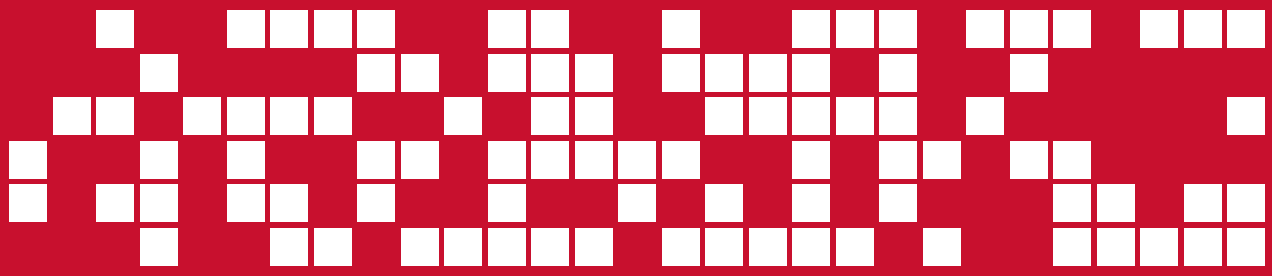
ETH ZÜRICH • D-INFK

DISS. ETH NO. 32483



Matteo
Scarlata

How Not to Encrypt

An Anthology of Cryptographic Attacks

DISS. ETH NO. 32483

HOW NOT TO ENCRYPT: AN ANTHOLOGY OF CRYPTOGRAPHIC ATTACKS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

MATTEO SCARLATA

born on 22.11.1995

accepted on the recommendation of

Prof. Dr. Kenny Paterson
Prof. Dr. Benjamin Dowling
Prof. Dr. Nadia Heninger
Prof. Dr. Juraj Somorovsky

2026

This dissertation was prepared with pdfLaTeX using a template designed by Timothy Gebhard.
The main text is set in Libertine. The sans-serif font is Gillius. The monospace font is Bera Mono.

Abstract

Attacks have marked much of the evolution of cryptography. *Provable security* formally defines the security goals and the threat model of a cryptographic system with the aim of mathematically guaranteeing that it is safe from attacks. *Cryptanalysis* uses much of the same tooling to *break* cryptographic systems. This is not a purely destructive process: as we will argue, the presence of attacks is an important guiding principle for cryptographic development.

In this thesis, we look at three examples of cryptography *in the wild*: cryptographic systems that have been deployed in practice, or proposed for practical deployment, often without much prior academic scrutiny. We study an encrypted messaging application, Threema; a proposal for a key derivation framework, MFKDF; and four widely used password managers, Bitwarden, LastPass, Dashlane and 1Password. Against all of these systems, we show practical attacks that undermine many of the security guarantees these systems were intended to provide. These range from metadata leakage, conversation tampering and key extraction attacks against Threema, to key recovery and entropy reduction attacks against MFKDF, to many attacks that leak passwords stored within password managers. For each of the systems, we started by modelling its cryptographic core and defining threat models. Most of the attacks emerged as conflicts between the expected security of the systems in a given threat model, and the concrete security they delivered. After this cryptanalysis phase, we designed and proposed mitigations and disclosed our attacks to the affected vendors. During disclosure, we often worked together directly with the vendors to review deployed mitigations, and coordinated on disclosure timelines to minimise harm.

We frame all of these attacks as case studies on *how (not) to encrypt*: that is, how not to build cryptographic systems in the broad sense. We draw lessons, extracting threat modelling, design and implementation principles to follow in order to largely avoid falling into the same mistakes. Even as we move towards more provable security in deployed systems, cryptanalysis of applications in the wild continues to provide important input for building cryptography. We show how attacks test the concrete security guarantees a system delivers, probing outside the boundaries of formal proofs. Each of our case studies, on its own, highlights some of these lessons. This thesis synthesises the common threads, in order to provide concrete guidance for designers and implementers of cryptographic systems. We therefore show *how to encrypt*, building on a contrapositive library of examples in the form of attacks.

Contents

Abstract vii

Contents ix

Notation xi

1 Introduction 1
- 1.1 Thesis outline and contributions . . . 5
- 1.2 Publications . . . 8

2 Three Lessons From Threema: Analysis of a Secure Messenger 9
- 2.1 Introduction . . . 10
- 2.2 Secure Messaging . . . 13
- 2.3 Threema Architecture and Protocols . . . 16
- Tabular Summary of Attacks . . . 24
- 2.4 Attacks on Threema . . . 25
- 2.5 Mitigations . . . 36
- 2.6 Conclusions . . . 39

3 MFKDF: Multiple Factors Knocked Down Flat 41
- 3.1 Introduction . . . 42
- 3.2 Background . . . 47
- Tabular Summary of Attacks . . . 57
- 3.3 Attacks . . . 57
- 3.4 Mitigations . . . 72
- 3.5 Discussion . . . 74
- 3.6 Conclusions . . . 76

4 A Comparative Security Analysis of Four Cloud-Based Password Managers 79
- 4.1 Introduction . . . 80
- 4.2 Threat Model . . . 85
- 4.3 The Password Managers . . . 88
- 4.4 Attacks . . . 93
- Tabular Summary of Attacks . . . 94
- 4.5 Mitigations . . . 115
- 4.6 Ethical Considerations . . . 119
- 4.7 Open Science . . . 121
- 4.8 Discussion . . . 122

4.9 Conclusions 124

5 Conclusions – How to Encrypt 125
5.1 Threat Modelling 126
5.2 Cryptographic Design 133
5.3 Implementation 137
5.4 Epilogue 139

A An Opinionated Selection of Primitives 141

Bibliography 143

CV 167

Acknowledgments 169

Notation

The following notation is used throughout this thesis. Chapter-specific notation is introduced in the respective chapters.

Symbol	Description
General	
$\{0,1\}^n$	The set of bit strings of length n
$\lvert x\rvert$	Length of string x
$\Vert$	String concatenation
$\oplus$	Bitwise exclusive-or (XOR)
$a \leftarrow x$	Assign value x to variable a
$a \xleftarrow{\$} \mathcal{X}$	Sample a uniformly at random from set $\mathcal{X}$
$\perp$	Failure / rejection symbol
$\mathbb{N}$	The natural numbers
Cryptographic Primitives	
$\mathsf{Enc}(k, m)$, $\mathsf{Dec}(k, c)$	Symmetric encryption / decryption
$\mathsf{Enc}(pk, m)$, $\mathsf{Dec}(sk, c)$	Public key encryption / decryption
$\mathsf{E}_K(m; n)$, $\mathsf{D}_K(c; n)$	AEAD encryption / decryption with nonce n
KeyGen	Key generation algorithm
$\mathsf{KDF}(K, \sigma, \tau)$	Key derivation function with key material K, salt σ, label τ
PBKDF	Password-based key derivation function
HKDF	HMAC-based key derivation function
HMAC	Hash-based message authentication code
g	Generator of a cyclic group
g^a	Public key corresponding to private key a
Keys and Parties	
$\mathcal{A}, \mathcal{B}, \mathcal{U}$	Protocol participants (users)
$\mathcal{S}$	Server
(a, A)	Private / public key pair of user $\mathcal{A}$
(x, X), (y, Y)	Ephemeral key pairs
sk, pk	Secret key, public key (generic)
K	Symmetric key (session key, AEAD key, or derived key)

Chapter 1

Introduction

> The development of cheap digital hardware [...] brought the cost of high grade cryptographic devices down to where they can be used in such commercial applications as remote cash dispensers and computer terminals. In turn, such applications create a need for new types of cryptographic systems which minimise the necessity of secure key distribution channels and supply the equivalent of a written signature.
> —W. Diffie, M. E. Hellman, 1976

We stood on the brink of a revolution in cryptography. And the revolution came: the price of computers sank, their size shrank. Computation became ubiquitous—we find it today not just in every shop and in every house, but on every person, in the form of phones and smartwatches. Cryptography flourished. The prophecy of Diffie and Hellman is now realised in full, and today's digital reality is closer to what cyberpunk novels had envisioned than anybody had any right to believe.

We *trust* computers with much of our lives: as computers integrate more and more with our physical world, the physical and digital keep blurring. We communicate, shop, invest, get our news, pay our bills—all with computers. Secure communication, digital signatures, and encryption are a ubiquitous, if invisible, part of our everyday use of technology.

The road to *secure* cryptosystems, though, was not paved with successes. Early academic cryptography evolved disjoint from practical work. Foundational work on mathematically secure cryptography had little impact, both because the results were hard for practitioners to understand and appreciate, and because the schemes we could prove secure were not practical enough.[1] This left us with a split community, and a large amount of non-proven cryptography. The troubled history of TLS, the protocol used to secure much of the Internet, is exemplary. The first iterations of the protocol were known as SSL. SSLv1, from 1995, was so broken that it never saw the light of day; SSLv2 used a weak hashing primitive (MD5), and did not authenticate its handshake, allowing a man-in-the-middle adversary to prematurely terminate a connection or to downgrade the negotiated cipher suite. SSLv3 was plagued by padding-oracle attacks (from the early ones by Vaudenay [284] to the "POODLE" attack [216]), and the only stream cipher it supported, RC4, turned out to be badly broken too [23, 283, 127]. The naming convention switched to TLS in 1999, but that didn't significantly shift the balance. TLS 1.0 traffic could be decrypted thanks to predictable initialisation vec-

[1] For instance, the Goldwasser–Micali cryptosystem [144], known as the first public-key encryption scheme to come with a proof of security, encrypts each bit of the plaintext as a quadratic (non-)residue for a large RSA modulus. This results in a massive *ciphertext expansion*: the ciphertext is, with modern RSA parameters, thousands of times bigger than the plaintext.

tors (the "BEAST" attack [250]); TLS 1.1 was still vulnerable to padding-oracle timing side channels ("Lucky 13" [24]); and even 2008's TLS 1.2, a major cleanup, still supported by many browsers and servers at the time of writing, was a constant source of attacks, ranging from "triple handshakes" [52] to the negotiation of weak Diffie-Hellman groups with "LOGJAM" [9] and a renegotiation flaw that let an attacker inject chosen plaintext into a client's session [247].

Across the board, support for RSA PKCS #1 v1.5 as a key transport primitive caused many problems, ranging from Bleichenbacher's original attack [61] to its cross-protocol instantiation DROWN [32], its "return" ROBOT [62], and cache side channels [254], among others [213]. The key exchange fared no better: beyond LOGJAM, TLS-ECDH succumbed to invalid-curve attacks [173], TLS-DH(E) leaked secrets through most-significant-bit timing oracles ("Raccoon" [212]), and sharing a certificate across application protocols enabled cross-protocol confusion ("ALPACA" [71]). Block ciphers with small block sizes were shown to be dangerous ("Sweet32" [53]), as was the mere support for weak hash primitives, because of downgrade attacks targeting collision resistance like SLOTH [54]; and data compression turned TLS against itself, leaking secrets through ciphertext length ("CRIME" and "BREACH" [251, 133]). Finally, the sheer complexity of the TLS specification was a source of both state-machine attacks [51] and implementation flaws, from nonce reuse in GCM [63] to memory-safety bugs ("Heartbleed" [111]) and broken certificate validation (Apple's "goto fail" [189])—a class of bugs that automated testing keeps uncovering to this day [200].

Why Attacks? The antidote to such an unending stream of attacks is, as any modern cryptographer knows, provable security. The tide eventually turned, and the gap between academic and practical cryptography began to narrow. Throughout the decades, more and more provably secure primitives were standardized. And in 2018, we saw the first version of TLS which was not only built on top of secure primitives, but also entirely designed with formal support, in several rounds of collaboration between academia, industry and the Internet Engineering Task Force (IETF): TLS 1.3. This represented a paradigm shift: from releasing protocols, breaking them, and releasing patches, to analysing protocols and updating them until they can be proven secure well before release [227]. We know how to do provable security, now. So why look for attacks, rather than build new secure cryptography?

One answer is that attacks themselves are, in a way, a proof. Not a proof of security, but the opposite. TODO: **given a certain threat model and security goal, the existance of an** An attack is a proof by contrapositive: the existence of attacks proves that a protocol is *not secure*. And yet, proving that an existing cryptosystem is broken does not appear to, in isolation, move us closer to having secure cryptography. The question remains: what is, ultimately, the value of cryptanalysis? Why break instead of build?

On the Virtue of Cryptanalysis. While it is evident that an attack is proof of insecurity, the converse proposition is well known to be a logical fallacy. Absence of attacks does not *prove* that something is secure, much like absence of evidence is not evidence of absence. On the other hand, prolonged absence of attacks should (and does) shift our belief in favour of security. After all, we are doing *science*: when there are theories that we cannot mathematically verify, we must proceed by Bayesian inference.

This is a somewhat uncomfortable truth that provable security has to face. A proof *reduces* the security of a system to the security of the underlying primitives, but we eventually hit the point where our primitives and building blocks cannot be reduced further. There, we must stop and just *assume* that some computational problem is hard, for lack of a known way to solve it. This is currently the case for most of the fundamental building blocks of modern cryptography: they rest on unproven assumptions. Examples are the PRF security of AES, the hardness of factoring and of the discrete logarithm problem, and more recently the hardness of decoding random linear codes (and, relatedly, of the learning with errors problem). While some progress has been made on these foundational questions—for instance, showing that AES-like substitution-permutation networks are provably close to t-wise independent [195, 194], that discrete-logarithm-based assumptions can be justified in idealized models such as the generic and algebraic group models [268, 126], or that the average-case hardness of the learning with errors problem follows from the worst-case hardness of lattice problems [244]—these underlying assumptions remain essentially unproven. At the time this thesis is being written, our knowledge of computational complexity is still incomplete: we cannot currently prove that cryptographic assumptions are hard, and cannot build cryptography out of problems that we know are hard. The only evidence we have that secure cryptography can be built lies in the absence of cryptanalytic results.

