical for maintaining integrity and traceability.

Even before writing any files, this function checks if the PDF file (or its metadata) is already present. If any of the files is available, a warning message gets printed and the download is skipped, so as to not overwrite by mistake. Especially in a forensic setting, any modification to the evidence gathered before could harm the investigation.

The function also cover cases of email attachments which could be encoded in **base64**. If the attachment is a string, it attempts to decode it; if decoding fails, it logs an error and safely skips the file. Once the decoding is done, the PDF is saved in binary so nothing of the original content get lost. File permissions for evidence are set to read-only for all users immediately after writing so the evidence cannot be manipulated once it has been written in its proper location.

If the email object is provided, the function generates detailed forensic metadata. This also includes the email UID, original/saved filenames, mime content type, the subject & sender of the email as well as full headers and cryptographical hashes (SHA-256 and MD5) of the stored PDF. The utility checks metadata for sender discrepancies by comparing envelope sender value to the header sender (From). Any variance is logged as an **alert to spoofing or tampering**.

Finally, metadata is saved in a JSON file using UTF-8 encoding and with read-only permissions. In this way, both the evidence and metadata linked to it are not modified. Overall, the `save_pdf` function guarantees integrity, authenticity, traceability and security, making it a fundamental component of the Email Fetcher's forensic workflow.

Another useful function is the `verify_p7m`:

Listing 5: Verify the .p7m signatures

```
def verify_p7m(p7m_file, PEC_CA_CERT):
    try:
        subprocess.run([
            "openssl", "smime", "-verify",
            "-in", p7m_file,
```

```
            "-CAfile", PEC_CA_CERT,
            "-out", "/dev/null"
        ], check=True, stdout=subprocess.PIPE,
           stderr=subprocess.PIPE)
        return True
    except subprocess.CalledProcessError:
        return False
```

The `verify_p7m` function is responsible for verifying the authenticity of PEC messages that are digitally signed in the `.p7m` format. These files encapsulate either a signed PDF or an entire email message, providing legal and forensic assurance of the sender's identity and message integrity.

This function delegates the verification process to the `openssl` command-line utility, a widely trusted tool for cryptographic operations. It runs the `smime -verify` command, which checks the digital signature embedded in the `.p7m` file against a trusted Certificate Authority (CA). The argument `-CAfile` specifies the trusted root certificate used in this verification, in this case the `AgIDCA1_20210921.pem`. This certificate corresponds to the AgID (Agenzia per l'Italia Digitale) root CA, which is the official Italian authority responsible for issuing and managing digital certificates for PEC providers.

So the verification is done in a try/except block. If verification completes without error (i.e. OpenSSL returns 0), the tool returns `True`, indicating the signature is genuine and that the content of the message has not been changed since it has been signed. If OpenSSL threw a `CalledProcessError`, meaning it was not able to verify successfully (e.g. the signature is bad, the certificate is untrusted or corrupted file), the function safely returns `False`. The use of `stdout` and `stderr` redirection ensures that verification logs do not clutter the terminal, keeping the forensic process clean and controlled.

The choice of the `AgIDCA120210921. pem` credential (2021, if relevant) is correct and adequate in this situation. Systems like PEC are based on this architecture with a trust chain at scale: as long as the Root CA is valid and trusted, all certificates of subordinated CAs (and hence also those used to sign messages for the emitters) can be reliably verified. The AgID CA certificate is a long-lived and publicly-trusted trust anchor for Italian PEC services, and it remains valid beyond its issuance date as long as it has not been revoked or superseded. As a consequence, the forensic tool can validate properly all PEC signatures produced by accredited providers under AgID's trust framework released messages in an authentic and legally reliable manner.

The last helper function is the `match_filters` one:

Listing 6: Filters are used on fetched emails

```
    def match_filters(mail):
    if args.subject and args.subject.lower() not in
       (mail.subject or "").lower():
        return False
    if args.from_:
        from_addresses = [addr[1].lower() for addr in
           mail.from_ or []]
```

```
        if not any(args.from_.lower() in addr for addr in
            from_addresses):
            return False
    return True
```

This is a simple yet effective utility that allows filtering of fetched emails based on the command-line arguments provided by the user. Its purpose is to reduce unnecessary processing by limiting the analysis to only those messages that match a specified subject or sender.

The function checks whether a subject filter has been defined and, if so, compares it against the email's subject in a case-insensitive manner. Similarly, if a sender filter is defined, the function extracts all sender addresses from the email's `From` header, converting them to lowercase for consistent comparison. If no match is found, the function returns `False`, effectively skipping that email. Otherwise, it returns `True`, signaling that the email meets the filter criteria and should be processed. This approach improves performance and efficiency, particularly when dealing with large inboxes, while maintaining a clear forensic focus on relevant communications only.

At the heart of the Email Fetcher lies the `parse_message` function, which is responsible for orchestrating the entire forensic extraction workflow:

Listing 7: Core of the tool: it is the main function

```
    def parse_message(raw_message, uid=None, prefix="",
        check_filters=True):
    """Parse email, verify PEC, extract PDFs, store forensic
        JSON."""
    old_stderr = sys.stderr
    sys.stderr = io.StringIO()
    try:
        mail = mailparser.parse_from_bytes(raw_message)
    finally:
        sys.stderr = old_stderr

    if check_filters and not match_filters(mail):
        return False

    pdf_found = False
    for idx, attachment in enumerate(mail.attachments):
        ctype = attachment.get("mail_content_type", "")
        fname = attachment.get("filename") or f"attach_{idx}"

        # ---- Extract PDFs ----
        if ctype == "application/pdf" or
            fname.lower().endswith(".pdf"):
            save_pdf(fname, attachment["payload"],
                mail=mail, uid=uid, prefix=prefix)
            pdf_found = True

        # ---- PEC .p7m ----
        elif fname.lower().endswith(".p7m") or ctype ==
```

```
            "application/pkcs7-mime":
25              temp_file = os.path.join(tempfile.gettempdir(),
                    fname)
26              with open(temp_file, "wb") as f:
27                  f.write(attachment["payload"])
28
29              # ---- Verify signature ----
30              if verify_p7m(temp_file, PEC_CA_CERT):
31                  print(f"[+] PEC signature verified: {fname}")
32                  inner_eml_file =
                        os.path.join(tempfile.gettempdir(),
                        f"{fname}_inner.eml")
33                  subprocess.run([
34                      "openssl", "smime", "-verify",
35                      "-in", temp_file,
36                      "-CAfile", PEC_CA_CERT,
37                      "-out", inner_eml_file
38                  ], check=True)
39                  with open(inner_eml_file, "rb") as f:
40                      inner_content = f.read()
41                  inner_pdf_found =
                        parse_message(inner_content, uid=uid,
                        prefix=f"{fname}_")
42                  pdf_found = pdf_found or inner_pdf_found
43                  os.unlink(inner_eml_file)
44              else:
45                  print(f"[!] Invalid PEC signature, skipping:
                        {fname}")
46              os.unlink(temp_file)
47
48          # ---- Nested .eml ----
49          elif fname.lower().endswith(".eml"):
50              nested_raw = attachment["payload"]
51              if isinstance(nested_raw, str):
52                  nested_raw = nested_raw.encode()
53              nested_pdf_found = parse_message(nested_raw,
                    uid=uid, prefix=f"{fname}_")
54              pdf_found = pdf_found or nested_pdf_found
55
56      if not pdf_found:
57          print("[*] No PDFs found.")
58      return pdf_found
59
60  # -------------------- CLI --------------------
61  parser = argparse.ArgumentParser(
62      description="Forensic PDF extraction from IMAP mailbox
            with PEC verification.",
63      formatter_class=argparse.RawDescriptionHelpFormatter,
64      epilog=textwrap.dedent("""\
```

```
        Examples:
          python script.py --subject "fattura" -n 5
          python script.py --from_ "pec@domain.it"
          python script.py -f "pec@domain.it" -s "fattura"
             -n 3
    """)
)
parser.add_argument("-s", "--subject", help="Filter emails
   by subject (case-insensitive).")
parser.add_argument("-f", "--from", dest="from_",
   help="Filter emails by sender email address.")
parser.add_argument("-n", type=int, default=10, help="Number
   of last emails to process (default 10).")
args = parser.parse_args()

# -------------------- IMAP Processing --------------------
with IMAPClient(HOST, ssl=True) as client:
    client.login(USERNAME, PASSWORD)
    client.select_folder("INBOX")

    if args.subject and args.from_:
        search_criteria = ["FROM", args.from_, "SUBJECT",
           args.subject]
    elif args.subject:
        search_criteria = ["SUBJECT", args.subject]
    elif args.from_:
        search_criteria = ["FROM", args.from_]
    else:
        search_criteria = ["ALL"]

    messages = client.search(search_criteria)
    last_n = messages[-args.n:] if args.n else messages
    print(f"[*] Found {len(last_n)} messages to process")

    for uid in last_n:
        raw_message = client.fetch(uid,
           ["RFC822"])[uid][b"RFC822"]
        parse_message(raw_message, uid=uid)
```

This function parses an email message, verifies any digital signatures (PEC), extracts all relevant attachments such as PDFs and stores the corresponding forensic metadata.

The function begins by temporarily redirecting the standard error stream to suppress unwanted warnings from the `mailparser` library, ensuring clean console output during forensic operations. The raw email content is then parsed into a structured `mail` object using the `mailparser.parse_from_bytes` method, which facilitates easy access to headers, subjects, attachments and sender information.

If email filters are enabled through command-line arguments, `parse_message` invokes the `match_filters` function to determine whether the email should be pro-

cessed. Emails that do not meet the criteria are skipped, ensuring that only relevant evidence is collected.

The core logic loops through all attachments in the mail. For each attachment gets its content type, filename and does one of the following action:

- **PDF extraction**: if the attachment is recognized as a PDF (either by way of MIME type or file extension), the method calls `save_pdf`, which saves it into the appropriate folder and creates a JSON metadata record. This process maintains the integrity and provenance of every document.
- **PEC signature verification**: if the attached file is a . p7m file (for example, a PEC-signed email or document), it is initially written to a temporary directory. If the verification passes, the signed content is then extracted using Openssl `smime -verify` and temporarily written out to a temporary file as an inner .eml file. The function then calls itself recursively on this inner email, so that even messages (and sub-attachments) within attachments get processed and validated. In case of an invalid PEC signature, the attachment is ignored, but a helpful message written to console. In every case, temporary files are deleted once the operation is completed to prevent from leaving any leftover data.
- **Nested emails**: if an attachment is an `.eml` file, indicating a nested email, the function encodes it if necessary and again recursively calls itself. This recursive implementation makes it possible to process emails inside other emails (forwarded PEC message for example) and to extract embedded PDFs without the user interference.

As the final step, if no PDFs are found in a message or its document attachments, it prints a message to alert us that no useful content was discovered there. The function returns a boolean value to represent the fact of an extraction having succeeded keeping code flow well-organized and traceable.

The last section of the Email Fetcher script takes care of the command-line interface (CLI), and connecting with the IMAP mailbox. It is defined by using the `argparse` library and it allows for filtering emails by subject (`--subject`) or by sender (`--from`), as well as analyzing only a predefined number of received messages (`-n`). To facilitate use by the user, default values and example usages are given in help messages.

After the arguments are processed, the script uses the IMAPClient library to make a secure connection to the mail server using IMAP and SSL. It logs in with the credentials found from the environment variables then switches to the INBOX folder. It builds the search query that is according to user-defined filters and retrieves the most recent n messages.