There is, therefore, inherent value in breaking things. Lest we discover, one day, to have been in Impagliazzo's *Algorithmica* or *Heuristica* all along—without having noticed, for lack of trying.[2]

On Societal Impact and Industry Inertia. We have established that cryptanalysis has value for testing the hardness of fundamental assumptions, but this thesis has no pretence of (attempting to) break any of those. The subjects of our attacks are much more mundane: we analyse messaging applications, key derivation frameworks, and password managers. Where's the value in breaking cryptographic *applications*?

The answer is twofold. First, these applications exist already, and there is little incentive to shift to provably secure alternatives in the industry. Software is expensive; rewriting to adopt a new protocol is not a minor effort. Developers have egos and grow attached to their creations. Backwards compatibility is a genuinely hard problem; it is unrealistic to simultaneously switch a big user base to a new version of an application. Thus, we need attacks. They create strong incentives to shift to a more secure protocol. Even partial or theoretical breaks are

[2] Russell Impagliazzo analyses the average case complexity of problems in NP, and proposes five possible worlds [166]: *Algorithmica*, where P = NP; *Heuristica*, where NP problems are hard in the worst case but easy on average; *Pessiland*, where NP problems are hard on average but no one-way function exists; *Minicrypt*, where one-way functions exist; and *Cryptomania*, where public-key cryptography is possible. The first two worlds preclude the existence of cryptography, since problems would be too easy for any cryptographic reduction to rest on solid foundations. This would imply that we are sitting on an enormous, undiscovered trove of attacks.

important. "*Attacks only get better*",[3] so not acting against a minor attack imposes future costs when the attack improves; over time, abandoning an old protocol altogether in favour of a provably secure alternative might even be the cheapest option.

[3] Old saying among cryptographers, attributed to the NSA by Bruce Schneier [181].

Second, these (possibly broken) cryptographic applications are in the hands of millions of users. We need attacks to understand the *concrete* security guarantees that these systems provide. This is important in the case a security analysis is absent, because the security of the system is actually unknown. It is as important, though, in the case a formal proof of security exists, because even the most extensive proofs only capture a subset of the entire technological stack. Unexpected interactions and surprising behaviours can emerge both at the very low level of software and hardware (e.g., side channels), and at the high level of application design (e.g., user interface mismatches and confusing user alerts).

On Proof Boundaries. We have now enunciated another important reason for cryptanalytic work: checking what security guarantees a system actually delivers, even when a security proof exists.

Formal security proofs offer strong assurance, but they are almost never all-encompassing. Modern cryptographic libraries are mostly composed of strong, proven cryptographic primitives. Their security holds in isolation. For example, a secure encryption scheme guarantees confidentiality in itself. But a proof for this property says nothing about how the cipher is used in the context of a wider system: if, say, the key for the encryption scheme is incorrectly derived, or the same key is used across multiple primitives, the provable security guarantees are moot. In general, the composition of secure building blocks does not yield a secure system. A classic example of this emerges when combining public-key encryption and signatures, while using the same key pair for both. In some cases, such as for RSA, the operations for encryption and signature are dual: a signature, then, can provide a trivial decryption oracle when the private key used for signing a ciphertext matches the public key used for encrypting the corresponding plaintext message.

Furthermore, even proofs that capture whole systems have to draw a boundary somewhere. Very often, proofs model cryptographic primitives as abstract objects: they would then miss attacks introduced by low-level architectural features, like speculative execution (GoFetch [77]), physical RAM layout (Flip Feng Shui [242]) or cache timing (from attacks on AES [48] and OpenSSL [233] to CacheBleed [289]).

Proofs also don't extend all the way to the user interface and the application design: this can result in applications that are concretely broken, even when the underlying cryptographic core is otherwise solid. Proofs for a protocol may miss interactions with other subprotocols that extend it. Notable cases of this kind of attack are the recent attacks on Signal [280], exploiting implementation bugs in the Android application and issues emerging from the composition of the Signal Double-Ratchet protocol with the Sealed Sender feature. On the TLS side, the otherwise unscathed TLS 1.3 suffered attacks on the resumption mechanism [155,

156], which is orthogonal to the secure channel component and not well-specified in the standard documents.

On Having a Hammer, and Many Nails. The justifications above are, clearly, somewhat post-hoc: while the author has been convinced of the societal good of his work all along, this alone would probably not have been sufficient motivation to do it. The truth is that there is a certain acquired taste for good cryptography that comes from sampling a large number of cryptosystems. It becomes easy to spot repeated errors in new protocols and unpalatable combinations of cryptography that are likely to lead to attacks. One builds a *toolbox* for cryptographic analysis of protocols, and starts exercising it, again and again. Targets are abundant, and cryptanalysis is fun. Why not break one more thing? And how to resist the temptation to break something, once one catches a whiff of it being broken? As we will see, even if the motivation for the work is selfish fun, value can be extracted from the results.

How Not To Encrypt? In this thesis, we will explore cryptographic attacks. Beyond the intrinsic and societal value, and the author's own delight, can we learn something from studying how cryptography went wrong? We believe that the answer is yes, and that attacks can and do inform the work of practitioners when it comes to deploying new cryptography. In this spirit, all of the case studies we present discuss *lessons*: what we have learnt from erring. We see this as the main contribution of our work: distilling bad cryptographic practice, so that, hopefully, we won't build the same mistakes into our designs again.

1.1 THESIS OUTLINE AND CONTRIBUTIONS

The next chapters will cover three different pieces of cryptanalytic work, which we treat as case studies for our study of cryptographic malpractice. We will draw overall lessons and conclude in Chapter 5.

We begin in Chapter 2 with the security analysis of Threema, a Swiss encrypted messaging application. Threema counts over 10 million users, and is widely used in businesses across Switzerland and German-speaking Europe; it is also the messaging application of choice of the Swiss army. Secure messaging is a particularly demanding setting for cryptography: sessions of a secure messaging protocol (that is, instances of the protocol between two communicating users) are asynchronous (communicating users are not required to be online at the same time) and long-lived (potentially spanning several years). Furthermore, messaging often takes place on mobile devices, which have a variety of possible hardware limitations, and a peculiar security posture, given the risk of scans at border crossings. All of these properties are hard to test for. It is hard to tell from the surface whether one secure messaging application achieves better security than another.

This chapter is a deep dive into the bespoke cryptographic protocols that secure Threema. We are not talking about *one* protocol, but protocol*s*: a messaging application's core is, clearly, the protocol protecting

end-to-end communication of its users. There is also a protocol to protect client-to-server communication, and protocols for registration and contact discovery; all of these can interact in surprising ways. Threema had not been studied before, making it a ripe target for cryptanalysis.

Our analysis did not break Threema badly enough to expose the contents of its users' conversations. But that is a low bar for a modern messenger. Because sessions are so long-lived, we want additional properties: forward secrecy, so that a compromise of the key material of a session does not affect the security of previous messages; post-compromise security, so that even after such a compromise, our conversations regain security; and robustness against compromise of ephemeral values and against an adversary with physical access to the device. Against those stronger goals we find Threema lacking: it offers neither post-compromise security, nor forward secrecy at the end-to-end layer. If the Threema server records encrypted conversations (whether by malice, or compelled by law enforcement action, like in the case of Encrochat [21]), these records can be decrypted as soon as the keys of a user are compromised. Beyond these structural weaknesses, we present seven attacks across three threat models, including several that exploit different kinds of ephemeral compromise, together with a number of low-impact but surprising breaks of authentication and integrity guarantees of both the client-to-server and end-to-end protocols. One of these attacks was only made practical by a large scale computation, earning us the label of "stunt cryptography" on the Security, Cryptography, Whatever podcast [85]. All of our attacks were disclosed to Threema and have since been patched.

In Chapter 3, we turn from messaging to key derivation, and to the Multi-Factor Key Derivation Function (MFKDF) framework of Nair and Song [220]. Password-based key derivation functions turn passwords, human-memorable strings, into cryptographic key material. Passwords have a long history of security shortcomings: they can be compromised, they are potentially low entropy, and they are often reused. Despite this, they remain ubiquitous: they don't require complex key synchronization across devices, since the users themselves carry passwords around in their memory. For end-to-end encrypted systems, like cloud storage, this is especially convenient: it means that users can access their data without access to cryptographic state stored in a particular device. MFKDF sets out to mitigate the shortcomings of passwords, by combining them with additional sources of entropy (the 'factors') before using them for deriving keys. MFKDF does not design purpose-made factors for this task, but tries to harvest entropy from existing and widely available primitives used for multi-factor authentication (MFA), such as one-time codes from authenticator apps (e.g., HOTP).

MFKDF-derived keys, Nair and Song propose, can then be used for cryptographic applications and for more flexible MFA. A crucial detail, though, is what it means to use a "one-time" code for deriving a key. After all, an input of the same password and the current one-time code must always produce the same key.

We find that one-time codes lose their one-timeness when used for MFKDF: by design, old codes are as security-sensitive as the current ones. 'Dynamic' authentication factors, normally such that an adversary cannot replay old values during the authentication process, are no better than static ones (like a never-changing 6-digit code) when used in MFKDF. This severely limits the usefulness of the proposed framework: an MFKDF-derived key is strictly worse than an equivalent multi-factor setup, when used for authentication. An adversary who learns that key gains irrevocable access to all authentication secrets, while a traditional MFA setup would require them to have access to the current one-time code.

We also find many problems with the MFKDF specification and reference implementation. These lead to practical attacks that weaken the security of derived keys, allowing us to reduce, weaken or outright bypass the contributions of many factors, and in some cases to even recover the entire key by compromising a single factor. Our attacks have resulted in a redesign of MFKDF, 'MFKDF2', which is still pending a comprehensive security proof.

Chapter 4 concludes our anthology of attacks with a study of the security of four password managers: Bitwarden, LastPass, Dashlane and 1Password. We already introduced some of the problems of passwords in our discussion of MFKDF—humans are not good at choosing passwords, and like to reuse them. Password managers try to solve the problem in an orthogonal fashion to MFKDF: instead of augmenting passwords, they just act as an (encrypted) database of passwords. This database (the *vault*) can be decrypted by remembering only a single password—the so-called 'master' password. This basic functionality—one may argue—is hard to get wrong. As long as a password manager uses a strong enough encryption algorithm, the confidentiality and the integrity of the content should follow. As it turns out, it is not that simple. The password managers we study are *cloud-based*, and therefore outsource storage of the vault to a server. They also provide a large amount of extra features, besides protecting the vault: sharing credentials, recovering lost master passwords, granular synchronization for vault updates, and backwards compatibility to previous versions of their applications. This provides us with a lot of attack surface. On one hand, the vault is outsourced to a server, which might be malicious or compromised. On the other, all of the extra features bolted on top of the vault encryption have not been previously studied formally. We thus find a plethora of attacks, many of which allow the recovery of passwords.

All of our attacks are in the threat model of a malicious server. The vendors promise 'zero-knowledge encryption', which is just, upon deeper inspection, a modern snake-oil term for 'end-to-end encryption'. We therefore believe this threat model to be the appropriate one: both because passwords are high-value secrets that deserve strong protection, and because a malicious server is the universally accepted adversary for end-to-end encryption.

We proposed mitigations, and we engaged in a months-long coordinated disclosure process with all of the affected vendors. Many of our attacks have already been mitigated. Others, however—particularly those exploiting the lack of authentication of public keys—require deeper, more far-reaching redesigns, and only a handful of the password manager companies have committed to addressing them.

1.2 PUBLICATIONS

This thesis is based on the following publications, presented here in extended form compared to the original manuscripts:

- Kenneth G. Paterson, **Matteo Scarlata** and Kien T. Truong. 'Three Lessons From Threema: Analysis of a Secure Messenger'. In: *USENIX Security 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 1289–1306. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/paterson
- **Matteo Scarlata**, Matilda Backendal and Miro Haller. 'MFKDF: Multiple Factors Knocked Down Flat'. In: *USENIX Security 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/scarlata
- **Matteo Scarlata**, Giovanni Torrisi, Matilda Backendal and Kenneth G. Paterson. 'Zero Knowledge (About) Encryption: A Comparative Security Analysis of Three Cloud-based Password Managers'. In: *35th USENIX Security Symposium (USENIX Security 2026)*. USENIX Association, 2026. URL: https://www.usenix.org/conference/usenixsecurity26

Over the course of his doctorate, the author also contributed to the following publications:

- Matilda Backendal, Mihir Bellare, Felix Günther and **Matteo Scarlata**. 'When Messages Are Keys: Is HMAC a Dual-PRF?'. In: *CRYPTO 2023, Part III*. ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. LNCS. Springer, Cham, 2023, pp. 661–693. DOI: 10.1007/978-3-031-38548-3_22
- Kien Tuong Truong, Simon-Philipp Merz, **Matteo Scarlata**, Felix Günther and Kenneth G. Paterson. 'Breaking and Fixing Content-Defined Chunking'. In: *ACM CCS 2025*. Ed. by Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie and Véronique Cortier. ACM Press, 2025, pp. 2294–2308. DOI: 10.1145/3719027.3744870
- Andris Suter-Dörig, **Matteo Scarlata** and Kenneth G. Paterson. 'Delegating Email Encryption to Gateways: Why Johnny Should Not'. In: *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. To appear. ACM, 2026

Chapter 2

Three Lessons From Threema: Analysis of a Secure Messenger

This chapter has been previously published as:

> Paterson, K. G., **Scarlata, M.**, Truong, K. T. "Three Lessons From Threema: Analysis of a Secure Messenger." *32nd USENIX Security Symposium (USENIX Security 2023), pp. 1289–1306. USENIX Association, August 2023.*
> DOI: https://doi.org/10.3929/ethz-b-000623004

For this chapter, I have made minor modifications from the original publication, including extended details that were present in the full version [230] or previously unpublished.