Once arguments are parsed, the script establishes a secure IMAP connection to the mail server using the `IMAPClient` library with SSL enabled. It authenticates using credentials loaded from the environment variables and selects the `INBOX` folder for processing. Based on the user-defined filters, it constructs the appropriate search criteria and retrieves the most recent `n` messages.

For each selected message, the script fetches the raw content in RFC822 [14] format and passes it to the `parse_message` function for forensic analysis. This design ensures modularity and scalability: the fetching logic remains separate from the forensic extraction process, while the recursive structure of `parse_message` guarantees that every level of email nesting and signature verification is handled automatically and securely.

Overall, the `parse_message` function and its integration with the CLI and IMAP components form the operational backbone of the Email Fetcher. Together, they enable automated, secure and reproducible forensic extraction of PDFs from certified email systems, maintaining full compliance with the principles of digital evidence preservation and traceability.

Finally, the following diagram illustrates the complete workflow: connecting to the IMAP mailbox, fetching emails, applying filters, parsing messages, verifying PEC signatures if any, extracting PDFs, generating metadata and ensuring forensic integrity.

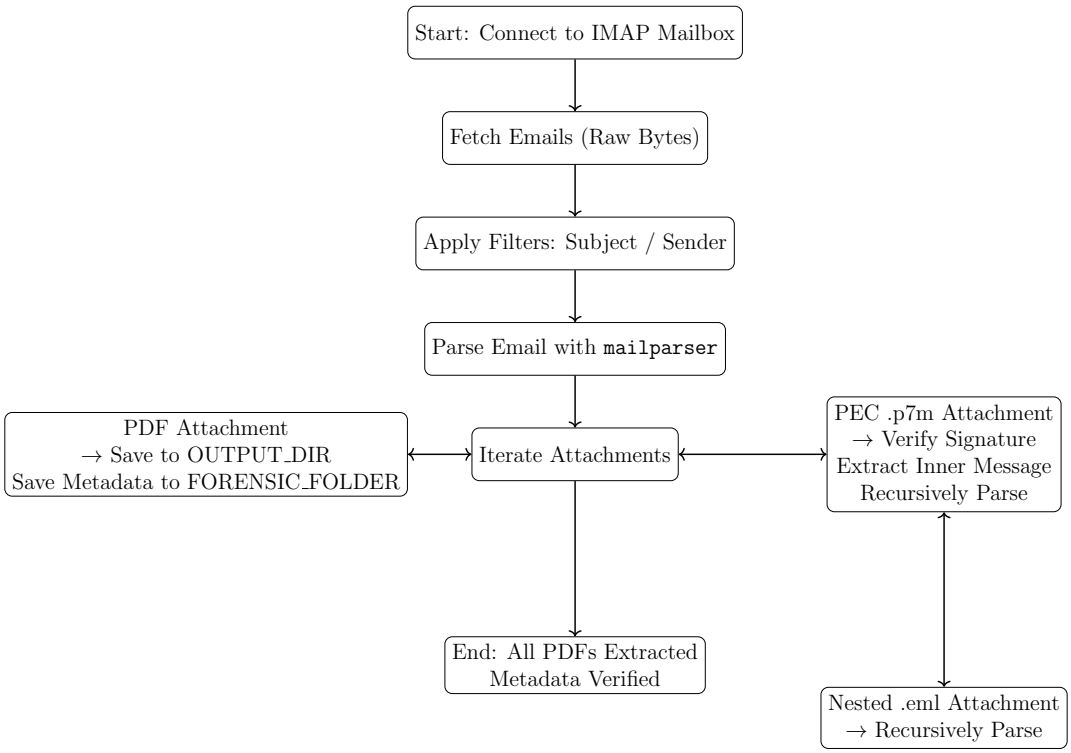

Figure 8: Email Fetcher Workflow

9.4.3 Enhanced Peepdf-3

The second vital component is the enhanced version of `peepdf` adapted for Python 3. During the analysis of the original tool, several limitations were identified:

- Lack of any analysis for embedded files.
- Lack of verification of digital signatures, especially concerning signature coverage with reference to the signed object. This could be used as an attack

vector.

- Lack of support for automatic report generation after the interaction with `peepdf`.

To overcome these issues, we developed additional functionalities to extend the tool and to enable automatic report generation after the analysis of a PDF document.

To integrate new features into the original `peepdf`, it was necessary to modify the `PDFConsole.py` file, where all the core functionalities are defined and implemented.

Embedded Analysis Functionality

The first new feature introduced is the `embedded_analysis` command. This functionality required the creation of two main components: the core function, following the naming convention `do_embedded_analysis`, and the supporting routines responsible for entropy computation, hashing, and signature checks.

The implementation of the core function is reported below:

Listing 8: Core implementation of the embedded_analysis functionality

```
def do_embedded_analysis(self, arg):
    def shannon_entropy(data):
        """Calculate Shannon entropy of a byte sequence."""
        if not data:
            return 0
        freq = [data.count(bytes([i])) / len(data) for i in
           range(256)]
        return -sum(f * math.log2(f) for f in freq if f > 0)

    def file_hashes(data):
        """Return MD5, SHA1, and SHA256 hashes of the given
           data."""
        return {
            "md5": hashlib.md5(data).hexdigest(),
            "sha1": hashlib.sha1(data).hexdigest(),
            "sha256": hashlib.sha256(data).hexdigest()
        }

    try:
        # Check if a PDF file has been loaded in the current
           session
        if self.pdfFile is None:
            self.log_output("embedded_analysis", "[!] Error:
               You must open a file!")
            return False

        # Load and compile YARA rules to detect suspicious
           content
        rules = yara.compile(filepath="../rules.yar")

```

```
        # Extract embedded files from the currently opened PDF
        embedded = self.getEmbeddedFiles()
        if not embedded:
            self.log_output("embedded_analysis", "[!] No embedded files found")
            return False

        # Process each embedded file individually
        for f in embedded:
            filename = f["filename"]
            data = f["data"]

            # Ensure data is in bytes format
            data_bytes = data.encode(errors='ignore') if isinstance(data, str) else data

            # --- Header section ---
            print("=" * 60)
            print(f"Analyzing embedded file: {filename}")
            print("=" * 60)

            # --- YARA Scan ---
            matches = rules.match(data=data_bytes)
            if matches:
                print(f"[!] Suspicious: YARA match - {matches}")
            else:
                print("[-] No YARA matches")

            # --- MIME Type Verification ---
            try:
                mime_type = magic.from_buffer(data_bytes, mime=True)
                print(f"[i] MIME type: {mime_type}")

                expected_mime, _ = mimetypes.guess_type(filename)
                if expected_mime and expected_mime != mime_type:
                    print(f"[!] Suspicious: Extension does not match MIME ({expected_mime} vs {mime_type})")
                elif expected_mime:
                    print(f"[+] Extension matches MIME type")
            except Exception as e:
                print(f"[!] Failed to detect MIME type: {e}")

            # --- Hash Computation ---
```

```
66              hashes = file_hashes(data_bytes)
67              print("[i] Hashes:")
68              for hname, hval in hashes.items():
69                  print(f"    {hname.upper():7}: {hval}")
70
71              # --- VirusTotal Query (if API key available) ---
72              if "yourAPIkey" in self.variables["vt_key"][0]:
73                  print("[i] No VirusTotal API key found
                       \rightarrow skipping check on md5")
74              else:
75                  ret = vtcheck(hashes["md5"],
                       self.variables["vt_key"][0])
76                  if ret[0] == -1:
77                      print(f"[!] Error querying VirusTotal:
                           {ret[1]}")
78                  else:
79                      jsonDict = ret[1]
80                      maliciousCount =
                           jsonDict["data"]["attributes"]
81                      ["last_analysis_stats"]["malicious"]
82                      totalCount =
                           sum(jsonDict["data"]["attributes"]
83                      ["last_analysis_stats"].values())
84                      selfLink =
                           f'https://www.virustotal.com/gui
85                      /file/{jsonDict["data"]
86                      ["attributes"]["sha256"]}'
87
88                      print(f"[i] VirusTotal report:")
89                      print(f"    Detection rate:
                           {maliciousCount}/{totalCount}")
90                      print(f"    Report link: {selfLink}")
91
92                      if maliciousCount > 0:
93                          print(f"    [!] Marked as MALICIOUS
                               by {maliciousCount} engines")
94                      else:
95                          print("    [+] No malicious
                               detections")
96
97              # --- Entropy Calculation ---
98              entropy = shannon_entropy(data_bytes)
99              if entropy > 7.5:
100                 print(f"[!] Suspicious: High entropy
                       ({entropy:.2f})")
101             else:
102                 print(f"[+] Entropy: {entropy:.2f} (normal
                       if < 7.5)")
103
```

```
104             print()  # Blank line between files
105
106     except Exception as e:
107         print(f"[!] Exception during embedded analysis: {e}")
108         return False
```

The function begins by verifying that a PDF file is currently loaded and accessible. Then extract all the possible embedded files and each of them is analyzed individually through four verification steps:

- **Yara Check**: the `embedded_analysis` functionality relies on a set of YARA rules designed to identify suspicious or potentially malicious embedded files within a PDF document. They can be modified directly by users to match with specific requirements. YARA is a static analysis engine widely used in malware research to detect patterns in binary or textual data based on user-defined rules. Each rule below defines a specific signature associated with known malicious file types or suspicious embedded structures.

Listing 9: YARA rules used for embedded file detection within enhanced Peepdf-3

```
1  rule contains_pe_file_attachment {
2      meta:
3          description = "Detects MZ header inside an
               extracted attachment"
4      strings:
5          $pe_header = { 4D 5A }   // 'MZ'
6      condition:
7          $pe_header
8  }
9
10 rule detect_activemime_attachment {
11     meta:
12         description = "Detects base64('ActiveMime') in
               an extracted attachment"
13     strings:
14         $activemime_b64 = "QWN0aXZlTWltZQ==" ascii
15     condition:
16         $activemime_b64
17 }
18
19 rule detect_embedded_excel_attachment {
20     meta:
21         description = "Detects Excel XML tag inside
               extracted attachment"
22     strings:
23         $xls_header = "<x:ExcelWorkbook>" ascii
24     condition:
25         $xls_header
26 }
```

```
27
28 rule detect_embedded_word_attachment {
29     meta:
30         description = "Detects Word XML tag inside
               extracted attachment"
31     strings:
32         $word_header = "<w:WordDocument>" ascii
33     condition:
34         $word_header
35 }
36
37 rule detect_embedded_mht_attachment {
38     meta:
39         description = "Detects 'mime' token inside
               extracted attachment"
40     strings:
41         $mht_header = "mime" ascii
42     condition:
43         $mht_header
44 }
45
46 rule detect_embedded_pdf_attachment {
47     meta:
48         description = "Detects embedded PDF header
               inside an extracted attachment"
49     strings:
50         $pdf_header = "%PDF-" ascii
51     condition:
52         $pdf_header
53 }
```

Each rule performs pattern-based detection on the content of the extracted embedded files:

- **contains_pe_file_attachment**: detects the `MZ` header (`0x4D5A`), which indicates the presence of a Windows Portable Executable (PE) file, a common container for malicious payloads.
- **detect_activemime_attachment**: identifies base64-encoded strings that include the `ActiveMime` header, which is a frequent feature of Microsoft Office documents and commonly used to conceal embedded macros
- **detect_embedded_excel_attachment**: detects the XML tag `<x:ExcelWorkbook>`, revealing the presence of an embedded Excel document.
- **detect_embedded_word_attachment**: searches for tag `<w:WordDocument>` indicating embedded Microsoft Word document.
- **detect_embedded_mht_attachment**: finds the "mime" keyword that is commonly present in MIME HTML content.