Author contributions
This chapter is based on joint work of all co-authors. KTT did the code analysis and the proof of concept code for the attacks, as well as an initial write-up, as part of his MSc thesis. KGP and I provided guidance during the analysis, and, together with KTT, ideated the attacks and authored the paper manuscript. My main contribution was in the extraction and the modelling of the cryptographic protocols in Threema: it was in their (informal) treatment as key exchange protocols that many of the attacks emerged.

Summary In this chapter, we provide an extensive cryptographic analysis of Threema, a Swiss-based encrypted messaging application with more than 10 million users and 7000 corporate customers. Our analysis took place at the end of 2022, and focuses on the October 2022 release of Threema (Android version 4.83). We present seven different attacks against the protocol in three different threat models. We discuss impact and remediations for our attacks, which have all been disclosed to Threema and patched. Finally, we draw wider lessons for developers of cryptographic protocols.

We integrate our discussion with details that emerged after our initial research, and comment on the state of Threema at the time of writing this thesis; a cryptographic analysis of the 2026 version of the application remains outside of the scope of this chapter.

2.1 INTRODUCTION

Chapter Overview:

2.1 Introduction 10
2.2 Secure Messaging . . 13
2.3 Threema Architecture and Protocols . 16
2.4 Attacks on Threema 25
2.5 Mitigations 36
2.6 Conclusions 39

Threema is a Swiss encrypted messaging application. It has more than 10 million users [134] and more than 7,000 on-premise customers [140]. It is among the top Android apps in Switzerland, Germany, Austria, Canada, and Australia in the "paid for" category [28]. It has undergone independent security audits [170, 157]. Along with Signal and Telegram, Threema was widely advertised as a more secure alternative to WhatsApp [263, 112, 269] in the wake of Facebook's acquisition of the company [112] and of the January 2021 change in WhatsApp Terms of Service [142, 285]. Threema is the messaging app of choice of the Swiss Government and of the Swiss Army [119], where its use is mandated for all official communications [273]. Threema is also used by German politicians, including the Chancellor at the time of writing, Olaf Scholz [248]. The company behind Threema makes strong claims for its security [139] and advertises its Swiss, non-US jurisdiction as a virtue.[1]

[1] Quoting from [134]: "Threema is 100% Swiss Made, hosts its own servers in Switzerland, and, unlike US services [...] it is fully GDPR-compliant...

In general, secure messaging protocols are designed to provide end-to-end encrypted communication between two (or more) parties. Modern designs like Signal achieve not only confidentiality, integrity and origin authentication of the exchanged messages, but also:

- **session key security**: compromise of a session key or ephemeral values in one session should not affect the security of keys in other, parallel sessions;
- **fine-grained perfect forward secrecy (PFS)**: compromise of session keys in an on-going session should not affect security of earlier keys and exchanged messages in that session and compromise of long-term keys should not undermine the security of completed sessions; and
- **post-compromise security (PCS)**: after a compromise of all key material, it should be possible to regain security after a protocol execution, provided the adversary is passive during that execution.

Such properties have been established for Signal for two-party messaging [80, 184, 81, 105].

At the same time, the landscape of messaging apps includes a wide variety of solutions, with different trade-offs with respect to centralization and federation, company jurisdiction, ease of use, openness of design and code, and security guarantees provided by the underlying cryptographic protocols. The absence of a standard for secure messaging means that many of the contenders feature bespoke cryptographic protocols with unclear security guarantees [20, 12, 13]. Indeed, these security guarantees are often much weaker than those provided by Signal, which can be seen as providing a "gold standard" for secure messaging.

2.1.1 Our Contributions

We provide the first in-depth security analysis of the cryptography used in Threema, focussing on its protocols for securing end-to-end (E2E)

and client-to-server (C2S) communications, for user registration, and for backing up users' private keys. We also examine the interactions between these protocols.

Our analysis uncovers seven different attacks against Threema, in three different threat models (as depicted in Figure 2.1). One of the attacks was discovered before us by Krebs, disclosed to Threema and patched in December 2021 [138]. We include it here for completeness. We produced proof-of-concept implementations for most of the attacks to verify that they work. We summarise the threat models and attacks next.

In the "external actor" threat model we consider attacks by outsiders with no special access to Threema servers or clients. Here, we find two attacks:

1. An adversary who is able to compromise a single ephemeral value used by a client in the C2S protocol is able to permanently impersonate that client to the server, and through this, access the metadata for all E2E-protected messages. This strongly violates the expected security properties of an authenticated key exchange.

2. An adversary is able to register the server's public key as a user public key and, by tricking the victim into sending a single carefully crafted message in the E2E protocol, can obtain a value that enables it to again permanently impersonate the victim to the server in the C2S protocol.

In the "compromised Threema" threat model we consider attacks by an adversary who has gained access to Threema servers. Such access should not enable breaking the end-to-end security guarantees. Here, we describe three attacks:

3. Due to peculiarities in nonce-handling in the E2E protocol, the adversary is able to arbitrarily and undetectably reorder and/or delete messages sent from any client to any other client.

4. Again, due to how nonces are handled in the E2E protocol, Threema clients must keep a local database of all nonces used in sending and receiving messages. This database is deleted when a user reinstalls the app or changes devices. After such an action, the adversary can replay old messages. Because of poor key separation, the adversary can also send back *to* user Alice any old messages originally sent *by* Alice.

5. For registration, a client proves possession of its private key by encrypting a server-chosen message to a server-chosen public key using the E2E encryption mechanism. This process is invisible to the user. This allows a malicious server to create "Kompromat": a potentially incriminating message that can be delivered at any later time to a target user. This is the attack that we rediscovered.

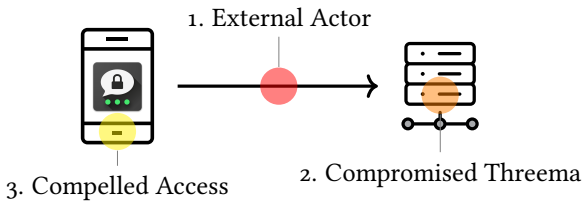


Figure 2.1: In the threat models we consider, the attacker can have access to network communication (1), the Threema servers (2) or the victim device itself (3).

In the "compelled access" threat model, we consider attacks by actors with direct access to a user's device, for example law enforcement officers or border guards. We present two attacks in this setting:

6. A trivial attack exploiting a Threema feature allowing users to easily export a backup of their long-term private key protected under a chosen password. Because of Threema's design, this key can be used to clone a Threema account and, through this, silently access all the user's future messages. In conjunction with a compromised Threema server, all old messages can be accessed too. This attack is only possible when the app and mobile device are both unlocked.

7. A more sophisticated attack exploiting one of Threema's backup mechanisms; this attack does not (by default) require an unlocked app (but does need an unlocked mobile device). It exploits Threema's use of "compress-then-encrypt" when creating the backup, in combination with the ability of the adversary to inject chosen strings into the backup file through Threema nicknames. By observing the size of the backup file over many operations with chosen nicknames, the adversary can recover the user's private key, with the same consequences as above.

Attacks 1 and 2 do not, by themselves, leak plaintext, but they target metadata that, under normal circumstances, only the server should be able to access. Furthermore, they break the PFS guarantees of Threema, allowing plaintext to be recovered if the long-term key of a user is ever exposed. Attacks 3, 4, and 5 show that the Threema server has the power to undetectably disrupt conversations, despite the E2E protocol (which, however, still guarantees message confidentiality). This requires users to put excessive trust in the server. Attacks 6 and 7 allow an attacker to obtain future plaintexts, as well as recover past plaintexts that are not protected by the C2S layer; however, they do require access to an end device. Our attacks are not directly caused by Threema's lack of PFS and PCS, but they showcase how such a deficiency can amplify the impact of smaller attacks. The attacks range in practicality. For example, attack 6 is feasible for an unskilled attacker that has access to the device for a very short time, while attack 1 is more of theoretical interest. See Table 2.1 for a summary of the attacks' prerequisites and impacts.

In totality, our attacks seriously undermine Threema's security claims. In particular, we show that Threema's approach to PFS (i.e. only providing it at the message transport layer) is insufficient even against a network attacker. All the attacks can be mitigated, but in some cases, a

major redesign is needed. We discuss appropriate mitigations and the remediations undertaken by Threema, after presenting the attacks.

We also draw three wider lessons for developers of secure protocols: using secure libraries for cryptographic primitives does not on its own lead to a secure protocol design; beware of cross-protocol interactions; and adopt a proactive rather than reactive approach to secure protocol design.

2.2 SECURE MESSAGING

The last 20 years have seen a rapid evolution of secure messaging: from virtually non-existent for the general public in the early 2000s [182] to the default means of communication for billions of people today [285].

The goal of secure messaging is that of establishing a secure communication channel between the communicating parties, but the nature of messaging sets it aside from other secure channel protocols. Sessions of a secure messaging protocol are characterized by:

- Low latency: the messages are expected to be received shortly after they are sent;
- Asynchronicity: parties can become unreachable, but their communicating partners should be able to encrypt and send messages to them for later (eventual) reception;
- Longevity: once established, one chat session can protract for years;
- Mobility of devices: a high volume of the instant messaging happens on smartphones, which are highly susceptible to being stolen or temporarily compromised.

These technical constraints make for engineering and cryptographic challenges: the authors of "OTR" [65], one of the earliest secure instant messaging protocols in the literature, note that email encryption systems such as PGP fall short of the security expectations of users in a messaging setting.

State-of-the-art secure messaging protocols, such as Signal, have gravitated towards demanding fine-grained forward secrecy and post-compromise security for their key exchange component, alongside standard confidentiality and integrity notions for stateful secure channels.

- Forward Secrecy: Following a compromise of the parties' secret keys, messages sent in the past should stay secure. Given the longevity of the sessions, this is of capital importance: a later compromise of a device should not allow an adversary who recorded the encrypted protocol messages to decrypt them.
- Post-Compromise Security: Following a compromise of the parties' secret keys, and a period of time during which the adversary does not tamper with protocol messages, messages sent in the future

should stay secure. The mobility of the devices gives adversaries many windows of opportunity for compromising a party's key: this is the case of device searches at borders, and with abusers in domestic settings.

2.2.1 *Related Work*

An independent early analysis of the Threema client-to-server protocol, dating back to the closed-source Threema version 1.3, was carried out by Ahrens [10], and was the basis for an open-source implementation by Berger [46]. Ahrens documents the protocol flow, the cryptographic primitives and the message format, but does not discuss the security properties of Threema. We analyse the October 2022 version (version 4.831, that was current when this research was carried out) of the Threema Android application, and base our analysis of the protocol on the 2020 open-source release of the Threema app [137] and the Threema security white paper [135].

Rösler et al. [255] considered the end-to-end security of group chats in various secure instant messaging services, including Threema. They observed the lack of forward secrecy and post-compromise security for Threema group chats, and described two attacks: a replay attack on group messages, and leakage of group membership information to non-members. Both attacks have since been patched by Threema.

Cremers et al. [87] investigated the post-compromise security of Threema (and other apps) in a black-box setting, by cloning a victim device and observing whether the clone retains access to exchanged messages. The authors noted that Threema prevents the clone from accessing messages exchanged while the clone is offline. While this result holds in a black-box setting, the weaknesses we uncover in Threema's client-to-server protocol enable even an adversary with limited technical proficiency to intercept messages if given access to the full cloned state of the victim.

Threema itself commissioned two security audits [170, 157]. Both audits focused on the Android and iOS apps for Threema, and did not touch on its cryptographic security beyond confirming that the implementation of the protocol matches the description provided in the Threema white paper [135].

Soatok [271] highlighted Threema's lack of forward secrecy as a deficiency of the app. They also note that Threema groups lack transcript consistency: that is, different participants in a group can have different views of what messages have been exchanged. The lack of transcript consistency has not yet been resolved by Threema at the time of writing this thesis and, as [271] argues, can be combined with the lack of robustness of the XSalsa20-Poly1305 AEAD cipher to obtain "Invisible Salamander"-style attacks [103, 11]: an attacker can send attachments in group chats that decrypt to different plaintexts for different chat participants. Recent work by Gegenhuber et al. [129] shows that WhatsApp, Signal and iMessage (as well as Threema) all fail to provide transcript consistency in a group setting.