– **detect_embedded_pdf_attachment**: identify the `%PDF-header`, detecting embedded or nested PDF files.

These rules are loaded at runtime by the `embedded_analysis` function and applied to each extracted attachment. If any of these signatures are matched, the tool flags the file as suspicious and reports the corresponding rule name, providing analysts with immediate insight into the nature of the embedded content.

- **MIME Type Verification**: compares the discovered MIME type of the embedded file (as discovered by libmagic) with its computed expected one given the file extension. A discrepancy between extension announced (e.g. `.jpg`) and the correct MIME type (e.g. `application/x-executable`), hinting that this file could be a deliberately disguised attempt to mask the actual file type, which is a common obfuscation technique used in malicious PDFs.

- **File hashes & VirusTotal lookup**: calculates the cryptographic hash functions of the embedded file (MD5, SHA1, SHA256) and outputs them for comparison against analysts results. If you have configured your own VirusTotal API key (`vt_key`), the tool sends only its calculated MD5 hash across the network to query the VirusTotal API and obtain a report on that file. The lookup provides the aggregated detection statistics (e.g. how many engines have labeled the file as malicious) and a URL for a direct report. As an aside, notice that we deliberately send the hash instead of the file to reduce data exposure.

- **Entropy analysis**: calculates the Shannon entropy of embedded file's byte stream, to check randomness level in its content. The entropy $H(X)$ of a discrete random variable X representing the byte values is computed using the classic Shannon formula:

$$H(X) = -\sum_{i=0}^{255} p(i) \log_2 p(i)$$

where $p(i)$ is the probability (frequency) of each possible byte value within the file. In practice, the function counts the occurrences of all 256 possible byte values, normalizes them by the file length to estimate $p(i)$, and then sums the resulting values weighted by their logarithms. The resulting entropy score ranges from 0 (no randomness, e.g. a file filled with a single repeated byte) to 8 (maximum randomness for uniformly distributed byte values). High entropy (typically above 7.5) is indicative that the content is compressed or encrypted, and can be a characteristic of obfuscated or malicious payloads embedded within PDF files.

Two auxiliary functions are introduced to implement this feature.

The first helper functions really should help end users have an understanding of how the new function can be used and what checks it does. Its implementation is provided in the listing above.

```
    def help_embedded_analysis(self):
        newLine = "\n"
        print(f"{newLine}Usage:")
        print("  embedded_analysis")

        print(f"{newLine}Description:")
        print("  Analyze all embedded files in the currently loaded PDF, including:")
        print("    • /EmbeddedFiles (standard PDF attachments)")
        print("    • Annotation-based file attachments")

        print(f"{newLine}Checks performed for each embedded file:")
        print("  1. YARA scan")
        print("     → Matches against predefined YARA rules to detect known malicious patterns.")
        print("  2. MIME type detection")
        print("     → Identifies the file type from its magic number and compares it with the extension.")
        print("  3. Hash computation")
        print("     → Calculates MD5, SHA1, and SHA256 for reference or external checks (e.g., VirusTotal).")
        print("  4. Entropy analysis")
        print("     → High entropy (> 7.5) may indicate encryption, compression, or obfuscation.")

        print(f"{newLine}Malicious indicators:")
        print("  [!] YARA match")
        print("      → The file matches a rule and is flagged as potentially malicious.")
        print("  [!] MIME type mismatch")
        print("      → Example: a file named '.jpg' but detected as 'application/x-dosexec'.")
        print("  [!] High entropy")
        print("      → Could indicate packed or encrypted content.")

        print(f"{newLine}Output format:")
        print("  For each embedded file, the analysis will display:")
        print("    • Filename")
        print("    • YARA results")
        print("    • MIME type and extension check")
        print("    • File hashes (MD5, SHA1, SHA256)")
        print("    • VirusTotal detection rate (if API key is configured)")
        print("    • Entropy value and suspiciousness indicator")
```

```
37          print(newLine)
```

The second helper function is responsible for extracting all embedded files from the currently parsed PDF object. The implementation of this extraction routine is provided in the following listing:

```
1  def getEmbeddedFiles(self):
2      from PDFCore import PDFTrailer
3
4      files = []
5
6      if not self.pdfFile:
7          return files
8
9      # Flatten trailer list
10     def flatten_trailers(trailers):
11         flat = []
12         for t in trailers:
13             if isinstance(t, list):
14                 flat.extend(flatten_trailers(t))
15             else:
16                 flat.append(t)
17         return flat
18
19     trailers = flatten_trailers(self.pdfFile.trailer)
20
21     # Helper: resolve a reference if needed
22     def resolve(obj):
23         if hasattr(obj, "getType") and obj.getType() ==
               "reference":
24             return self.pdfFile.getObject(obj.getId())
25         return obj
26
27     # Helper: extract a file spec into bytes
28     def extract_file(file_spec, filename_hint=None):
29         file_spec = resolve(file_spec)
30         if hasattr(file_spec, "hasElement") and
               file_spec.hasElement("/EF"):
31             ef_dict =
                   resolve(file_spec.getElementByName("/EF"))
32             file_stream_ref =
                   resolve(ef_dict.getElementByName("/F"))
33
34             file_stream = resolve(file_stream_ref)
35
36             # Get raw bytes
37             if hasattr(file_stream, "stream"):
38                 raw_bytes = file_stream.stream
39             elif hasattr(file_stream, "getRawStream"):
40                 raw_bytes = file_stream.getRawStream()
```

```
            else:
                return None

            filters = file_stream.getElementByName("/Filter")
            decode_params =
               file_stream.getElementByName("/DecodeParms")

            if filters:
                status, data_bytes = decodeStream(raw_bytes,
                   filters, decode_params)
                if status != 0:
                    return None
            else:
                data_bytes = raw_bytes
            return data_bytes
        else:
            return None

    for idx, tr in enumerate(trailers):
        if not isinstance(tr, PDFTrailer):
            continue

        tr_dict = tr.trailerDict
        root_ref = tr_dict.getElementByName("/Root")
        if not root_ref:
            continue
        root = resolve(root_ref)

        # --- Extract /EmbeddedFiles ---
        if root.hasElement("/Names"):
            names_dict =
               resolve(root.getElementByName("/Names"))
            if names_dict.hasElement("/EmbeddedFiles"):
                embedded_dict = resolve(names_dict
                .getElementByName("/EmbeddedFiles"))
                if embedded_dict.hasElement("/Names"):
                    names_array_obj = resolve(embedded_dict
                    .getElementByName("/Names"))
                    if hasattr(names_array_obj,
                       "getElements"):
                        names_array =
                           names_array_obj.getElements()
                    else:
                        names_array = list(names_array_obj)

                    for i in range(0, len(names_array), 2):
                        name_obj = names_array[i]
                        file_spec_ref = names_array[i + 1]

```

```
                        filename = name_obj.getValue() if
                            hasattr(name_obj, "getValue")
                            else str(name_obj)
                        file_spec = resolve(file_spec_ref)

                        data_bytes = extract_file(file_spec,
                            filename)
                        if data_bytes:
                            files.append({"filename":
                                filename, "data": data_bytes})

    # --- Extract /FileAttachment annotations ---
    if root.hasElement("/Pages"):
        def process_pages(pages_dict):
            pages_dict = resolve(pages_dict)
            kids = pages_dict.getElementByName("/Kids")
                if pages_dict.hasElement("/Kids") else []
            if hasattr(kids, "getElements"):
                kids_list = kids.getElements()
            else:
                kids_list = list(kids)

            for kid_ref in kids_list:
                page = resolve(kid_ref)
                if page.hasElement("/Annots"):
                    annots_array = resolve(page
                    .getElementByName("/Annots"))
                    if hasattr(annots_array,
                        "getElements"):
                        annots =
                            annots_array.getElements()
                    else:
                        annots = list(annots_array)

                    for annot_ref in annots:
                        annot = resolve(annot_ref)
                        subtype =
                        annot.getElementByName("/Subtype")
                        subtype_val = subtype.getValue()
                            if subtype and
                            hasattr(subtype, "getValue")
                            else None
                        if subtype_val ==
                            "/FileAttachment":
                            file_spec =
                            resolve(annot
                            .getElementByName("/FS"))
                            filename_obj =
                                resolve(file_spec
```

```
122                                     .getElementByName("/F")) if
                                           file_spec.hasElement("/F")
                                           else None
123                                     filename =
                                           filename_obj.getValue()
                                           if filename_obj and
                                           hasattr(filename_obj,
                                           "getValue") else "unknown"
124                                     data_bytes =
                                           extract_file(file_spec,
                                           filename)
125                                     if data_bytes:
126                                         files.append({"filename":
                                               filename, "data":
                                               data_bytes})
127                         # Recurse into nested /Pages
128                         if page.hasElement("/Kids"):
129                             process_pages(page)
130
131                 process_pages(root.getElementByName("/Pages"))
132
133     return files
```

This function is responsible for extracting all embedded files from the currently loaded PDF. This includes both standard PDF attachments stored under the `/EmbeddedFiles` name tree and file attachments associated with annotations (`/FileAttachment`) on PDF pages.

The function does the following:

1. **Initialize storage**: creates an empty list which is used to store all extracted files.
2. **Flatten trailer structure**: PDF document may contain several trailers, and these can be nested in lists. The function recursively flattens these into a single list to be processed.
3. **Reference resolution**: introduces a helper function which resolves object references to their real content from within the PDF. This makes sure that any later accesses use the right objects.
4. **File extraction helper**: defines a function extracting raw bytes from file spec object. This should also manage whatever filters (`/Filter`) and decode parameters (`/DecodeParms`) might be in effect to get the actual file data.
5. **Iterate through trailers**: for each PDF trailer, the function resolves the `/Root` object and looks for embedded content.
6. **Extract /EmbeddedFiles**: if the PDF has an `/EmbeddedFiles` name tree, the method loops over name-value pairs and dereferences each file specification, returning its bytes.

7. **Extract /FileAttachment annotations**: for each page in the PDF, it searches for annotation objects of subtype `/FileAttachment` and extracts the associated files. The function recursively processes nested pages to ensure all attachments are captured.

8. **Return collected files**: the function returns a list of dictionaries, each containing the filename and the corresponding byte stream of the embedded file.

To conclude, this functionality allows a thorough and multi-layered analysis of embedded files within PDF documents. By combining signature-based detection (YARA rules), file type verification (MIME type), cryptographic hashing and entropy analysis, it provides several levels of inspection to identify potentially malicious or suspicious content.

This is particularly useful in forensic investigations, as PDFs' embedded files are frequently leveraged by attackers to propagate malware, obfuscate scripts or exfiltrate data. The function enables analysts to:

- Detect known malicious patterns in attachments via YARA rules.
- Identify inconsistencies between file extensions and actual content, which could suggest intentional obfuscation.
- Calculate file hashes to check them against threat intelligence sources like VirusTotal.
- Detect encrypted or compressed payloads via entropy analysis, which often signify attempts to evade detection.

Overall, by automatically extracting and investigating each embedded objects, this tool offers a consistent and repeatable approach for forensic analysis, assisting investigators in identifying concealed threats, and accurately evaluating the security risks of PDF.

Check Signatures Functionality

The second functionality implemented in the tool is `do_check_signatures`, which allows the extraction and verification of PDF signatures. Its primary purpose is to determine which objects in a PDF are covered by digital signatures and which are not, providing a detailed and structured overview of signed and unsigned objects. The core function performs several steps, combining reference resolution, byte-range analysis and object mapping to give forensic insight into the PDF's signature integrity.

Listing 10: Core function of do_check_signatures

```
1 def do_check_signatures(self, arg):
2         """
3         Checks PDF objects against signature ByteRange(s)
             and prints which objects are signed.
4         Marks the actual signature object(s) and includes
             full debug output.
```

```
        """

        def resolve(obj):
            """Recursively resolve references to get the
               actual object."""
            try:
                while hasattr(obj, "getType") and
                   obj.getType() == "reference":
                    obj = self.pdfFile.getObject(obj.getId())
            except Exception as e:
                return
            return obj

        def is_object_signed(obj_start, obj_end,
           byte_ranges):
            """Returns True if the object falls entirely
               within any ByteRange segment."""
            for i in range(0, len(byte_ranges), 2):
                start = byte_ranges[i]
                length = byte_ranges[i + 1]
                if obj_start >= start and obj_end <= start +
                   length:
                    return True
            return False

        if not self.pdfFile:
            print("[!] No PDF loaded")
            return

        # --- Flatten objects from offsets ---
        try:
            offsets_array = self.pdfFile.getOffsets()
        except Exception as e:
            return

        objects = []
        for offset_block in offsets_array:
            if "objects" in offset_block:
                for obj_id, start, size in
                   offset_block["objects"]:
                    objects.append({
                        "Id": obj_id,
                        "Start": start,
                        "End": start + size - 1
                    })

        # --- Build resolved object map for marking
           signature objects ---
        resolved_obj_map = {}
```

```
47          for o in objects:
48              try:
49                  obj =
                       resolve(self.pdfFile.getObject(o["Id"]))
50                  resolved_obj_map[id(obj)] = o["Id"]
51              except Exception as e:
52                  return
53
54          # --- Identify signature fields ---
55          sig_objs = []
56          for o in objects:
57              try:
58                  obj =
                       resolve(self.pdfFile.getObject(o["Id"]))
59                  ft_val = None
60                  v_val = None
61
62                  try:
63                      ft = obj.getElementByName("/FT") if
                           hasattr(obj, "getElementByName") else
                           None
64                      ft_val = ft.getValue() if ft and
                           hasattr(ft, "getValue") else ft
65                  except:
66                      return
67
68                  try:
69                      v_val = obj.getElementByName("/V") if
                           hasattr(obj, "getElementByName") else
                           None
70                  except:
71                      return
72
73                  if ft_val == "/Sig" and v_val is not None:
74                      sig_objs.append(resolve(v_val))
75
76              except:
77                  return
78
79          if not sig_objs:
80              print("\n[!] No signatures found\n")
81              return
82
83          # --- Check each object against each signature ---
84          for idx, sig in enumerate(sig_objs, start=1):
85              try:
86                  br = sig.getElementByName("/ByteRange") if
                       hasattr(sig, "getElementByName") else None
87                  if not br:
```

```
                    continue

                if hasattr(br, "getElements"):
                    byte_ranges = [int(x.getValue()) if
                        hasattr(x, "getValue") else int(x)
                        for x in br.getElements()]
                else:
                    byte_ranges = [int(x.getValue()) if
                        hasattr(x, "getValue") else int(x)
                        for x in br]

                valid, msg = self.verify_signature(idx)

                print("{:<8} {:<10} {:<10}
                    {:<30}".format("Object", "Start", "End",
                    "Signed?"))

                RED = "\033[91m"
                RESET = "\033[0m"

                for o in objects:
                    start = o["Start"]
                    end = o["End"]
                    obj_id = o["Id"]
                    signed = is_object_signed(start, end,
                        byte_ranges)
                    is_sig_obj = id(sig) in resolved_obj_map
                        and resolved_obj_map[id(sig)] ==
                        obj_id

                    marker = "YES" if signed else "NO"
                    if is_sig_obj:
                        marker += " (Signature Object)"

                    if not signed and not is_sig_obj:
                        print(f"{RED}{obj_id:<8} {start:<10}
                            {end:<10} {marker:<30}{RESET}")
                    else:
                        print(f"{obj_id:<8} {start:<10}
                            {end:<10} {marker:<30}")

            except:
                return

        # --- Recap for multiple signatures ---
        if len(sig_objs) > 1:
            recap = {}
            for o in objects:
                recap[o["Id"]] = {
```

```
                    "Start": o["Start"],
                    "End": o["End"],
                    "covered_by": []
                }

            for sig_idx, sig in enumerate(sig_objs, start=1):
                br = sig.getElementByName("/ByteRange") if
                   hasattr(sig, "getElementByName") else []
                if hasattr(br, "getElements"):
                    byte_ranges = [int(x.getValue()) if
                       hasattr(x, "getValue") else int(x)
                       for x in br.getElements()]
                else:
                    byte_ranges = [int(x.getValue()) if
                       hasattr(x, "getValue") else int(x)
                       for x in br]

                for o in objects:
                    if is_object_signed(o["Start"],
                       o["End"], byte_ranges):
                        recap[o["Id"]]
                        ["covered_by"].append(sig_idx)

            last_sig_obj_id =
               resolved_obj_map.get(id(sig_objs[-1]), None)

            print("\nFINAL RECAP OF ALL SIGNATURES:")
            print("{:<8} {:<10} {:<10} {:<20} {:<25}".format(
                "Object", "Start", "End", "Signed by",
                   "Status"
            ))

            RED = "\033[91m"
            RESET = "\033[0m"

            for obj_id, info in recap.items():
                signed_by = ",".join(str(s) for s in
                   info["covered_by"])
                if obj_id == last_sig_obj_id:
                    status = "Last Signature Object"
                elif not signed_by:
                    status = "Unsigned / Suspicious"
                else:
                    status = "Covered"

                line = "{:<8} {:<10} {:<10} {:<20}
                   {:<25}".format(
                    obj_id, info["Start"], info["End"],
                       signed_by or "-", status
```

```
164                     )
165                     if not signed_by and obj_id !=
                            last_sig_obj_id:
166                         line = RED + line + RESET
167                     print(line)
168
169                 print("\n")
```

The function performs the following tasks in detail:

- **Reference resolution**: the function defines a helper `resolve(obj)` that recursively dereferences PDF objects. Since PDF objects can be indirect references, resolving them ensures the function operates on the actual content of the document.
- **Object offset mapping**: the function reads the `/ByteRange` for each signature object indicating which byte ranges of the PDF is covered by that signature. Then, it inspects each object to see whether its range of bytes is completely within any one of the signed segment.
- **Signature object identification**: the function iterates through all PDF objects and identifies those that are signature fields. This is done by checking for an `/FT` (field type) of `/Sig` and the presence of a `/V` (value) element. These objects are stored for further analysis.
- **ByteRange verification**: for each signature object, the function retrieves the `/ByteRange`, which specifies the portions of the PDF covered by the signature. It then checks each object to determine whether its byte range falls entirely within any of the signed segments.
- **Marking and output**: it prints in a table format all objects showing their identifier, starting and ending offset, as well as the information about if it is signed. If the object is a signature object, it is marked in a special way. Unsigned objects are highlighted in red to draw attention to potential inconsistencies or tampering.
- **Multiple signature handling**: if multiple signatures are present, the function constructs a summary that reports which objects are covered by each signature. It also identifies the last signature object, the signing time, and marks unsigned elements as "Unsigned/Suspicious", thus giving a comprehensive forensics brief of all signatures in a document.
- **Debug and validation**: the function optionally invokes `verify_signature` to validate each signature. Detailed debug logging can be examined by analysts to determine signature coverage and validate that the PDF has not been tampered with.

The `check_signatures` functionality relies, similarly to `embedded_analysis`, on two helper functions. The first helper function guides users in understanding the purpose of the function and interpreting its output:

Listing 11: Helper function describing the signature check workflow

```
    def help_check_signatures(self):
        newLine = "\n"
        print(f"{newLine}Usage: check_signatures")
        print(f"{newLine}Checks the digital signatures in a
            PDF and validates them.")
        print(f"For each signature, the tool verifies:")
        print(f"  1. Cryptographic validity of the signature
            (hash and signed content).")
        print(f"  2. Certificate validity (trusted, expired,
            or self-signed).")

        print(f"{newLine}For each signature found, it prints
            a block with:")
        print(f"  SUMMARY       : Signer, status (VALID,
            INVALID, or UNTRUSTED), and reason if applicable")
        print(f"  PDF METADATA  : SubFilter, Signing Date,
            Location, and ByteRange")
        print(f"  CERTIFICATE INFO : Certificate details
            such as CommonName, Organization, SerialNumber,
            Issuer, and validity dates")

        print(f"{newLine}After that, it prints a table for
            all PDF objects:")
        print(f"  Object   Start      End        Signed?")

        print(f"{newLine}Legend:")
        print(f"  YES                      : Object fully
            covered by the signature")
        print(f"  NO                       : Object not
            covered (may require further inspection)")
        print(f"  NO (Signature Object)    : Object contains
            the signature itself (normal, not suspicious)")

        print(f"{newLine}Color codes (if supported by
            terminal):")
        print(f"  RED     : Object outside signature
            coverage (possible tampering)")
        print(f"  Default : Object properly covered or
            signature object itself")

        print(f"{newLine}Notes:")
        print(f"  - Multiple signatures are supported; each
            ByteRange is checked independently.")
        print(f"  - Unsigned objects may indicate tampering
            or unsigned content.")
        print(f"  - Signature objects are marked separately
            because they naturally fall outside their own
            ByteRange.")
```

The second helper function, `verify_signature`, is responsible for verifying the validity of digital signatures embedded within a PDF document. This function is used by `do_check_signatures` to provide detailed information about each signature and assess its trustworthiness.