Follow-ups After disclosing our findings to Threema, we learned that they had been working on adding forward security to their end-to-end encryption: this would be achieved by nesting the existing, non-forward secure E2E layer with a forward secure E2E protocol dubbed *Ibex*. Our attacks precipitated the release of Ibex, and also prompted Threema to commission a formal security analysis of their new protocol. Schröder, Rösler and Gerhart [131] showed that Ibex does achieve forward secrecy. Notably, they noted that Ibex messages being wrapped in the legacy E2E protocol results in a circular security assumption which is an obstacle to full provable security [131, §9]. At the time of writing this thesis, forward secrecy is still not available when using Threema on multiple devices [136, p. 17]. This means that, for multi-device users, a malicious yet passive server can decrypt all past and future communication after the compromise of long-term keys. This is in stark contrast with, for instance, Signal, where the compromise of long-term keys does not affect past communication, and still requires an actively malicious server in order to compromise future communication [80]. Signal has offered these security guarantees since its 2014 initial release [204], and has supported linking secondary devices since 2015 [121].

2.2.2 *Ethical Considerations*

We only used the description of Threema in [135] and the client-side source code to conduct our analysis. We did not attempt any reverse-engineering of the server behaviour other than what could be observed passively through normal client interaction. We performed experiments only with test accounts under our full control; no other user accounts were targeted. We reduced as much as possible the number of messages sent to Threema servers, and we avoided sending malformed messages to servers so as to avoid triggering DoS attacks.

We did purchase a "Threema Gateway" account (with Threema ID `*LYTAAAS`) and registered a public key under that account identical to the Threema server public key (obviously, without knowing the corresponding private key). This step was necessary in order to validate one of our attacks (Section 2.4.1.2). It did not violate any terms or conditions of the service, so far as we could determine.

We disclosed the results of this work to Threema on 03.10.2022, proposing a 90-day disclosure period. They acknowledged receipt also on 03.10.2022. We agreed on the details of the coordinated disclosure: a first batch of mitigations would be released in Q4 of 2022, while further mitigations would be released in Q1 of 2023, along with a post on Threema's blog. The first batch of mitigations was included in Threema 5.0 for both Android and iOS[2] on 29.11.2022. To further remediate the issues we reported, Threema included their custom protocol, Ibex, in the same release. The new protocol aims to provide forward secrecy at the E2E layer.[3] Threema also updated their whitepaper [135] on 15.12.2022 to add discussion of the forward security feature. We have not audited this new protocol. We agreed to publicly disclose our findings on 09.01.2023.

[2] https://threema.ch/en/whats-new

[3] https://threema.ch/en/blog/posts/ibex

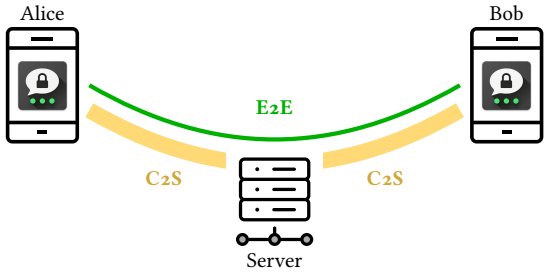


Figure 2.2: E2E and C2S protocol composition. Clients establish a secure channel with the server using the C2S protocol (in yellow) to send and receive E2EE messages from other users, which are relayed via the server (in green). Note how the server sees E2E encrypted messages directly.

The patches released by Threema in response to our disclosure addressed all of our attacks.

In one of our attacks, we leveraged a vulnerability in a library, Zip4j, used by Threema to create backup zip files. We disclosed our findings to the author of Zip4j on 08.10.2022, proposing a 60-day disclosure period. A CVE advisory for the bug (CVE-2023-22899) was published on 10.01.2023 [215]. A fix was released in version 2.11.3 on 26.01.2023.

2.2.3 *Chapter Structure*

In Section 2.3 we introduce the Threema architecture and provide a description of Threema's core cryptographic protocols and subsystems. Section 2.4 presents our attacks, grouped by threat model. We propose mitigations in Section 2.5. We discuss three wider lessons for secure protocol designers in Section 2.6.

2.3 THREEMA ARCHITECTURE AND PROTOCOLS

The design of Threema can be studied as the ensemble of several interoperable cryptographic protocols, all sharing the same user state and cryptographic primitives. The functional core of Threema consists of an end-to-end (E2E) protocol, which employs the long-term cryptographic secrets of the users to encrypt messages, and a client-to-server (C2S) protocol, a secure channel protocol that protects these messages in flight from clients to the Threema servers. Because of the composition of these two protocols, each message is wrapped in two layers of encryption: an inner end-to-end layer, and an outer client-to-server layer. We depict the composition in Fig. 2.2. Also relevant to our analysis and attacks are the registration protocol, data backup and contact discovery system.

We first describe the cryptographic primitives used by Threema and establish our notation. Then we analyse the registration protocol, the E2E protocol, and the C2S protocol. Finally, we briefly cover the backup mechanisms and the contact discovery mechanism.

2.3.1 *Cryptographic Primitives and Notation*

The protocols used by Threema rely on standard cryptographic primitives. The main tool used is the `crypto_box` abstraction provided by the Networking and Cryptography Library (NaCl) [47] which consists

of a Curve25519 Diffie-Hellman key agreement, followed by encryption with an Authenticated Encryption with Additional Data (AEAD) algorithm. The AEAD algorithm used by NaCl is the nonce-based XSalsa20-Poly1305, which is a composition of the XSalsa20 stream cipher and the Poly1305 polynomial hash. Similarly to other nonce-based AEADs, in particular the related ChaCha20-Poly1305, XSalsa20-Poly1305 has the requirement not to use the same nonce twice, since such a reuse could lead to loss of confidentiality, as well as the ability for an attacker to forge arbitrary authentication tags. The loss of confidentiality is due to the fact that using the same nonce with the same key leads to the same keystream being generated. This keystream is used both to encrypt the message and to generate the keys used by Poly1305; such a repetition leads to the XOR of the ciphertexts revealing the XOR of the plaintexts, as well as breaking the security guarantees provided by Poly1305. In the specific case of XSalsa20, the size of the nonce is 24 bytes, which allows users to generate random nonces with negligible probability of collision, rather than using a counter as would be required when using Salsa20. Well-known attacks against polynomial MACs [153] exploit nonce reuse to recover the polynomial MAC key, enabling forgeries.

As a convention, long-term keys will be indicated with the same letter as the symbol for the user, with the public key in uppercase and the private key in lowercase (e.g. user $\mathcal{A}$ has private key a and public key A). We reserve (x, X), (y, Y), and (z, Z) for values used as ephemeral keys. Let $\mathsf{X25519}(\cdot)$ denote the NaCl function that takes a private key and a public key, and outputs a shared secret byte string. In NaCl, this string results from applying the hsalsa20 key derivation function to the byte representation of the shared elliptic curve point and a fixed input.[4] Internally, the X25519 algorithm operates on the Curve25519 elliptic curve: the algorithm uses 32-byte scalars as private keys and computes the scalar multiplication of the chosen base point of the curve g with the private key to obtain a 32-byte public key. We will use multiplicative notation for such operations: e.g. secret key a has a corresponding public key $A = g^a$ for some fixed generator g. We denote by $(x, X) \xleftarrow{\$} \mathsf{KeyGen}()$ the key generation procedure of Curve25519, which outputs a private key x and a public key $X = g^x$. Let K be an AEAD key; we denote by $\mathsf{E}_K(m; n)$ the encryption of message m under key K using nonce n, and by $\mathsf{D}_K(c; n)$ the corresponding decryption operation. If the decryption fails, we assume that the algorithm returns a special value $\perp$. This may happen, for example, if the ciphertext has been tampered with, or if the wrong key was used to decrypt.

[4] Note that hsalsa20 is not a standalone KDF, but it is used as a PRF in the XSalsa20 implementation.

In general, nonces can be randomly sampled, but it can be convenient to use a counter to generate them if the counter value can be kept in persistent storage: if the counter monotonically increases, then no nonce is used twice, unless the counter wraps around. In Threema's C2S protocol, counters are initialized with a 16-byte value called a *cookie*: the cookie is used to set the most significant 16 bytes of the generated nonce, while the lower 8 bytes are set to zero and the counter value is incremented by 1 before returning a new nonce value. We denote by

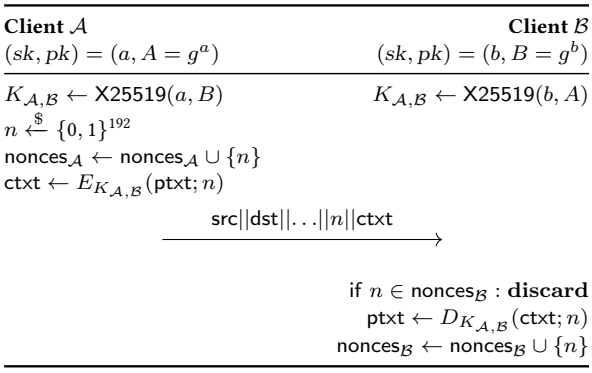


Figure 2.3: User $\mathcal{A}$ sending ptxt to user $\mathcal{B}$ with the E2E protocol. The set $\mathsf{nonces}_{\mathcal{U}}$ represents the nonce database of user $\mathcal{U}$.

$\mathsf{ctr} \leftarrow \mathsf{Ctr.Init}(C)$ the function that returns a counter object initialized with cookie C and $n \leftarrow \mathsf{Ctr.Next}()$ as the counter returning the next counter value n. Threema's E2E protocol simply samples random nonces.

We denote by $k \leftarrow \mathsf{KDF}(K, \sigma, \tau)$ a Key Derivation Function taking some key material K, a salt σ, and a label τ as input, and producing key k as output. Threema uses a KDF based on the BLAKE2b hash function [258].

2.3.2 *Registration Protocol*

In order to create a new account and register it with the Threema server, each user generates a Curve25519 key pair (a, A). The entropy for the generation of the key pair is taken from system randomness, as well as random user input which is processed and mixed with the former to obtain the 32-byte private key a. The client later runs a registration protocol with the server to prove knowledge of a. This is done via a challenge-response exchange run over a TLS-protected connection: the client sends A to the server. The server replies with an ephemeral public key X and a message m. The client computes $K = \mathsf{X25519}(a, X)$ and AEAD-encrypts m using K and a fixed nonce ("`createIdentity response.`") to create a response. If this response decrypts correctly and yields m as plaintext, the server accepts the registration and then issues a new Threema ID, storing it along with the public key of the user in the Threema database.

2.3.3 *Threema End-to-End Protocol*

The Threema E2E protocol, depicted in Figure 2.3, is concerned with guaranteeing end-to-end security of messages. Every user $\mathcal{U}_i$ of the protocol has an alphanumeric 8-character identity $\mathrm{ID}_{\mathcal{U}_i}$ (the *Threema ID*, or simply *ID*), a private key u_i, and a corresponding public key $U_i = g^{u_i}$. Messages sent from user $\mathcal{U}_i$ to user $\mathcal{U}_j$ are encrypted under $K_{\mathcal{U}_i,\mathcal{U}_j} = \mathsf{X25519}(u_i, U_j) = \mathsf{X25519}(u_j, U_i)$, a static Diffie-Hellman key. Note that the same key is derived by user $\mathcal{U}_i$ for $\mathcal{U}_j$ as the one from user $\mathcal{U}_j$ for $\mathcal{U}_i$: the key $K_{\mathcal{U}_i,\mathcal{U}_j}$ is bidirectional.

A message m is first serialized into a byte string, and prefixed with a byte that indicates the message type. The specific serialization method

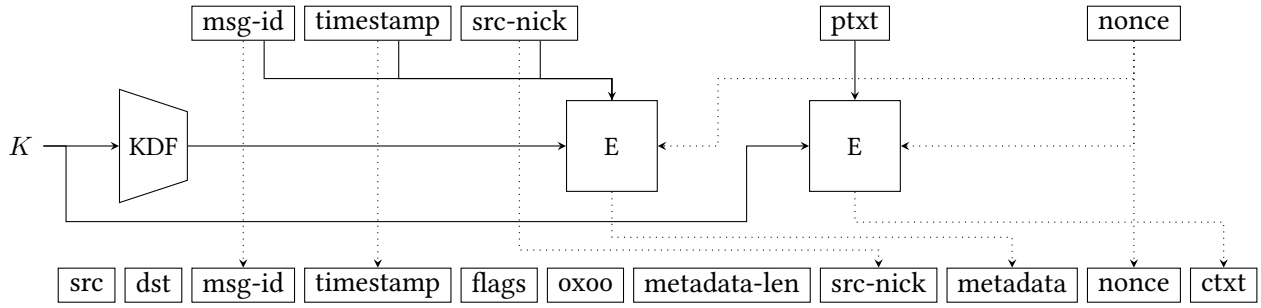


Figure 2.4: Structure of an E2E encrypted message. The input to the KDF is the key material K. The salt and label are implicitly input into the KDF, taking values "mm" and "3ma-csp", respectively.

depends on the message itself. For instance, a *text message* would be directly encoded as a byte string representing the UTF-8 string, and prefixed with `0x01`. A random amount of PKCS7 padding, between 1 and 254 bytes, is appended to the message. The padding is used to make the message at least 32 bytes long and to hide the true length of the message. We will refer to the serialized message, along with the leading byte and the padding as the *plaintext* (`ptxt` in Figure 2.4). The resulting plaintext `ptxt` is then encrypted under K using the AEAD algorithm with a random nonce, `nonce`, to obtain a ciphertext `ctxt`.

To create the final message packet, the following information is prepended to the ciphertext: the Threema ID of the sender `src`, and of the receiver `dst`, a random 8-byte message ID `msg-id`, the timestamp at which the message was sent `timestamp`, the nickname of the sender `src-nick`, the nonce `nonce` and an *optional* `metadata` value. To construct this `metadata`, the app takes the message ID, the nickname of the sender and the timestamp, concatenates them, and encrypts the resulting plaintext under $\mathsf{KDF}(K, \text{"mm"}, \text{"3ma-csp"})$, using the same nonce value as was used to encrypt `ptxt`. Since the presence of the metadata box is optional and it is not cryptographically bound to the ciphertext, it can always be stripped off and its length set to `0` by an adversary. In Fig. 2.4 we depict the structure of an E2E-encrypted packet in Threema.

When sending an E2E message, the associated random nonce is stored in the device's local storage in order to avoid reusing it. If a generated nonce matches a nonce in the local storage, up to five attempts are made to sample a new nonce. When receiving a message, the app checks if the corresponding nonce has already been seen: if so, it rejects the message. If the nonce is otherwise fresh, the app tries to decrypt the message, verifying the MAC tag as part of the process. If successful, the app stores (a hashed representation of) the nonce. This storage of nonces is aimed at preventing trivial replay and reflection attacks (which would otherwise be possible since the key $K_{\mathcal{U}_i,\mathcal{U}_j}$ is bidirectional) and hedges against the possibility of a faulty randomness source.[5]

Remarkably, the E2E protocol does not provide any forward secrecy, since it only uses the long-term keys of the users, with no ephemeral values being exchanged. This means that a malicious server could take

[5] Note that this nonce database is not *required* to defend against replay and reflection attacks—but this appears to be a somewhat common misunderstanding of NaCl [27]. A more principled design could, for instance, rely on counters to hedge against replay attacks, and use different keys for each communication direction to prevent reflection attacks. This would remove the cost of storing the nonces, reduce the complexity of managing a nonce database, and, as we will see, prevent attacks in the case the nonce database is lost.

advantage of its position to execute a "store now, decrypt later" attack. This has supposedly been fixed with the introduction of the Ibex protocol.

The simplicity of the key exchange part of the protocol, relying on a static Diffie-Hellman key exchange, makes it superfluous to study its security in traditional authenticated key exchange models such as eCK [188], ACCE [171] or MSKE [120]. Indeed, the protocol is trivially broken if the adversary has access to a party's secret key material; different sessions of the same two parties always derive the same secrets and are not independent.

2.3.4 *Threema Client-to-Server Protocol*

The Threema C2S protocol is concerned with establishing a secure channel between a client and the server to protect messages in flight, shielding metadata and the non-forward secret E2E protocol messages from a network adversary. Note that this protocol is only used for messaging: other functionalities such as registration, backup, and contact matching are implemented over TLS.

[6] We learned from the Threema team that the C2S protocol is actually inspired by Bernstein's CurveCP [49], announced in 2010 at 27C3 [50]. They also told us that the decision to not use TLS directly as a transport stemmed from the abundance of TLS attacks at the time the C2S protocol was designed (2012).

Threema's C2S protocol is a novel[6] secure channel protocol, broadly comparable in function and construction to the TLS protocol. It assumes a reliable transport, and can be decomposed as a *handshake subprotocol*, in which the server and the client interact to establish a session key, and a *transport subprotocol*, where the server and the client use the established session key to exchange messages.

The C2S protocol requires that each user connects to the server from only one device at any given moment. If two devices of the same user try to connect to the server at the same time, the older connection is dropped in favour of the newer one. Before terminating the connection, an error message is sent to the device, informing it that another device has connected with the same Threema ID.

2.3.4.1 *Handshake Subprotocol*

The handshake subprotocol runs between the server $\mathcal{S}$ (with long-term key pair $(s, S = g^s)$) and a client $\mathcal{U}$ (with long-term key pair $(u, U = g^u)$). Fig. 2.5 gives a pictorial representation of the protocol. We refer to the figure for the naming of the messages below. We can assume that the server and the client already know each other's public keys: the client knows the server's key as it is pinned in the app and the server receives the client's key at registration time.

The client first generates an ephemeral key pair $(x, X = g^x)$ and a 16-byte value c called the *client cookie*, and sends (X, c) to the server (`client-hello`). After receiving the client's ephemeral public key and the client cookie, the server generates its own ephemeral key pair $(y, Y = g^y)$ and a 16-byte value called the *server cookie*. It then computes the first shared symmetric key of the exchange $K_1 = \mathsf{X25519}(s, X)$, mixing the long-term key of the server with the ephemeral key of the client, and uses it to encrypt its ephemeral public key concatenated with the

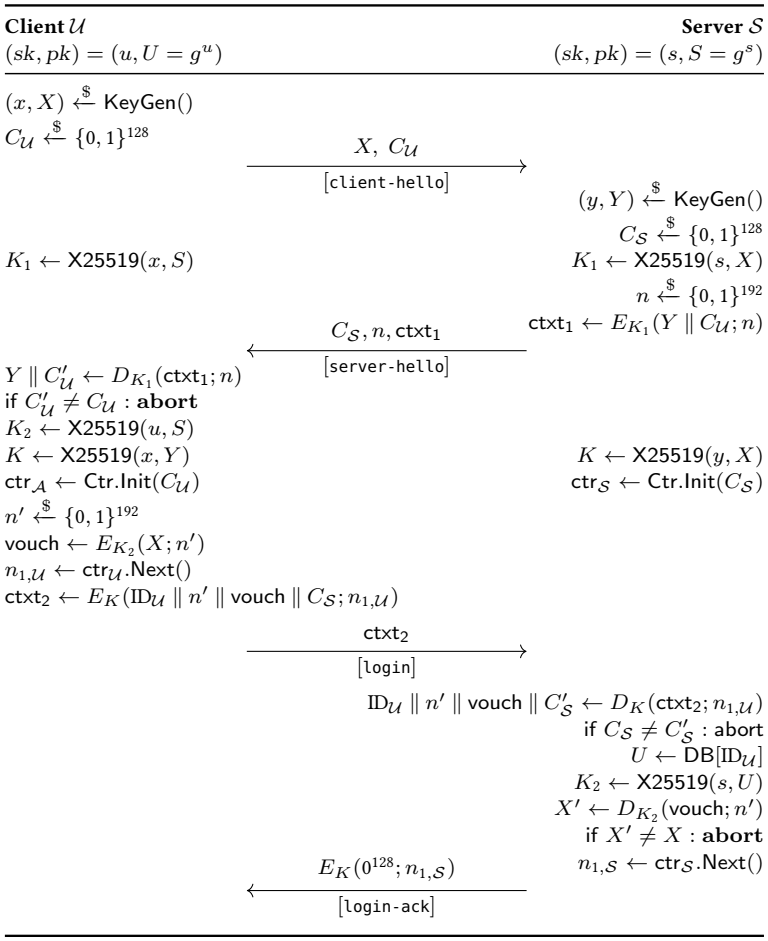


Figure 2.5: The Threema C2S Protocol (Handshake Subprotocol). Nonces are derived from the counters; they are implicitly re-derived on the receiving end and are not transmitted. The table DB is an abstraction for the database that maps Threema IDs to public keys. Message names are taken from the Android code and depicted in square brackets.

client cookie, using a random nonce n. The server sends the resulting ciphertext, along with its own cookie (`server-hello`).

The client and server cookies prevent replay attacks and are used to initialize client and server *counters*. For each counter, the most significant 16 bytes of the counter are set to the respective cookie (denoted $\mathsf{Ctr.Init}(C)$), while the lower 8 bytes function as a monotonically increasing value. Subsequent AEAD nonces for the exchanged messages are drawn from these counters (denoted $\mathsf{Ctr.Next}()$).

When the client receives the message, it recomputes K_1 and checks if the ciphertext decrypts correctly and if the client cookie in the plaintext corresponds to the one that the client previously generated. If either check fails, it will abort the protocol. The client and the server derive two additional keys: K, obtained by combining the two ephemeral key pairs (x, X) and (y, Y), and K_2, obtained by combining the two long-term key pairs (u, U) and (s, S). The key K is used in the transport subprotocol as the *session key*. The long-term symmetric key K_2 is used by the client to create the so-called *vouch box* $\mathsf{vouch} \leftarrow E_{K_2}(X; n')$ using a random nonce n'. The vouch box is used by the server to authenticate the client, and to bind the client's ephemeral key X to the client's identity.

The vouch box is concatenated with $\mathrm{ID}_{\mathcal{U}}$ and the server cookie $C_{\mathcal{S}}$, and encrypted using the session key K (`login`). The server decrypts

the message with K and tries to decrypt the vouch box with K_2. If the decryption fails or the value contained in the vouch box does not correspond to the client's ephemeral key, the server will abort the protocol. Otherwise, the server sends a final confirmation message composed of 16 zero bytes, encrypted with the session key K (`login-ack`).

Taken as a standalone protocol, the C2S handshake could be formally analysed using, for example, the extended Canetti-Krawczyk security model [188] or the MSKE security model of [120] that was used in [80, 81] to analyse Signal. We will (informally) use these models when relating the attacks in Section 2.4 to the formal security properties they violate.

2.3.4.2 *Transport Subprotocol*

In the transport subprotocol, the client and the server exchange messages, encrypted using the session key K. The client and server nonces are derived from the same (respective) counters inherited from the handshake subprotocol, ensuring that handshake messages cannot be used as transport messages (and *vice versa*). If a decryption fails, the entire connection is dropped. This prevents message reordering and message deletion by a network adversary, since either attack would result in a party decrypting with a nonce different from the correct one, and then the AEAD integrity guarantees would ensure that an error is generated on decryption.

When the connection is dropped, the client attempts to reconnect, with up to five reconnection attempts being made. If these all fail, the app will display an error to the user and will require a restart to be able to connect to the server. If two devices are actively trying to connect to the server, they will keep interfering with each other until one of them relinquishes the connection.[7] This is relevant for any attacker that wants to attempt an undetectable impersonation attack, since using all five attempts will alert the user that another device is trying to connect to the server with the same Threema ID.

[7] Note that at the time of writing this thesis, limited multi-device support appears to be available [277]; however, this was introduced after our analysis was conducted and is not covered in this work.

While the connection is alive, the server will send messages that are meant for the user encrypted under the session key K. The client replies with an (encrypted) ACK in response to each message. If the client does not send an ACK, the server will try to send the message again when the client later reconnects.

2.3.5 *Threema Safe*

Threema allows a user to activate, at any point in time, the Threema Safe feature. This allows the user to backup their long-term private key, their nickname, the list of their contacts and additional account information on the Threema servers. Note that the backup does not store any message history.

When the Threema Safe is first set up, the user chooses a password for their backups. A minimum length of 8 characters is enforced, and the password is checked against an offline list of weak passwords. The app derives two 32-byte strings, the Backup ID BID and the Backup Key BK,

using scrypt with the password as input and the Threema ID of the user as salt. For every backup, the app creates a JSON payload containing all the data to be backed up and serializes it into a byte string, which is then compressed using gzip. Finally, the app encrypts this compressed JSON string with XSalsa20-Poly1305 under BK with a random nonce. The resulting ciphertext is sent to the server over a TLS connection.

The server will store ciphertexts indexed by BID, supposedly without any other information related to the user. In the Threema white paper [135], the developers claim that this process is designed to make it impossible for the Threema Safe server to distinguish which backup comes from whom by looking at the uploaded data. We note, however, that when a backup is created the app is likely to have an active C2S protocol session, which reveals the Threema ID of the user. By linking the two connections coming from the same IP, the server can learn the identity associated with a given backup.

A backup is scheduled to happen every day or, in case the previous backup failed to be uploaded, as soon as the app is restarted. This happens even if access to the app is protected by a PIN or biometric login.

2.3.6 Other Backup Methods

Threema provides two additional methods to save personal data: the first one is called a *Threema ID export* and the second one is called a *data backup*.

The Threema ID export requires that the user provide a password, which is used to derive a 32-byte symmetric key by using the PBKDF2 algorithm with a random 8-byte salt. This key is used to encrypt the ID of the user, as well as their long-term private key using XSalsa20 with a zero nonce (but with no integrity protection). The result is base32-encoded and given to the user in order to be saved outside of the app.

A data backup consists of an encrypted zip containing various files. The `identity` file stores the Threema ID and the long-term private key of the user; the `contact` file stores Threema IDs, nicknames, and public keys of contacts in CSV form, and for each chat, a file named with the Threema ID of the correspondent stores the chat history. The zip is encrypted following the WinZip AE-2 specification [82]. This essentially means that the user-provided password is used to derive a master key. For each file, a pair consisting of an encryption key and an authentication key are derived from the master key. A different salt value is used for each key derivation. Then, the file is compressed, encrypted with AES-CTR and authenticated using HMAC-SHA-1.

We note that, because WinZip does not hide the file names, this encrypted zip leaks information about with whom the user has been communicating. Due to the length of each file being known, an attacker may also infer the length of the conversation from the encrypted zip.

Table 2.1: Summary of the attacks, listing the prerequisites and their impacts. The last column focuses on the ability of an attacker to obtain plaintext after executing the attack.