Listing 12: Function to verify digital signature

```
    def verify_signature(self, sig_id=None):
        import logging, traceback
        from pyhanko.pdf_utils.reader import PdfFileReader
        from pyhanko.sign.validation import
           validate_pdf_signature, ValidationContext
        from pyhanko.sign.validation.errors import
           DisallowedAlgorithmError
        from asn1crypto import x509
        from dateutil import parser

        logging.getLogger("pyhanko.sign.validation")
        .setLevel(logging.CRITICAL)
        logging.getLogger("pyhanko_certvalidator")
        .setLevel(logging.CRITICAL)

        pdf_path = self.pdfFile.getPath()
        results = []

        def extract_asn1_cert_info(cert: x509.Certificate):
            def get_name_value(name, attr):
                for rdn in name.chosen:
                    for ava in rdn:
                        if ava['type'].native == attr:
                            return ava['value'].native
                return None

            subject = cert.subject
            issuer = cert.issuer

            return {
                "CommonName": get_name_value(subject,
                   'common_name'),
                "Organization": get_name_value(subject,
                   'organization_name'),
                "OrganizationalUnit":
                   get_name_value(subject,
                   'organizational_unit_name'),
                "SerialNumber": hex(cert.serial_number),
                "IssuerCN": get_name_value(issuer,
                   'common_name'),
                "ValidFrom":
                   cert['tbs_certificate']['validity']
                ['not_before'].native,
```

```
                "ValidTo":
                    cert['tbs_certificate']['validity']
                ['not_after'].native,
            }

        def normalize_pdf_value(v):
            """If v is a PDF string-like object with
               getValue(), return its value, else str or
               None."""
            try:
                if v is None:
                    return None
                if hasattr(v, "getValue"):
                    return v.getValue()
                return str(v)
            except Exception:
                return str(v)

        try:
            with open(pdf_path, "rb") as f:
                reader = PdfFileReader(f)
                sig_fields = reader.embedded_signatures

                if not sig_fields:
                    return False, "No signatures found in
                       the PDF"

                for idx, sig in enumerate(sig_fields,
                   start=1):
                    if sig_id is not None and idx != sig_id:
                        continue

                    # --- PDF metadata ---
                    md = {}
                    try:
                        sig_dict = sig.sig_object
                        if sig_dict:
                            md["Signer"] =
                               normalize_pdf_value(sig_dict
                            .get("/Name", "Unknown"))
                            md["SubFilter"] =
                               normalize_pdf_value(sig_dict
                            .get("/SubFilter"))
                            md["Reason"] =
                               normalize_pdf_value(sig_dict
                            .get("/Reason"))
                            md["ContactInfo"] =
                               normalize_pdf_value(sig_dict
                            .get("/ContactInfo"))
```

```
                md["Location"] =
                    normalize_pdf_value(sig_dict
                .get("/Location"))
                md["SigningDate"] =
                    normalize_pdf_value(sig_dict
                .get("/M"))
                md["ByteRange"] = sig.byte_range
            else:
                md = {"Signer": "Unknown"}
        except Exception:
            md = {"Signer": "Unknown"}

        # --- Validation & certificate ---
        cert_info = {}
        summary = "VALIDATION SKIPPED"
        trusted = False
        reason_invalid = []

        try:
            status = validate_pdf_signature(sig)
            valid = getattr(status, "valid",
                False)
            trusted = getattr(status, "trusted",
                False)

            if valid and trusted:
                summary = "VALID & TRUSTED"
                reason_invalid.append("Signature
                    cryptographically valid;
                    certificate trusted and
                    verified")
            elif valid and not trusted:
                summary = "VALID but UNTRUSTED"
                reason_invalid.append("Certificate
                    is valid but not trusted
                    (self-signed or unrecognized
                    CA)")
            elif getattr(status,
                "weak_algorithms", None):
                summary = "VALID but WEAK
                    ALGORITHM"
                reason_invalid.append("weak
                    algorithm")

            cert = getattr(status,
                "signer_cert", None)
            if cert:
                cert_info =
                    extract_asn1_cert_info(cert)
```

```
110
111                     except Exception as e:
112                         summary = f"VALIDATION ERROR ({e})"
113                         reason_invalid.append(str(e))
114                         try:
115                             cert = getattr(sig,
                                    "signer_cert", None)
116                             if not cert and hasattr(sig,
                                    "embedded_signature"):
117                                 cert =
118                                 getattr(sig.embedded_signature,
119                                 "signer_cert", None)
120                             if cert:
121                                 cert_info =
122                                 extract_asn1_cert_info(cert)
123                         except:
124                             pass
125
126                     result = {
127                         "Signature": idx,
128                         "Signer": md.get("Signer",
                                "Unknown"),
129                         "Trusted": trusted,
130                         "Summary": summary,
131                         "Metadata": md,
132                         "Certificate": cert_info,
133                         "ReasonInvalid": ";
                                ".join(reason_invalid) if
                                reason_invalid else ""
134                     }
135
136                     results.append(result)
137
138                     # --- Print everything divided by
                            section ---
139                     print(f"\nSignature {idx}:")
140
141                     # SUMMARY
142                     print("  SUMMARY")
143                     print(f"    Signer    :
                            {result['Signer']}")
144                     status_text =
                            result['Summary'].split('(')[0].strip()
145                     print(f"    Status    : {status_text}")
146                     print(f"    Trusted   :
                            {result['Trusted']}")
147
148                     reason_text = result['ReasonInvalid']
149                     if status_text == "VALIDATION SKIPPED"
```

```
                    and not reason_text:
150                     reason_text = "No signer certificate
                            or unsupported signature"
151                 if reason_text:
152                     print(f"    Reason    :
                            {reason_text}")
153
154                 # PDF METADATA
155                 print("  PDF METADATA")
156                 print(f"    SubFilter     :
                        {md.get('SubFilter')}")
157                 # Print Reason and ContactInfo (if
                        present)
158                 print(f"    Reason        :
                        {md.get('Reason') or '-'}")
159                 print(f"    ContactInfo   :
                        {md.get('ContactInfo') or '-'}")
160                 # Format SigningDate
161                 signing_date_raw = md.get('SigningDate')
162                 if signing_date_raw:
163                     try:
164                         signing_date_clean =
                                signing_date_raw[2:] if
                                signing_date_raw
165                         .startswith('D:') else
                                signing_date_raw
166                         signing_date_clean =
                                signing_date_clean.replace("'",
                                "")
167                         signing_dt =
168                         signing_date_fmt =
                                signing_dt.strftime('%Y-%m-%d
                                %H:%M:%S%z')
169                         signing_date_fmt =
                                signing_date_fmt[:-2] + ':' +
                                signing_date_fmt[-2:]
170                     except Exception:
171                         signing_date_fmt =
                                signing_date_raw
172                 else:
173                     signing_date_fmt = '-'
174                 print(f"    Signing Date :
                        {signing_date_fmt}")
175                 print(f"    Location      :
                        {md.get('Location')}")
176                 print(f"    ByteRange     :
                        {md.get('ByteRange')}")
177
178                 # CERTIFICATE INFO
```

```
                    if cert_info:
                        print("  CERTIFICATE INFO")
                        for ck, cv in cert_info.items():
                            if not cv:
                                cv = "-"
                            print(f"    {ck:<20}: {cv}")

            return True, results

        except Exception as e:
            return False, f"Signature verification failed:
               {e}"
```

Its main operations can be summarized as follows:

- **PDF reading and signature extraction**: the function opens the PDF file using `PdfFileReader` and extracts all embedded signatures. If no signatures are found, it returns immediately.
- **ASN.1 certificate parsing**: for each signature the function retrieves associated X.509 certificate and parses key attributes, such as common name, organization, organizational unit, serial number, issuer and validity period. This gives us a full overview of certificates which were used to sign a PDF.
- **Normalization of PDF values**: the function safely converts PDF string-like objects to Python strings, handling cases where the object may not provide a direct value. This ensures metadata is extracted reliably without errors.
- **Signature validation**: performs cryptographic verification of each signature using the `pyHanko` library. It determines if the signature is valid, trusted or uses weak algorithms. Validation results are stored in a structured format, including reasons for untrusted or invalid signatures.
- **Metadata extraction**: alongside cryptographic validation, the function extracts signature metadata such as signer name, subfilter, reason, contact info, location, signing date and the `/ByteRange` of the signed content. The signing date is converted into a human-readable format.
- **Certificate information display**: if a certificate is available, detailed certificate information is printed, including subject and issuer attributes, validity period, and serial number.
- **Structured output**: the method returns the results in a list including all pertinent information for each signature, including validation status, trust level, metadata, certificate details, and reasons for invalidity. It prints a formatted forensic summary as well.

By combining cryptographic verification, certificate inspection and metadata extraction, this function provides a comprehensive assessment of digital signatures in

a PDF, which is essential for forensic analysis, document integrity verification and detecting potential tampering.

In general, `do_check_signatures` offers a solid solution for forensic investigation of the digitally signed PDF. By parsing the object coverage, recognizing unsigned objects or partly covered ones, and processing multiple signatures, it makes the investigators to be able to expediently evaluate reliability and integrity of each document. Overall, `do_check_signatures` provides a robust method for forensic analysis of digitally signed PDFs. By mapping object coverage, detecting unsigned or partially covered objects, and handling multiple signatures, it allows investigators to quickly assess the trustworthiness and integrity of the document.

Support for Report Generation

The functionality for supporting report generation is implemented through a *peepdf* wrapper script, designed to facilitate the use of *peepdf* within the container:

Listing 13: Wrapper script to run peepdfy from anywhere

```
#!/bin/bash
# Forensic wrapper for peepdf generating human-readable TXT report
set -euo pipefail

PDF_FOLDER="/app/pdfs/files"
FORENSIC_FOLDER="/app/pdfs/forensic_copy"
mkdir -p "$FORENSIC_FOLDER"

# --- Argument check ---
if [[ $# -eq 0 ]]; then
    echo "[!] Usage: peepdf -h | peepdf <filename.pdf>"
    exit 1
fi

ARG="$1"
shift

# Allow only help
if [[ "$ARG" == "-h" ]]; then
    cd /app/peepdf-3/peepdf || exit 1
    python3 peepdf.py -h
    exit 0
fi

# Reject additional flags
if [[ $# -ne 0 ]]; then
    echo "[!] Only 'peepdf -h' or 'peepdf <filename.pdf>' are allowed"
    exit 1
fi

```

```
# Ensure original file exists
ORIGINAL="$PDF_FOLDER/$ARG"
if [[ ! -f "$ORIGINAL" ]]; then
    echo "[!] File not found: $ORIGINAL"
    exit 1
fi

# --- Create forensic copy ---
COPY="$FORENSIC_FOLDER/$ARG"
cp "$ORIGINAL" "$COPY"

# --- Compute pre-analysis hashes ---
SHA256_BEFORE=$(sha256sum "$COPY" | awk '{print $1}')
MD5_BEFORE=$(md5sum "$COPY" | awk '{print $1}')

# --- Run peepdf and capture output ---
OUTPUT_LOG="$FORENSIC_FOLDER/${ARG}_peepdf.log"
cd /app/peepdf-3/peepdf || exit 1
script -q -c "python3 /app/peepdf-3/peepdf/peepdf.py -i
   '$COPY'" "$OUTPUT_LOG"

# --- Compute post-analysis hashes ---
SHA256_AFTER=$(sha256sum "$COPY" | awk '{print $1}')
MD5_AFTER=$(md5sum "$COPY" | awk '{print $1}')

# --- Prepare fetch_email section ---
FETCH_JSON="$FORENSIC_FOLDER/${ARG}.json"
if [[ -f "$FETCH_JSON" ]]; then
    FETCH_SECTION="=== FETCH_EMAIL METADATA ===
$(cat "$FETCH_JSON")"
    # Compute JSON hashes
    JSON_SHA256=$(sha256sum "$FETCH_JSON" | awk '{print $1}')
    JSON_MD5=$(md5sum "$FETCH_JSON" | awk '{print $1}')
    JSON_HASH_SECTION="JSON SHA256 : $JSON_SHA256
JSON MD5    : $JSON_MD5"
else
    FETCH_SECTION="=== FETCH_EMAIL METADATA ===
No fetch_email metadata available."
    JSON_HASH_SECTION="JSON SHA256 : N/A
JSON MD5    : N/A"
fi

# --- Prepare peepdf section (cleaned for report) ---
PEEPDF_SECTION="=== PEEPDF OUTPUT ===
$(sed -r 's/\x1B\[[0-9;]*[JKmsu]//g' "$OUTPUT_LOG")"

# --- Prepare forensic hashes section ---
HASH_SECTION="=== FORENSIC HASHES ===
Original file: $ORIGINAL
```

```
79  Forensic copy: $COPY
80  SHA256 before: $SHA256_BEFORE
81  MD5 before   : $MD5_BEFORE
82  SHA256 after : $SHA256_AFTER
83  MD5 after    : $MD5_AFTER
84  Hash match   : $( [[ "$SHA256_BEFORE" == "$SHA256_AFTER" &&
        "$MD5_BEFORE" == "$MD5_AFTER" ]] && echo "True" || echo
        "False")"
85
86  # --- Combine all sections into TXT report ---
87  REPORT_TXT="$FORENSIC_FOLDER/${ARG}_report.txt"
88  {
89      echo "$FETCH_SECTION"
90      echo
91      echo "$JSON_HASH_SECTION"
92      echo
93      echo "$PEEPDF_SECTION"
94      echo
95      echo "$HASH_SECTION"
96  } > "$REPORT_TXT"
97
98  # --- Compute hashes of the report itself ---
99  REPORT_SHA256=$(sha256sum "$REPORT_TXT" | awk '{print $1}')
100 REPORT_MD5=$(md5sum "$REPORT_TXT" | awk '{print $1}')
101
102 echo "[+] Forensic report saved: $REPORT_TXT"
103 echo "    SHA256: $REPORT_SHA256"
104 echo "    MD5   : $REPORT_MD5"
```

Reports are generated automatically for each interaction with *peepdf*. Each report is a `.txt` file containing a comprehensive summary of the processed PDFs, including metadata, forensic hashes and results from the `peepdf` analysis. The report is divided into several sections to facilitate forensic review and traceability. The layout is:

- **FETCH_EMAIL METADATA**: this section contains metadata generated by the email fetching process. If no metadata is available, a placeholder message such as `No fetch_email metadata available.` is displayed. For existing metadata, cryptographic hashes of the `.json` file are computed and reported in two fields: `JSON SHA256` and `JSON MD5`. If no metadata exists, placeholders such as `N/A` are used.
- **PEEPDF OUTPUT**: contains results from `peepdf` and interactions with the tool for each PDF file analyzed.
- **FORENSIC HASHES**: displays the SHA256 and MD5 hashes for both the original file and the forensic copy, before and after processing with *peepdf*, along with a hash match indicator. This provides integrity of evidence across the entire workflow.

This structure guarantees that each step of the forensic processing is fully documented, verifiable, and easily reviewable by experts. By including both metadata and file analysis results, the report provides a **transparent audit trail** for all PDF files processed by the system.

9.5 Generated Files Example

Each step of the process produces different files, each serving a specific purpose. The files generated depend on the workflow adopted when using `foredf`, but from a high-level perspective, as previously stated, the main file types that can be produced are the following:

- The `.json` file contains metadata about the emails from which PDFs are downloaded. For PEC (Posta Elettronica Certificata) messages, it includes metadata about both the original message and the wrapper `.eml` file. Its general structure is as follows:

```
{
  "uid": 1003,
  "original_filename": "document.pdf",
  "saved_filename": "example.eml_document.pdf",
  "content_type": "application/pdf",
  "subject": "Example Message",
  "from_header": [
    ["User Name", "user@example.com"]
  ],
  "headers": {
    "To": [["", "recipient@example.com"]],
    "Subject": "Example Email",
    "Date": "Thu, 18 Sep 2025 14:06:03 +0200"
  },
  "hashes": {
    "sha256": "REDACTED_FOR_PRIVACY",
    "md5": "REDACTED_FOR_PRIVACY"
  },
  # This field is present only in case of mismatch
  "sender_mismatch": {
    "from_header": "user@example.it",
    "envelope_sender": "user1@example.it>"
  }
}
```

- For each downloaded PDF, the enhanced version of `peepdf` can be used. Once executed, the user sees a general overview of the analyzed file, which also includes additional indicators about the file status, such as whether it is signed or not:

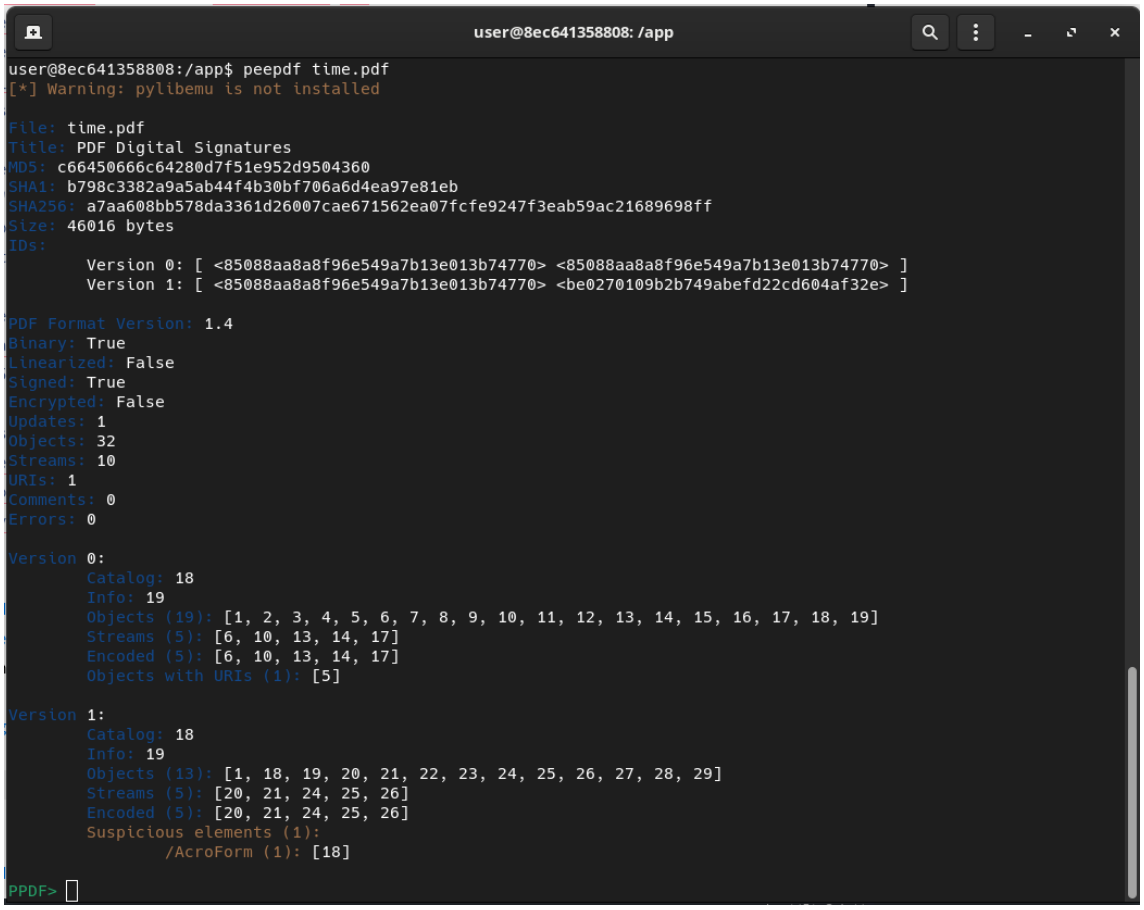

Figure 9: General overview of the analyzed file.

Since the file is signed, as we can see at line 11 of the Fig.9, the new command can be executed to check the signature. The resulting output will resemble the following, depending on the specific signature:

```
1  PPDF> check_signatures
2
3  Signature 1:
4    SUMMARY
5      Signer    : Unknown
6      Status    : VALIDATION ERROR
7      Trusted   : False
8      Reason    : The algorithm rsassa_pkcs1v15 is not
                   allowed by the current usage policy. Reason: Key
                   size 1024 for algorithm rsassa_pkcs1v15 is
                   considered too small; policy mandates >= 2048.
                   [AdESIndeterminate.CRYPTO_CONSTRAINTS_FAILURE]
9    PDF METADATA
10     SubFilter     : /adbe.pkcs7.detached
11     Reason        : I approve these details.
12     ContactInfo   : Contact info
13     Signing Date  : 2018-09-01 12:57:33-04:00
14     Location      : City
```

```
    ByteRange    : [0, 18374, 43144, 2872]
  CERTIFICATE INFO
    CommonName          : Test Signing Certificate
    Organization        : Tecxoft
    OrganizationalUnit  : Test
    SerialNumber        : 0x7a10d
    IssuerCN            : Tecxoft Public CA
    ValidFrom           : 2011-12-31 19:00:00+00:00
    ValidTo             : 2022-01-01 18:59:59+00:00
Object   Start      End         Signed?
5        15         142         YES
6        144        1036        YES
1        1038       1180        YES
9        1182       1268        YES
8        1270       1333        YES
10       1335       8117        YES
11       8119       8311        YES
12       8313       8613        YES
13       8615       8982        YES
2        8984       9120        YES
3        9122       9208        YES
14       9210       16295       YES
15       16297      16477       YES
16       16479      16732       YES
17       16734      17066       YES
4        17068      17195       YES
7        17197      17246       YES
18       17248      17307       YES
19       17309      17600       YES
27       18165      18349       YES
23       18351      43435       NO (Signature Object)
28       43437      43534       YES
29       43536      43611       YES
22       43613      43700       YES
20       43702      43948       YES
26       43950      44156       YES
25       44158      44378       YES
24       44380      44553       YES
21       44555      44904       YES
19       44906      45218       YES
18       45220      45401       YES
1        45403      45552       YES
```

If no object is marked as suspicious (i.e., none are highlighted in red), the tool still identifies the object that contains the signature information, along with its validation details and related metadata.

Furthermore, the analyzed PDF may include embedded files. In that case, something similar to the following output is displayed:

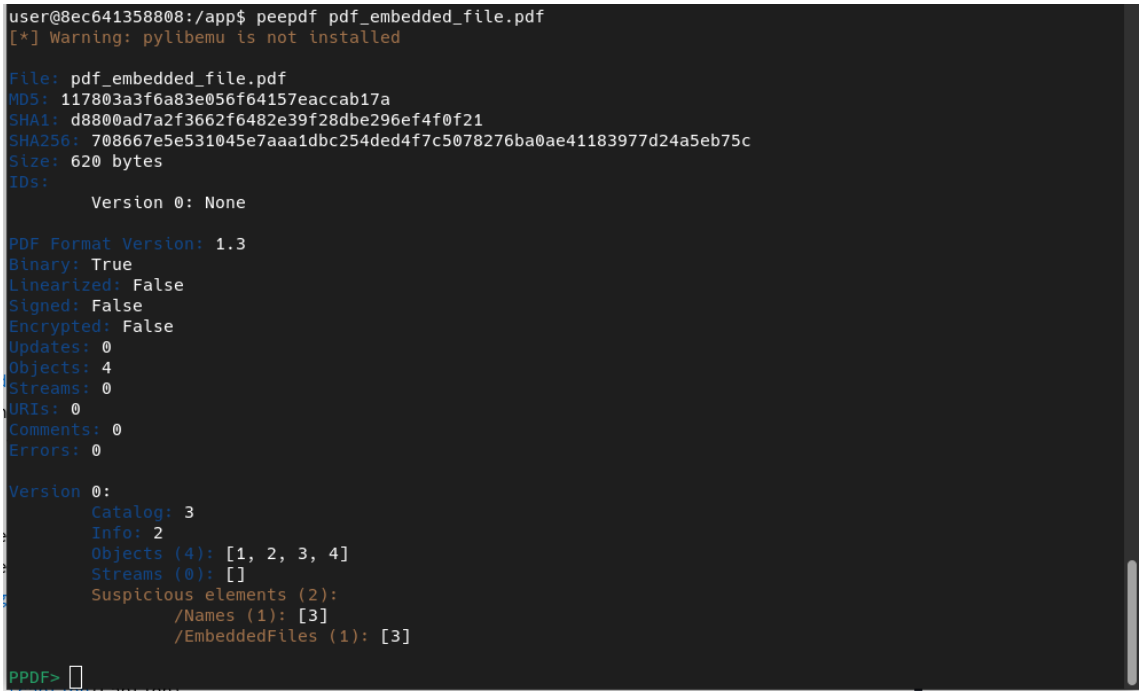

Figure 10: Indicator showing the presence of embedded files.