	Network Attacker	C2S Eph. Key Leak	User Action	Unlocked Phone	No UI Lock	No Passphrase	Server Compromise	Leak in-metadata	Leak out-metadata	Break of PFS	C2S MitM	Permanent C2S Impersonation	Conversation Integrity Break	Long-term key leak	Recovers Plaintext
C2S Key Compromise	✓	✓	✗	✗	✗	✗	✗	✓	✓[a]	✓	✓[a]	✓	✓	✗	✗
Vouch Box Forgery	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✗
Reordering/Deletion	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	–
Replay/Reflection	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	–
Kompromat	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	–
Threema ID Export	✗	✗	✗	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓
Backup Side-Channel	✓	✗	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓
	Prerequisites							Impact							

[a] As long as the client keeps using the revealed ephemeral key (i.e. up to one week)

2.3.7 *Contact Discovery*

If the user gives permission to do so, Threema can run a protocol to check which of the people in the user's phone contact list are using Threema. To do so, it must compare the user's contact list with the list of all registered Threema users, while at the same time preserving both the privacy of the user and the confidentiality of the list of registered Threema users.

Threema does this by hashing all the contacts of the user and sending the hashes to the Threema server over TLS. The server will reply with the hashes that correspond to users that have previously registered with Threema. The hash algorithm is implemented using HMAC-SHA-256 with a fixed key. This is done to "ensure that hashes generated by Threema are unique and do not match those of any other app" [135]. This is similar to the process used by apps such as Whatsapp, Telegram and Signal [152]. Such protocols tend to raise privacy concerns [152] due to the information leaked to the server, including information about people who might not even be aware of the app's existence. Hashing the information is insufficient, since, for example, phone numbers carry little entropy and can be brute-forced. This allows the service provider to build a partial social graph for an entire population. Signal tries to provide stronger guarantees by implementing their contact discovery protocol in an Intel SGX enclave [203], minimising the information leak due to memory access patterns, and using techniques such as Oblivious RAM to minimise the information leaked to the server [83].

To prevent abuse of the system, anyone who wants to run the contact discovery protocol must first obtain a *match token* by authenticating to the Threema server. To obtain one, the client must pass a challenge-response protocol which is essentially the same as the one done for the registration process, except for the fact that the nonce is now randomly chosen by the client, rather than being fixed.

2.4 ATTACKS ON THREEMA

We consider three threat models: first, an external actor who can monitor communications between a device running the app and the Threema servers; second, an actor that has compromised the Threema servers; third, an actor that can take control of the device for a short period of time.

The first of these models corresponds to the standard adversary for secure network protocols, often called the Dolev-Yao adversary. We refer to it as the *external actor* threat model.

Concerning the second model, any reasonable security analysis must consider the possibility that the messaging server is malicious: national security agencies and hacking groups are tasked with penetrating servers like those run by Threema in order to gain persistent access to concentrated amounts of sensitive data. It is then reasonable to ask what security guarantees remain for users in the event of such a breach. This is especially so for a system that claims to provide end-to-end security. We refer to this model as the *compromised Threema* threat model.[8]

We call the third model the *compelled access* threat model. It is relevant, for example, in the case of border searches of electronic devices, when protesters are detained by police forces and searched for incriminating evidence, or in the setting of intimate partner violence [122]. We may subdivide this setting into two sub-cases: one where the app is locked, and the other where it is unlocked (we will always assume the device itself is unlocked).

We next introduce our attacks, grouped by threat model. Table 2.1 summarises our findings.

[8] Within this threat model, we could further distinguish between a *fully malicious* server, which actively deviates from the expected behaviour to harm users, a *persistent passive adversary*, which can observe all the data transiting through the server, and a *snapshot adversary*, which only obtains snapshots of said data. All of our attacks in Section 2.4.2 assume the fully malicious server; Threema lacks forward secrecy against adversaries that are merely passive; even a snapshot attacker gains non-trivial leakage, e.g., users' social graphs from the Threema IDs in the backups, as mentioned in Section 2.3.6.

2.4.1 *External Actor Threat Model*

Our first two attacks allow the attacker to impersonate a victim in the C2S protocol. The core of both attacks lies in the structure of the *vouch box* 'vouch. This vouch box is used in the protocol to authenticate clients but contains only client-chosen freshness. By either replaying (Attack 1) or forging (Attack 2) a user's vouch box, the attacker can authenticate itself as that user to the server.[9] These impersonation attacks are *permanent*, despite the fact that the attacker never learns the value of the victim's long-term key.

Our attacks show that the weak notion of forward secrecy achieved by Threema is insufficient: by using either of our attacks, the adversary gains access to E2E-encrypted messages, which use long-term keys only. The reuse of ephemeral keys which we describe in the first attack further weakens the guarantees given by forward secrecy, since an attacker gains access to messages up to a week in the past. Comparing this design to Signal's, we note that in the latter each E2E message is encrypted with a different key, providing a much stronger, fine-grained notion of forward secrecy.

[9] While vouch does depend on a nonce n', we have verified that the server does not detect replays of this value. By design, the server also accepts reuse of the same client ephemeral key. The server will then accept any replayed vouch box as valid.

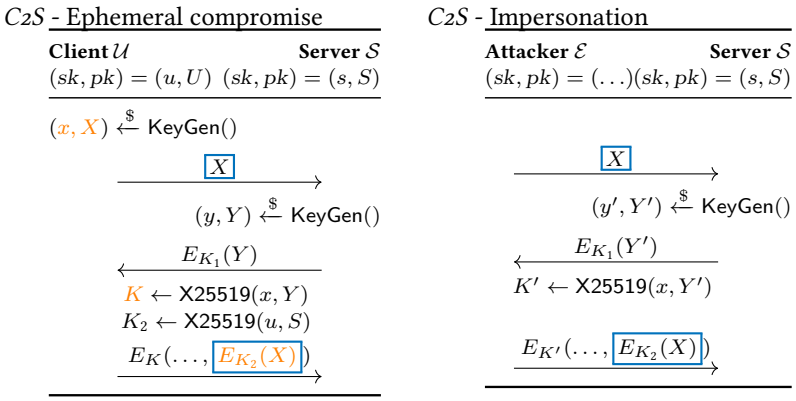


Figure 2.6: Attack 1: two C2S sessions. The attacker compromises an honest session by revealing the ephemeral private key x, from which they can compute the session key K and obtain the vouch box $E_{K_2}(X)$ (in orange, Left side). The values are replayed by the attacker in a new session (blue boxes, Right side). Note the lack of *session independence*.

2.4.1.1 *Attack 1 (C2S Ephemeral Key Compromise)*

Assumptions: We assume that the attacker has discovered one of the ephemeral keys x used by the client in a C2S handshake session. This capability is commonly referred to in formal security models as *ephemeral key reveal*, and it can arise in practice from a randomness failure or via a side-channel attack. We further assume that the attacker has passively recorded the corresponding C2S session.

Execution: We now show that, with the knowledge of x and the C2S session transcript, the attacker can create new sessions with the server in which they impersonate the victim and can tamper with the flow of messages.

As highlighted in Figure 2.6, the attacker has all the material necessary to derive K_1 and decrypt the `login` message from the recorded C2S protocol run, from which it can recover the value of vouch. Now the attacker initiates a new run of the C2S protocol, playing the role of the victim user. In this run it replays the ephemeral public value X. A new session key K' known to the attacker is generated (the server should send a fresh Y' in its `server-hello` but the attacker still knows x). The attacker can then reuse vouch (without learning K_2) when constructing its response in the `login` message, which makes the handshake complete successfully. We have verified that the server does not detect such replays.

Impact: Since vouch can be reused repeatedly, this attack (involving the loss of just a single ephemeral value) allows the attacker to permanently impersonate the victim user to the server, enabling the attacker access to all E2E-encrypted messages meant for the victim. Despite the E2E layer of encryption, this still reveals the metadata information in all communications sent to the victim, allowing the attacker to learn with whom and when the victim has been communicating.

Most importantly, this attack leads to a break of the forward secrecy property that the C2S protocol is intended to provide. Indeed, the E2E-encrypted messages that the attacker now sees are encrypted only using long-term keys. Full decryption of those messages becomes possible if

the victim's long-term key is later revealed (for example through attacks 6 or 7).

In addition, the attacker gains the opportunity to selectively drop messages from the conversation by deciding which messages to ACK to the server. If a message is ACK-ed by the attacker, it will not be received by the victim when they next connect to the server, virtually deleting that message from the conversation on the victim's end.

We stress that, as long as the attacker and the victim do not connect to the server at the same time, this attack is not detectable by the victim. If they were to connect at the same time, the attacker could observe error messages from the server to learn this. At this point, the attacker can relinquish the connection in order to avoid detection.

This attack shows that, for the C2S protocol, ephemeral keys are as important to protect as long-term keys. This is a very strong assumption to have to make in order to guarantee security for a key exchange protocol.

Off-The-Record (OTR) [66], a 2004 secure messaging protocol, suffered a similar vulnerability in its initial version [239]. The authors of [239] highlight the consequent lack of the *session independence* property in OTR: "the exposure of ephemeral session-specific secrets should have no bearing on the security of other sessions". Threema's C2S protocol suffers from the same issue.

Non-ephemeral Ephemerals: An additional problem with the C2S protocol lies in how the ephemeral keys are handled client-side: if the app is never restarted, then the same ephemeral key is used by the client for up to seven days before generating a new one. Leaking such a reused ephemeral key then allows an attacker to also impersonate the server to the user by forging a `server-hello` using the server's public key and the compromised ephemeral key (this is known as an *ephemeral Key Compromise Impersonation* attack in the literature). In turn, this implies that the attacker has the power to also read the outbound E2E-encrypted messages of the victim that are meant for the real Threema server. This reveals additional metadata, allows the attacker to drop arbitrary messages and allows a network adversary to execute attacks that would otherwise be in the "compromised Threema" threat model, such as the ones described in Sections 2.4.2.1 and 2.4.2.2. If ephemeral keys were to be freshly generated for each handshake, this problem would be confined to the single compromised session. However, by reusing the same ephemeral key for a long time, the effect of this attack is heavily amplified.

Finally, there is similar reuse of ephemeral keys on the server side. We noticed that whenever the same ephemeral key X is used multiple times by a user within a short span of time, the server will use the same ephemeral key Y as well. An attacker is thus able to force the same session key to be used for a long time by regularly connecting to the server with the same value X. We hypothesise that the server uses a least-recently-used cache for ephemeral keys y, indexed by the client's ephemeral key X.[10] If a key is not used for a period of time longer than

[10] This hypothesis was confirmed by the Threema developers: this design was necessary because `Curve25519` operations used to be excessively slow on smartphones.

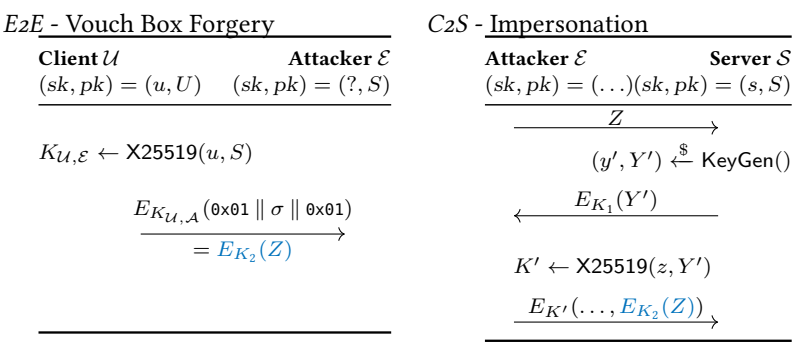


Figure 2.7: Attack 2: a cross-protocol interaction of an E2E and a C2S session. The attacker claims the public key of the server, and knows a key pair of the form $(z, Z = \texttt{0x01} \parallel \sigma \parallel \texttt{0x01})$. They convince the victim $\mathcal{U}$ to send σ to them as a E2E text message (in blue, Left side). The attacker can now start a session of the C2S protocol (Right side) where they use the "ephemeral" key pair (z, Z) and the corresponding vouch box $E_{K_2}(Z)$ (in blue) in order to authenticate as $\mathcal{U}$ to the server.

a certain threshold (approximately 24 hours) it is evicted from the cache. In our experiments, we managed to prolong the lifetime of the server ephemeral key for up to 3 months by repeatedly connecting with the same ephemeral key every hour. We note that compromising said server ephemerals would allow the impersonation of multiple users to Threema for an indefinite period of time, as long as the attacker has recorded C2S handshakes involving the revealed ephemeral keys. In fact, it suffices for the attacker to replay the `client-hello`, inducing the server to reuse a previously revealed key y. As the attacker can compute the session key, they can create a new `login` message, which contains an old vouch box, thus completing the handshake.

In order to test all these weaknesses, we created a Python script which, given a valid transcript of a C2S handshake and the client ephemeral key used, can impersonate the user to the server. The script shows all the metadata of incoming messages and can drop arbitrary messages from the conversation.

2.4.1.2 Attack 2 (Vouch Box Forgery)

An alternative route to achieve the same goal as the previous attack is to forge a new vouch box containing an ephemeral key X for which the adversary knows the private value x. This is feasible due to a cross-protocol interaction between the E2E and the C2S protocols. The consequence is that the attacker again gains the ability to permanently impersonate the victim—this time without requiring any ephemeral key compromise.