If the new functionality is executed, the tool provides a summary of the performed checks:

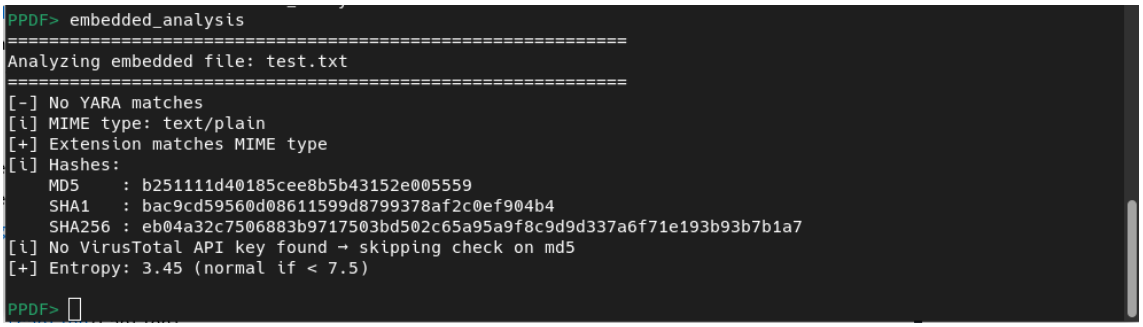

Figure 11: Analysis of embedded files.

In this case, the embedded file is not considered harmful, and the user can decide to trust it at their own risk since no indicators of potential threats are present.

- During the interaction with `peepdf`, a `.log` file is also generated. This file is later used to create preliminary forensic reports.

  Finally, based on all the collected information, the report is automatically saved at the end of the analysis. The user is notified with a message such as:

```
[+] Forensic report saved:
   /app/pdfs/forensic_copy/time.pdf_report.txt
SHA256: 50972e35ad4f1454d379060437da0d9adefe
```

```
6216591d64a3aa0bca33f5f30662
MD5    : 5f488230f43a399fbcce4b65e39783bd
```

In this way, the user can easily locate the generated report, verify and maintain the corresponding hash values for forensic integrity and traceability purposes.

9.6 Strengths and Weaknesses of foredf

The main **strength** of `foredf` lies in its ability to automate and integrate multiple forensic operations into a single, reproducible workflow. By extending the `peepdf` tool, it provides advanced features such as digital signature verification, embedded file inspection, automated metadata extraction from PEC and standard emails, and forensic report generation, all within a containerized and controlled environment. This unified design reduces the need for manual intervention and minimizes the fragmentation typical of current forensic workflows, ensuring higher reproducibility and forensic soundness. Furthermore, the integration of hashing, signature validation and scanning mechanisms (e.g. YARA and VirusTotal) strengthens its evidential reliability and broadens its analytical scope.

However, `foredf` still has certain **limitations**. Its reliance on static analysis restricts dynamic behavioral assessment, meaning it cannot directly observe runtime exploits or evasive code execution. In addition, while the containerized environment enhances portability and isolation, it introduces dependencies that may limit usability on constrained systems or environments without container support. Future developments could address these aspects by incorporating lightweight sandboxing capabilities and extending support for incremental report comparison and visualization.

10 Foredf: Attack Prevention and Analysis

To better illustrate the utility and functionality of **foredf**, this chapter presents several basic scenarios in which the tool plays a fundamental role. Each use case defines a specific context, explains how `foredf` can be applied to address the situation, and describes the expected outputs as well as any complementary analyses or tools that may be required to achieve a complete forensic analysis.

10.1 Use Case 1 — Malicious PDF Delivered via Corporate Email

Description. Employee gets a seemingly legitimate invoice or report in PDF format, with an embedded malicious payload (JavaScript, Shellcode or an embedded executable). If the PDF is opened on a workstation, this will exploit it, allowing to move laterally and compromise everything in case of execution.

Why it matters. Email is the top initial vector of attack – and as PDF files are widely accepted they often slip through the simplest filtering systems purely because they are generally perceived to be 'safe'.

How `foredf` could be used.

1. Before opening any PDF attachments in an email communication, **foredf** can fetch emails from specific mailboxes and extract PDF attachments.

2. The generated `.json` metadata file, useful for maintaining chain-of-custody and logging message headers, should be analyzed to verify sender and envelope consistency.

3. Once you have downloaded it, a static analysis with the improved `peepdf` module is able to find embedded Javascripts, obfuscated objects and streams. If the analyzed PDF has embedded files, they can be analyzed as well. Finally, `peepdf` can act as validation tool to establish if the file has been manipulated or even to check if a digital signature was invalidated (there are several ways to do that like adding some new objects which are not covered by the signature).

4. Finally, preliminary reports and hashes generated can be saved and used to flag high-risk files for immediate containment and dynamic analysis by the security team.

Expected results. Using **foredf** in this context allows a user to open a file only when its content has been verified, particularly when the sender is untrusted or email addresses may have been spoofed. This demonstrates the tool's strength in enabling employees to autonomously detect and analyze suspicious PDFs locally, without opening them directly and without requiring extensive resources. Moreover, all the supporting documents generated can serve also as a fundation for legal proceedings against the sender of that email. In addition, security team is not overloaded with unuseful requests since it can be alerted only when a file is considered really unsafe and flagged as dangerous.

However, `foredf` provides only static evidence; a follow-up dynamic sandbox analysis (e.g. with Cuckoo or Joe Sandbox) is recommended to observe runtime behavior and confirm payload activation and malicious intent.

10.2 Use Case 2 — Forged or Tampered Signed PDF in Legal/PEC Communications

Description. There is a certified email (PEC) with an, on the surface, signed contract. A party has challenged the authenticity of the signature or denied that the document was modified after it was signed.

Why it matters. Legal and forensic contexts require precise, object-level verification of signature coverage and proof that no document objects were altered following the signing event.

How `foredf` could be used.

1. Consume the PEC message and extract both the wrapper `.eml` file and the inner PDF, and some metadata of interest for the analysis of timing as well as sender authenticity.

2. Object-level digital signature verification with `peepdf` can be performed. This allows to visually see which objects are covered within and which are outside of the signature. If certain objects fall outside the signed range, incremental updates can be analyzed to determine whether the object existed at signing time or was added later. `peepdf` can also report discrepancies between the signed byte ranges and the current object map.

3. Finally, the tool can generate a structured report suitable for legal presentation, showing signed versus unsigned objects, temporal information about object insertion and signing and details of the certificate chain.

Expected outputs/evidence. Using the `check_signatures` functionality of `peepdf`, this scenario can be resolved efficiently by providing object-level proof of digital signatures, their coverage and their validity status.

However, if certificate revocation or PKI provenance requires deeper validation, integration with PKI validation services or consultation with legal experts is necessary. Moreover, while `foredf` provides object-level forensic evidence, it does not guarantee legal adjudication; every result must be independently verified, discussed and properly contextualized within legal procedures.

10.3 Use Case 3 — Post-Incident Investigation: PDF as Suspected Ransomware Vector

Description. A company gets hit with ransomware. A forensic triage indicates that it may have come from a PDF attachment which was the initial point of entry. Investigators must figure out if the PDF provided an initial entry point or delivered a malicious payload.

Why it matters. Identifying the true initial vector is critical for containment, eradication and remediation. PDFs are commonly used as stealthy delivery mechanisms that can evade basic security controls and may carry or link to malicious content.

How `foredf` could be used.

1. `foredf` can ingest the suspicious email(s) and their associated PDFs from a forensic image, mail archive or endpoint, preserving both `.eml` and PDF files.

2. Perform static analysis of the PDF to identify any anomalous structures, scripts or other indicators of suspicious behavior.

3. Write a structured forensic report by analyzing the PDF's elements, possible vulnerabilities and suspicious objects with `foredf` (using peepdf module).

Expected outputs/evidence. Using `foredf`, investigators obtain the original PDF, metadata for chain-of-custody, analysis logs detailing suspicious structures or behaviors and a comprehensive initial forensic report that can support incident response, remediation, real final reports and further investigative actions.

10.4 Use Case 4 — Mass Automated Triage in Security Operations Center (SOC)

Description. Big organizations may receive hundreds or thousands of emails daily; businesses depend on automated triage for PDF attachments to quickly assess threats.

Why it matters. Manual review is not intended for high volume environments, so the more this part of detection is automated, the faster the attack can be detected.

How `foredf` could be used.

1. Deploy containerized pipeline to poll monitored mailboxes and batch process PDF attachments.

2. For each PDF, generate metadata (`.json`) during fetching emails, later perform a static `peepdf` scan, analyze embedded files, any suspicious objects and perform digital signatures checks if any. In this case, the automation of the `peepdf` tool is required based on the security teams requirements.

3. Produce machine-readable reports or ticketing systems, ensuring traceable forensic artifacts.

Expected outputs/evidence. By using the extensibility of `foredf`, security teams can construct personalized containerized images suited to their organization's operational requirements. These iamges, when used, can generate aggregated forensic artifacts, including JSON metadata and `peepdf` analysis logs. They can be utilized to automatically rank suspicious PDFs while preserving a useful chain of custody if the PDFs are to be subjected to further forensic analysis or regulatory compliance auditing.

10.5 Use Case 5 — Post-Incident Investigation: Malicious PDF Delivered via USB

Description. During a forensic investigation, a compromised workstation is found to have connected a USB device. Preliminary evidence suggests that a PDF file on the USB may have been the vector for malware or ransomware infection. The forensic analyst must determine whether the PDF contributed to the compromise.

Why it matters. USBs are still one of the primary vectors for malware and that PDF files accessed from removable media can evade typical security controls. Analysis of the malicious file is crucial for incident containment, remediation and to gain insight into how was infection achieved.

How `foredf` could be used.

1. Acquire the USB device in a forensically sound manner and mount it in a controlled environment.

2. Extract PDF files from the USB and preserve original artifacts, including file metadata, timestamps and any related system logs.

3. Use `foredf` within a containerized reproducible environment to analyze each PDF, performing static inspection with `peepdf`, checking for anomalous structures, scripts, suspicious objects, embedded files and digital signatures if present.

4. Create a formal forensic report of the what you determine, all potential exposures and any proof that links the PDF back to an attack.

Expected outputs/evidence. From `foredf`, the examiner receives original PDF(s), associated metadata for chain-of-custody, analysis logs of suspicious behaviour or structures and a very good base to start to do forensic report. These outputs can be used by an analyst to verify if the PDF on USB was a part of the attack that may help remediation and incident documentation.

10.6 Mapping of `foredf` features to use cases

Feature	**Relevant Use Cases**
Automation & ingestion	Essential for Use Cases 1 (email fetch & triage), 4 (SOC automated processing), 5 (USB acquisition & batch processing).
Signature verification & object mapping	Essential for Use Case 2 (forged/tampered PDFs).
Static analysis of PDF structure	Relevant for Use Cases 1, 3, 5 (malicious content inspection, ransomware vector analysis).
Report & artifact preservation	Relevant for Use Cases 1, 2, 3, 4, 5 (forensic logs, chain-of-custody, structured reports).

Table 2: Mapping of `foredf` features to the described use cases

11 Conclusions and Future Works

The work carried out in this project highlights the practical and methodological significance of the developed tool, especially in areas that require reproducibility, transparency and standardization of methodology. In the context of investigations, audits or regulations, it is a basic necessity to be able to replicate results in the same manner. The described tool will support these methods directly as it automatizes a chain of clear steps, which ensures consistency while human action is minimized. This systematic approach enables results to be reproduced, read and compared in a way compatible with best practice and standards in open scientific data.