Assumptions: The attack requires two steps. In the first step, the attacker $\mathcal{E}$ has the victim use the public key S of the C2S protocol server when communicating with $\mathcal{E}$ in the E2E protocol. In the second step, the attacker, now acting as a Threema user, has the victim user send a specifically crafted text message to the attacker in the E2E protocol. Under normal circumstances, these actions should not lead to an attack: if the victim sends a message encrypted against the server's public key, then the attacker should not be able to decrypt it, nor use it for other purposes. Furthermore, sending an E2E message should not compromise security of the C2S protocol.

We will show two different methods to implement the first step of the attack. In the first method, we use part of the infrastructure of Threema to

create a new account which has the server's public key S as its own public key. In the second method, we leverage a vulnerability in a dependent library used by the Android version of Threema.

Execution: The attacker's objective is to forge a vouch box vouch for this key X that would allow the attacker to authenticate as the victim in the C2S protocol. Let (u, U) be the long-term key pair of the victim. Assume that the first step of the attack has already been executed, meaning that, whenever the victim wants to send a message to the attacker, they will use their private key u and the server's public key S to encrypt their message. Assume also that the attacker manages to find a private key x such that the corresponding public key $X = g^x$ has the following form: a `0x01` byte, followed by a string σ composed of 30 printable UTF-8 characters, followed by another `0x01` byte. If the victim sends σ as a text message to the attacker in the E2E protocol (for example, via social engineering, e.g. sending a special code to win a prize), this would result in the victim deriving a key $K^* = \mathsf{X25519}(u, S)$ and encrypting the message. Since σ is sent as a text message, the ciphertext will include `0x01` as the first byte, followed by the string σ and then PKCS7 padding at the end. The attacker hopes to obtain a ciphertext $c = \mathsf{E}_{K^*}(\texttt{0x01} \parallel \sigma \parallel \texttt{0x01}; n) = \mathsf{E}_{K^*}(X; n)$ (for some nonce n). This has probability $1/254$ of working, since this is the probability of obtaining `0x01` as a PKCS7 padding. We note that K^* here is actually equal to K_2, the key that would be derived by the victim during the C2S protocol and used in creating a vouch box. Because of this, by construction, the ciphertext c that was created and sent to the attacker is thus a valid vouch box vouch for X. Since the attacker knows the corresponding x, the attacker can now use (x, X) and vouch to authenticate to the Threema server as the victim. The attack is depicted in Figure 2.7.

Finding a good X: Creating a suitable key X is feasible, simply by randomly sampling x and computing $X = g^x$ until X has the desired form. We have empirically estimated the probability of each trial being successful in finding a good X as being $\approx 2^{-51}$. Thus, after 2^{51} trials, the success probability would be 0.63. We implemented this approach, using some batch inversion optimisations to reduce the cost of coordinate conversions. We found a key of the correct form after $2^{50.0}$ trials. In total, this required approximately 8100 core-days across a variety of computational clusters available to us.[11] The results are shown in Figure 2.8. We used this key in our experimental environment to demonstrate that the attack works in practice. We note that this is a one-time cost: once a suitable key pair (x, X) is found, it can be reused to target any user.

[11] The main computation ran on the ETH cluster Euler, using 2740 cores over 3 days, after several preparatory runs on smaller servers. A bug in our code meant that we had to run the computation twice. The estimated cost of running the attack on AWS EC2 instances (4 `m6a.48xlarge` for one week, spot pricing) would have been around 3.5kUSD at the time when we carried out the research.

Claiming the server's public key: We now describe two methods allowing an attacker to claim S as their long-term public key, the first step in the attack.

Our first method exploits the fact that Threema provides a paid API, the "Threema Gateway", for companies to interface with the Threema messaging system automatically. When registering a new account, the user is required to provide a public key. The gateway, however, does not

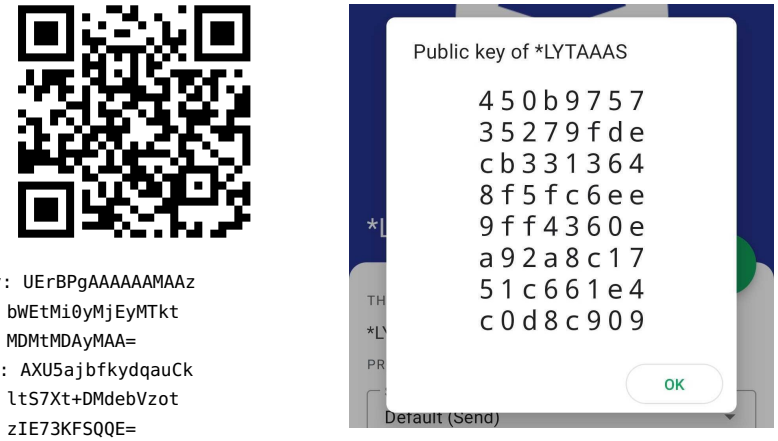


Figure 2.8: Attack 2 in practice: on the left, a suitable key pair (base64 encoded). The public key bytes 1 to 31, also encoded in the QR code, all consist of printable UTF-8 characters. On the right, the ∗LYTAAAS Threema gateway account (since revoked), with the hijacked public key of the server. User $\mathcal{U}$ sending the contents of the QR to ∗LYTAAAS as a message will allow ∗LYTAAAS to authenticate to Threema as $\mathcal{U}$.

check (1) if the user has the corresponding private key and (2) that the key does not correspond to the server's public key S.[12] This allows the creation of an account which has the public key of the server as its own. We created such a Threema account under the Threema ID "∗LYTAAAS".[13]

Our second method uses a vulnerability in one of the dependencies that Threema has on the Android version of the app for creating data backups (described in Section 2.3.6). The library used by Threema to create zip files is called Zip4j [193]. This library possesses a bug where the MAC is not checked when decrypting the zip file, if certain conditions are met. This is the case whenever a Threema backup is restored, meaning that any tampering with the zip would not be detected. In the context of Threema, this allows an attacker that has write access to the encrypted zip to modify the contacts file within it, allowing it to overwrite their public key with the server's. For example, a user might choose to save their zip file in shared storage (e.g. a cloud service, or a folder on a computer which the attacker can access), believing it to be both encrypted and integrity-protected. While this second method does not exploit a vulnerability in Threema *per se*, a more robust design would have prevented this bug from being escalated to client impersonation in the C2S protocol.

Impact: This attack yields the same effects as the previous one. The attacker gains access to messages (still E2E-encrypted), along with their metadata. This puts the attacker in a position where they can store messages meant for the victim and decrypt them at a later point, when the long-term key of the victim is compromised. The attacker, again, has the ability to drop arbitrary messages from the conversation.

2.4.2 *Compromised Threema Threat Model*

The following attacks assume adversarial server infrastructure. Attack 5 requires access to the server's secret cryptographic material. Attacks 3 and 4 only require being able to "strip away" the C2S security layer. This

[12] Note that performing this check properly is non-trivial: we also successfully registered keys $T \neq S$ such that $T = S \cdot P$, where P is a Curve25519 point of order 8. When computing a Diffie-Hellman share using the private key a and the public key T, due to clamping $T^a = P^a \cdot S^a = S^a$, thus claiming T is equivalent for our attack to claiming S.

[13] "Lose Your Threema Account As A Service"

is trivial for the Threema server, but could also be achieved by exploiting C2S protocol weaknesses, like the ones we present in Section 2.4.1.

On forward secrecy: We again stress that, from the point of view of an attacker located at the Threema server, the E2E-encrypted communications have no forward secrecy whatsoever, since it is only provided at the client-to-server level rather than the end-to-end level. This leaves the user completely vulnerable in the event of a long-term key compromise in the "compromised Threema" threat model. This is explicitly acknowledged in the Threema white paper [135].

In a messaging app, compromising the server should not give away significant data to the attacker: the server should merely act as a message router, which cannot read messages and has no gain in storing the messages that it sees, or tampering with them. Due to the lack of forward secrecy on the E2E level, this expectation is not met by Threema.

2.4.2.1 Attack 3 (Message Reordering and Deletion)

The lack of an ordering mechanism in the E2E protocol combined with the lack of authentication of the metadata in that protocol allows the server to carry out trivial reordering attacks on E2E messages. This means that the server can change the order in which the messages are delivered to the receiver, thereby potentially changing the semantics of the conversation. Furthermore, the server can withhold messages for as long as it pleases and send them at a later time with an updated timestamp, in order to evade detection. The C2S protocol does use counters, but these do not offer any protection against a malicious server in the E2E protocol.

In addition to the above, we noticed that incoming messages are shown in the order in which they are received by the Threema app, rather than being ordered by timestamp.

2.4.2.2 Attack 4 (Message Replay and Reflection)

In Section 2.3.3 we explained how Threema tries to prevent replay and reflection attacks on the E2E protocol by storing nonces of both outgoing and incoming messages. Fundamentally, this requires that the nonce database is always kept updated and is never deleted in the client app. Unfortunately, this cannot be guaranteed whenever the app is reinstalled or when the user changes device: in either case,[14] the nonce database is reset and a compromised Threema server can replay messages that were received by the victim in the past or reflect messages that the victim has sent.

[14] Note that on iOS, migrating the app to a new device using *Quick Start* would preserve the database.

To make this attack more effective, we note that the Threema server can easily tell when a victim is likely to have reset their nonce database, as long as the victim has decided to use Threema Safe as their backup option. (A reasonable assumption, since the user is strongly encouraged at registration time to use it.) Whenever a user restores a backup from Threema Safe, it will connect to the server with the C2S protocol shortly after, most likely with the same IP address. Thus, not only can the server

learn the identity associated with a given backup, but it will also know that, with high probability, the user's nonce database has been deleted, making that user a target for the aforementioned attacks.

To test Attacks 3 and 4, we created a simulated Threema server in Python to which clients can connect. The server provides control over the order of the messages and their inclusion in the conversation, and allows the attacker to save the messages in order to be later replayed or reflected. We experimentally verified all the behaviours observed above.

2.4.2.3 Attack 5 (Kompromat)

This attack provides another example of a dangerous cross-protocol interaction, in this case between the registration protocol and the E2E protocol, or the contact discovery protocol and the E2E protocol.

Execution: Recall from Section 2.3.2 that, when a user $\mathcal{A}$ registers with Threema, they must prove ownership of their public key to the server by use of a challenge-response protocol: the user combines their private key a with a public key provided by the server $X = g^x$ and encrypts a message m chosen by the server. When the contact discovery protocol is run (see Section 2.3.7), the same challenge-response protocol is executed to prevent an excessive number of queries. The encryption method used is identical to that in the E2E protocol. In this protocol, the server does not necessarily have to own the private key x corresponding to the provided ephemeral public key, and they can present the long-term public key of any other Threema user $\mathcal{B}$ instead. Thus, $\mathcal{A}$ will be encrypting the challenge with the same key that they would use to communicate with $\mathcal{B}$ in the E2E protocol. Hence, the resulting ciphertext is a valid E2E message that can be sent to either $\mathcal{A}$ or $\mathcal{B}$ (and which can be made to appear to come from the other party).

Impact: This attack targets the integrity of the E2E protocol in Threema. In the unpatched protocol, when the registration protocol is run, or every time the contact discovery protocol is run (usually once per day), the server is able to forge a valid ciphertext. Each ciphertext can only be sent once per user due to the replay/reflection protection provided by the nonces. This flaw was already fixed when we re-discovered it. At the time our research was carried out, the app required that the message m provided by the server have a type byte equal to `0xff` when either the registration or the contact discovery protocol is run. This type does not correspond to any valid message type, thus preventing it from being accepted as a valid message. While this does effectively prevent the attack, it does not tackle the other cross-protocol attacks that we present. In Section 2.5 we discuss mitigations that offer full protection.

2.4.3 Compelled Access Threat Model

The following attacks are in the strongest threat model of compelled access, which grants the attacker access to an unlocked device. These attacks allow the recovery of the long-term key of the victim. Due to the

lack of forward secrecy and post-compromise security, this irreversibly forfeits all security. For instance, considering a combined attack setting of compelled access *and* a compromised Threema server, we can obtain an attack that can decrypt all past and future communications.

2.4.3.1 Attack 6 (Cloning via Threema ID Export)

Assumptions: Access to an unlocked phone, with no Threema App UI lock.

Execution: Among the backup methods provided, Threema allows the user to export the Threema ID (as explained in Section 2.3.6). However, this method directly exposes the long-term secret key of the user, and does not by default require any form of authentication, allowing a compelled access attacker to backup the private key with a password of their choice and later decrypt it on their own device.

This is an intentional feature in Threema, but one which we deem to be a grave security danger: all security in Threema relies on maintaining the secrecy of long-term private keys, and this attack allows an adversary to recover such keys by having access to the phone and app for a few seconds.

Impact: While similar mechanisms are present in other messengers, their security impacts are much smaller. For instance, Signal allows users to transfer their accounts to a new device, but it requires physical proximity of the new and the old devices, and disables the old device permanently after such a transfer is completed. This is in contrast to Threema, where a Threema ID export is undetectable on the victim device after it has happened. Such a scenario is particularly impactful in the context of intimate partner violence [122] and due to the usage of Threema by high-ranking officials [248, 119].

The Threema app does provide some optional locking mechanisms. For example, the app can be locked with a PIN, a passphrase or by using biometrics, thus preventing this attack. However, in an oppressive compelled access setting, they would provide only limited protection.

2.4.3.2 Attack 7 (Compression Side-Channel)

While the app locking mechanism does prevent a compelled access attacker from accessing the user's data via the user interface, it does not prevent background processes from being executed. In particular, the app will attempt to make a new Threema Safe backup whenever the application is launched if the last successful backup is more than one day old. Moreover, incoming messages will be processed and may affect the contents of the backup. We exploit these facts in our final attack to recover a victim user's private key.