The features provided in the current version have proven to be highly useful in real scenarios. The tool also accepts standard input and output types, making it straightforward to include in larger processing pipelines or collaborative environments. The modular design makes it easy to maintain and extend, while its lightweight reporting and logging mechanisms provide essential feedback on operations, errors, and results. These features combined make the tool a powerful and versatile tool that balances ease of use with methodological strength.

The utility of this tool is not only self-evident but it also also promotes the adoption of standardized workflows across different environments. It therefore improves the comparability of results obtained by different experimenters at different laboratories, something particularly crucial in fields where reproducibility is increasingly recognized as a scientific and ethical imperative. In addition, the open and extensible design can be easily adapted to future standards or domain-specific requirements for long time sustainability.

For the future, there are several potential developments. One of the steps would be to improve reporting system: instead of simple `.txt` summaries, the generation of structured and visually enriched reports (e.g. in PDF or HTML format) would provide a more comprehensive overview of the operations performed and their outcomes. Furthermore, the inclusion of `sandboxing` mechanisms would improve security and reproducibility and enable the tool to recognize malicious files based on their runtime behavior.

Another promising direction is the integration of machine learning components to further exploit the outcomes produced by `foredf` during batch analyses of PDF files. Instead of directly inspecting the documents, ML models could be applied to the structured results generated by the tool, such as extracted features, metadata and analytical indicators, to detect patterns, anomalies, or suspicious behaviors across large datasets. This approach would enable automatic prioritization of potentially problematic cases, improve the interpretability of large-scale analyses, and support decision-making processes by identifying trends that may not be immediately evident through manual inspection. Such functionality would therefore not only enhance efficiency but also contribute significantly to data integrity and overall analytical robustness.

Some longer-term ideas include adding a GUI (graphical user interface) so that it can be used by non-technical users, while others are having detailed configuration profiles to better fit the tool in diverse workflows. Following these directions, the tool could then become an integrated secure framework not only for safe and smart

applications deployment, but also to support reproducible and standardized data processing at scale.

In conclusion, this project establishes a solid foundation for reliable, reproducible, and standardized document processing. Through continued refinement, expansion, and integration of advanced features, the tool can become an indispensable asset for both researchers and practitioners who value transparency, repeatability, and methodological precision.

References

[1] A. Abdallah et al. “A Survey on Malicious PDF Detection: Static, Dynamic, and Hybrid Approaches”. In: *Journal of Information Security and Applications* (2025). URL: https://www.sciencedirect.com/science/article/pii/S1877050925008798.

[2] Adobe Systems Incorporated. *PDF Reference, Sixth Edition: Adobe Portable Document Format Version 1.7*. Adobe Technical Reference Manual. Nov. 2006. URL: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/pdfreference1.7old.pdf.

[3] Adobe Systems Incorporated, Tim Bienz, and Richard Cohn. *Portable Document Format Reference Manual*. First Printing, June 1993. Reading, Massachusetts: Addison-Wesley, 1993. ISBN: 0-201-62628-4.

[4] X-Ways Software Technology AG. *X-Ways Forensics*. Integrated computer forensics software based on WinHex. URL: https://www.x-ways.net/forensics/index-m.html.

[5] Agenzia per l’Italia Digitale. *Linee guida per la gestione e conservazione della Posta Elettronica Certificata*. Describes PEC operational rules, retention, and log management for certified providers. 2021. URL: https://www.agid.gov.it/it/linee-guida/pec.

[6] Mehran Alidoost Nia, Ali Sajedi, and Aryo Jamshidpey. “An Introduction to Digital Signature Schemes”. In: *arXiv preprint arXiv:1404.2820* (2014). URL: https://arxiv.org/abs/1404.2820.

[7] *ANY.RUN Interactive Malware Analysis*. https://any.run/. Online service for interactive malware sandbox analysis.

[8] Autopsy. *Autopsy - Digital Forensics*. Open-source digital forensics platform developed by Sleuth Kit Labs. URL: https://www.autopsy.com/.

[9] Monya Baker. “1,500 scientists lift the lid on reproducibility”. In: *Nature* (2016). DOI: 10.1038/533452a.

[10] Buffer. *Pylibemu*. https://cybersectools.com/tools/pylibemu. 2025.

[11] Cloudflare. *STPyV8: Python 3 and JavaScript Interoperability*. https://github.com/cloudflare/stpyv8. 2025.

[12] Cognyte. *Emerging Trends and Technologies in Digital Forensics*. 2025. URL: https://www.cognyte.com/blog/digital-forensics-investigations/.

[13] M. Crispin. *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. RFC 3501. Mar. 2003. URL: https://datatracker.ietf.org/doc/html/rfc3501.

[14] David H. Crocker. *Standard for the Format of ARPA Internet Text Messages*. RFC 822. Obsoleted by RFC 5322. Aug. 1982. DOI: 10.17487/RFC0822. URL: https://datatracker.ietf.org/doc/html/rfc822.

[15] *Cuckoo Sandbox*. https://github.com/cuckoosandbox. Open-source automated malware analysis system.

[16] Guillaume Delugré. *Origami Framework for PDF analysis and creation*. https://code.google.com/archive/p/origami-pdf/.

[17] Docker, Inc. *Docker*. https://www.docker.com/. 2025.

[18] Docker, Inc. *Docker Compose: Define and Run Multi-Container Applications*. https://docs.docker.com/compose/. 2025.

[19] *Document Management — Portable Document Format — Part 2: PDF 2.0*. International Standard. Developed by ISO/TC 171/SC 2/WG 8. Geneva, Switzerland: International Organization for Standardization, 2020.

[20] José Miguel Esparza. *peepdf – Python tool to analyze PDF files*. https://github.com/jesparza/peepdf. 2012.

[21] PDF Examiner. *PDFExaminer Tool - Analyse PDF Malware*. https://github.com/tylabs/pdfexaminer. 2021.

[22] Exterro. *Forensic Toolkit (FTK)*. Commercial digital forensics software formerly developed by AccessData. URL: https://www.exterro.com/forensic-toolkit.

[23] Oxygen Forensics. *CEO Lee Reiber: The Digital Forensics Landscape in 2025*. 2025. URL: https://www.oxygenforensics.com/en/resources/digital-forensics-trends-2025/.

[24] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045. Nov. 1996. URL: https://datatracker.ietf.org/doc/html/rfc2045.

[25] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. Nov. 1996. URL: https://datatracker.ietf.org/doc/html/rfc2046.

[26] Google Inc. *PyV8*. https://code.google.com/archive/p/pyv8. 2010.

[27] Christopher Hargreaves et al. "DFPulse: The 2024 digital forensic practitioner survey". In: *Forensic Science International: Digital Investigation* (2024). DOI: https://doi.org/10.1016/j.fsidi.2024.301844. URL: https://www.sciencedirect.com/science/article/pii/S2666281724001719.

[28] *ISO/IEC 27037:2012 – Guidelines for Identification, Collection, Acquisition and Preservation of Digital Evidence*. International Organization for Standardization, 2012.

[29] A. Javed et al. "A Comprehensive Survey on Computer Forensics: State of the Art, Tools, Techniques, Challenges, and Future Directions". In: *CDU Researchers Journal* (2022). URL: https://researchers.cdu.edu.au/en/publications/a-comprehensive-survey-on-computer-forensics-state-of-the-art-too.

[30] *Joe Sandbox*. https://www.joesecurity.org/. Commercial malware analysis and sandboxing platform.

[31] Karen Kent et al. *Guide to Integrating Forensic Techniques into Incident Response*. Tech. rep. NIST SP 800-86. National Institute of Standards and Technology, 2006. URL: https://csrc.nist.gov/pubs/sp/800/86/final.

[32] S. Kitterman. *Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1*. RFC 7208. Apr. 2014. URL: https://datatracker.ietf.org/doc/html/rfc7208.

[33] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. Defines the core SMTP protocol used for email transmission. Oct. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5321.

[34] M. Kucherawy and E. Crocker. *DomainKeys Identified Mail (DKIM) Signatures*. RFC 6376. Sept. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6376.

[35] M. Kucherawy and E. Zwicky. *Domain-based Message Authentication, Reporting, and Conformance (DMARC)*. RFC 7489. Mar. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7489.

[36] X. Liu et al. "Analyzing PDFs like Binaries: Adversarially Robust PDF Malware Analysis via Intermediate Representation and Language Model". In: *arXiv preprint arXiv:2506.17162* (2025). URL: https://arxiv.org/abs/2506.17162.

[37] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1996. ISBN: 9780849385230.

[38] Michele Merico. *foredf: Forensic tool which allows users to fetch emails and analyze them with an enhanced version of peepdf*. https://github.com/emfourem/foredf. 2025.

[39] D. Müller et al. "Shadow Attacks: Hiding and Replacing Content in Signed PDFs". In: 2020. URL: https://www.pdf-insecurity.org/.

[40] J. Myers and M. Rose. *Post Office Protocol - Version 3*. RFC 1939. May 1996. URL: https://datatracker.ietf.org/doc/html/rfc1939.

[41] National Institute of Standards and Technology (NIST). *CVE-2008-2992: Stack-based buffer overflow in Adobe Acrobat and Reader 8.1.2 and earlier*. 2008. URL: https://nvd.nist.gov/vuln/detail/CVE-2008-2992.

[42] National Institute of Standards and Technology (NIST). *CVE-2013-3346: Adobe Reader and Acrobat JavaScript Memory Corruption Vulnerability*. National Vulnerability Database (NVD). https://nvd.nist.gov/vuln/detail/CVE-2013-3346. 2013. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-3346.

[43] National Institute of Standards and Technology (NIST). *CVE-2015-1282: Multiple use-after-free vulnerabilities in PDFium*. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-1282.

[44] *PEC: Posta Elettronica Certificata*. Agenzia per l'Italia Digitale. Available at https://www.agid.gov.it/it/piattaforme/posta-elettronica-certificata. 2014.

[45] Red Hat. *What is containerization?* 2021. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization.

[46] P. Resnick. *Internet Message Format*. RFC 5322. Specifies the format of email message headers and bodies. Oct. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5322.

[47] Rev.com. *Digital Evidence in the Age of Virtual Trials*. 2024. URL: https://www.rev.com/blog/digital-evidence.

[48] B. Russell and G. Trostle. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. https://datatracker.ietf.org/doc/html/rfc2315. Mar. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2315.

[49] National Institute of Standards and Technology (NIST). *CVE-2018-18689: Signature Wrapping Vulnerability in Foxit Reader and PhantomPDF*. 2018. URL: https://nvd.nist.gov/vuln/detail/CVE-2018-18689.

[50] Italian State. *DPCM 2 novembre 2005: Regole tecniche per la formazione, la trasmissione e la validazione, anche temporale, della posta elettronica certificata*. Gazzetta Ufficiale della Repubblica Italiana. 2005. URL: https://www.agid.gov.it/sites/default/files/repository_files/leggi_decreti_direttive/dm_2-nov-2005.pdf.

[51] Didier Stevens. *PDF Tools*. https://blog.didierstevens.com/programs/pdf-tools/. 2009.

[52] American Military University. *How Is Digital Evidence Preserved in Modern Investigations?* 2025. URL: https://www.amu.apus.edu/area-of-study/criminal-justice/resources/how-is-digital-evidence-preserved/.

[53] Unknown author. *Digital Signature diagram*. https://upload.wikimedia.org/wikipedia/commons/2/2b/Digital_Signature_diagram.svg. Licensed under CC BY-SA 3.0 via Wikimedia Commons. 2012.