Assumptions: Access to an unlocked phone, ability to observe the network.

```
{ "info": { ... },
  "user": { "privatekey": <private key>,
            "nickname": "user_nickname" },
  "contacts": [
    { "identity": "ABCDEFGH",
      "nickname": "other_user_nickname",
      ... } ],
  ... }
```

Figure 2.9: Example of a backup, with the private key (normally 44 base64 characters) redacted.

Execution: Recall that a Threema Safe backup contains the private key of the user, and a list of the user's contacts. Before being sent to the server for storage, the backup is *compressed and then encrypted*. It is well-known that this combination of processing steps can be vulnerable to attack, especially if the attacker can control part of the plaintext before compression [180, 251]. Specifically, the length of the ciphertext may leak information about how much plaintext compression has been achieved, and this in turn can leak information about the degree of overlap between the attacker-controlled and the unknown parts of the plaintext. In Threema Safe, the XSalsa20 stream cipher is used for encryption, so a ciphertext leaks the exact length of the plaintext. Furthermore, usernames are handled locally in Threema. This means that if a user $\mathcal{U}$ sends a message to victim $\mathcal{V}$, then $\mathcal{U}$'s Threema ID and chosen nickname are updated in the local contact list of $\mathcal{V}$ on receipt of the message. This happens even if $\mathcal{V}$'s app is locked. This provides a means for an attacker to inject chosen plaintext into the backup of a victim. In this way, the conditions needed to build a compression-based side-channel attack are fulfilled. We provide a more detailed description of the attack below.

Detailed description: The compression format used by Threema is gzip [101], with the DEFLATE [100] compression algorithm. DEFLATE internally applies an LZ77 [294] pass to the input data, replacing repeating strings with back-references, and then applies Huffman coding [163] to the result. LZ77 scans the input data; within a sliding window (of default size 32KiB), it looks for the longest substring that matches the current position in the input. If such a substring is found, the substring is replaced with a back-reference to its previous occurrence. In this attack, we only target the LZ77 component.

From a high-level perspective, the attack leverages the fact that nicknames are handled client-side: when a message is received, the nickname contained in the metadata is set as the current nickname of the sender. Since nicknames are automatically included in the backup, this gives the attacker partial control of the plaintext in the victim's backup. The objective of the attacker is to leverage this partial control in order to force a back-reference to be created from the attacker's nickname in the `contacts` field to the private key in the `user` field. We depict an example of a JSON backup, with only the relevant information present, in Fig. 2.9. If the attacker's nickname and the private key are sufficiently close together, they will both fall within the same sliding window of the

compression algorithm. This ensures that there is the possibility of a back-reference being created. In order to increase the chances of such a back-reference being created, we include a string called a *canary* containing a known part of the string that we want to leak. This tricks the compression algorithm into finding redundancy between the nickname and the private key, creating the compression side-channel that we need.

For our experiments, we used two canaries: "`privatekey`" for the beginning of the private key and "`=`" for the end of the private key. `privatekey` forces a back-reference to be created right before the private key in the backup, therefore ending the current match: this technique has been formally studied and dubbed *align gadget* in [272].[15] The equals sign always appears as the last base64 character of the key due to padding of the private key during encoding, since the key is 32 bytes long and must be padded to a multiple of 6 bits. Between the two canary strings, the attacker needs to place, in order, their guess for the next character of the key, and the currently known characters of the key (initially none, thus an empty string). If the guess is correct, it should result in a shorter Threema Safe ciphertext. If the shortest ciphertext was induced by only one character among all guesses, then the currently known characters of the key are updated to include the current guess. Otherwise, two-character guesses can be attempted. This method effectively tries to guess characters of the base64 encoding of the private key from the last one to the first one.

[15] This was not known to us at the time this research was carried out; we stumbled upon the requirement for this first canary by trial and error.

To obtain the required queries, it suffices for an attacker to forcibly close the app and start it again. This prompts the app to retry uploading the backup. The forced closure can be done by either using debugging tools or by simulating user input on the unlocked phone. Then, in the setting of a compromised Threema Safe server, it is sufficient for the server to reply with an HTTP error code. Otherwise, a network attacker can forcibly close the connection, making the upload fail. Both of these techniques ensure that a new upload will be attempted at the next restart of the app.

Not all the bytes of the private key have to be recovered, since algorithms such as the van Oorschot-Wiener algorithm [226] can be used to recover the remaining bits. Indeed, this creates a trade-off between the number of queries and the amount of offline work required by the attacker. Since the latter does not require access to the device, it is convenient to execute just the sufficient number of queries required to make the offline attack feasible. For example, the van Oorschot-Wiener algorithm has time complexity $\mathcal{O}(2^{n/2})$, where n is the bit-size of the set to be searched, and it is parallelisable. Recovering 31 base64 characters (that is, 178 bits, accounting for the 2 bits of padding and the known "`=`") via the compression side-channel described above leaves 13 characters to be recovered, equivalent to a search space of size 2^{78} after base64 decoding. This makes the offline part of the attack perfectly feasible.

Proof-of-Concept: We created a custom Threema server and instrumented the client code in order to redirect requests to our server rather than the actual Threema Safe server. This allows us to both see the

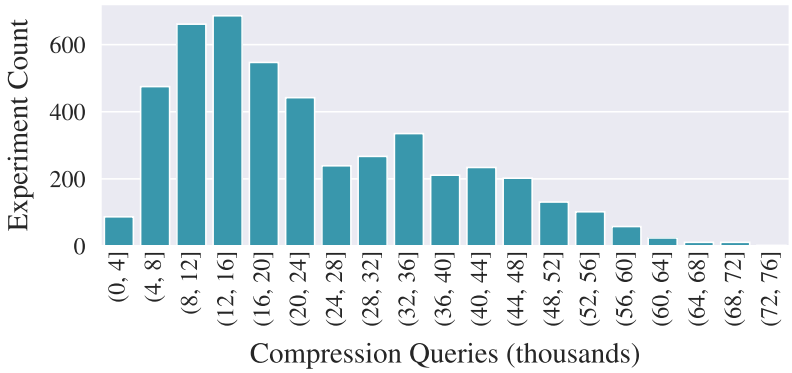


Figure 2.10: The distribution of the number of compression side-channel oracle queries in our successful key-recovery experiments. Each "query" is a backup attempt by the Threema app. The x-axis shows the number of queries, grouped into bins of width 4,000. The y-axis shows the number of experiments falling into each bin. The minimum number of queries was 1,920, the maximum 90,176 and the mean 23,382.

length of the backups and to make backups fail by returning an HTTP error code. To automatically induce a new backup, we use Android debugging tools which can force an app to stop and restart. In order to measure the number of queries needed to complete the attack at scale, we also reimplemented the client backup mechanism, and simulated the attack locally. Figure 2.10 summarises our experimental results. We successfully recovered the user's private key in 4.7k out of 10k experiments; in these successful runs, we required a median of 19.4k backup attempts (mean 23.4k).[16] In our PoC, each attempt requires around 2s, implying a total running time of around 11 hours for the attack to recover 31 base64 characters of the private key.

[16] We remark that "failure" here means that our *automated* exploit did not successfully retrieve the key. If a targeted attack were to be run, one could manually guide the process. In fact, when a failure happens it is often because the exploit begins revealing characters of other parts of the backup, which can be easily spotted as the exploit is running.

The backup is normally transmitted to the server on a TLS connection, but with modern cipher suites (AES-GCM or ChaCha20-Poly1305), TLS does not hide the length of its payloads. So we consider the attack to remain viable even if the attacker can only see TLS-protected traffic as in our compelled access threat model.

Impact: Recovering the private key leads to a complete loss of security: after doing so the attacker can impersonate the user in any action and decrypt E2E protocol messages.

2.5 MITIGATIONS

We describe the mitigations we proposed to Threema, along with the ones that Threema employed in practice, and comment on what our attacks mean in practice for Threema users.

Forward Secrecy at the E2E level: We identify the lack of forward secrecy as one of the main weaknesses of Threema, since it accentuates the severity of many of our attacks. There is no simple fix: a protocol redesign supporting the use of ephemeral keys is necessary. We recommend that Threema adopt the Signal protocol, whose cryptographic API is available through the libsignal library [196]. The Signal protocol has received extensive security analysis [125, 80, 184, 81] and provides forward secrecy and post-compromise security at the end-to-end layer. We note that Whatsapp has transitioned towards using the Signal proto-

col in the past [205], showing that such a change is feasible even for a messaging app with a huge user base.

Following the disclosure, Threema released Ibex, an additional protocol developed independently of our work. Ibex is layered on top of the E2E protocol, and aims to provide forward secrecy via an asynchronous, ephemeral key exchange. The keys derived from this exchange are used to instantiate a symmetric ratchet analogous to Signal's. The lack of an asymmetric ratchet means that Threema still lacks post-compromise security. Ibex was opt-in during the initial phases of the roll-out, but should now be default for all users; as noted in Section 2.2.1, it is at the time of writing incompatible with the multi-device mode. Assuming any past compromise of the Threema server, forward secrecy of conversations that happened prior to the Ibex roll-out would be irrecoverable; to the best of our knowledge, though, Threema has not been compromised.

Strengthening the C2S Protocol: The least invasive modification to the C2S protocol that directly prevents our attack of Section 2.4.1.2 would be to include the server cookie inside the vouch box. This ensures that, as long as the server picks a fresh cookie every time the protocol is run, the vouch box cannot be trivially replayed. We recommend that Threema replaces its bespoke C2S protocol with the IK protocol of the Noise Protocol Framework [235], which allows for 0-RTT encrypted communications between the client and the server and provides authentication and forward secrecy. Furthermore, it prevents the attack of Section 2.4.1.1 (Impersonation by Randomness Failure). The Noise protocols have been analyzed and proven secure [107, 108], and Noise protocols have seen practical deployment in Wireguard [104] and Whatsapp [237].

Threema fixed attacks 1 and 2 via key separation using an additional KDF step with distinct labels, as well as by including the server cookie in the vouch box. Caching of ephemeral keys has been removed on the client and server. We do not believe these attacks affected Threema users at large, since we are not aware of practical mechanisms to recover ephemeral keys for (attack 1), and Threema reported that no Threema Gateway account was registered with the public key of their server in the past (attack 2).

Preventing Cross-Protocol Attacks: The attacks in Sections 2.4.1.2 (Vouch Box Forgery) and 2.4.2.3 (Kompromat Forgery) exploit the fact that the same "X25519-then-Encrypt" paradigm is used multiple times for different purposes but with the same keys. By using payloads created in one protocol in another, the adversary gains capabilities that they would not have when viewing the protocols separately. A secure application should cryptographically compartmentalise its protocols, taking a very conservative approach when using the same cryptographic material in different protocols (if doing so at all). A common way to prevent this class of attack is to use proper *key separation*: the X25519 key material will then only be used to derive context-dependent keys using a KDF with proper labels. We stress that key material that is used to derive other keys should not be used itself for other purposes, since it is important to ensure that leaking one of the keys during its usage does not

affect any other key, i.e. the keys should remain effectively *independent* of each other.

In correspondence, Threema indicated that they would make extended use of KDFs to ensure key separation. The 2026 white paper [136] shows BLAKE2b *being used for key separation in the newly designed protocols (e.g. multi-device), but does not fully document the registration, contact discovery, E2E, and C2S protocols. We did not study this new design in depth, but we did confirm that registration and contact discovery now use separate keys, derived with different labels from the C2S protocol key [141].*

Protecting Metadata: The simplest way to prevent the attacks of Sections 2.4.2.2 (Replay and Reflection Attacks) and 2.4.2.1 (Message Reordering) is to protect the integrity of the metadata contained in the E2E packet. This can be simply achieved by including the metadata in the (authenticated but not encrypted) "additional data" field of the AEAD scheme. This modification has zero message overhead and supersedes the usage of the metadata box, eliminating the need for an additional ciphertext and tag. Protecting the metadata in this way would prevent an adversary from swapping the source and destination of the message and thus prevent reflection attacks. To prevent message reordering attacks, the app could use a per-recipient counter for the AEAD nonces in the E2E protocol. This would also allow the recipient to detect lost or adversarially deleted messages, but would still leave the problem of how to avoid nonce repetitions when the Threema app is reinstalled or when the user changes device. We suggest that the app should re-run the registration protocol in this event, establishing a fresh key pair (a, A).

Threema's introduction of Ibex fixes the reordering and replay attacks. The metadata box is now mandatory. Threema did not adopt our suggestion of using AD, citing backwards compatibility concerns; additionally, the NaCl library they depend on does not expose an interface for setting additional data during encryption. Attacks 3 and 4 have limited implications if the underlying E2E plaintext is unknown, so they are unlikely to have affected Threema users.

Preventing Cloning via Threema ID Export: The ID export feature should be protected, possibly by requiring the user to provide a passphrase chosen at registration time, or at least by enforcing the use of the phone OS's authentication mechanisms such as biometrics and PIN codes. It is up to Threema to choose how to balance usability and security, but we would err on the side of the latter.

The client now keeps track of ephemeral keys used in the C2S protocol and the user is alerted if an ephemeral key has been used in a C2S session that does not match the main session. Attack 6 is the most likely to have been exploited in the real world. This attack did not impact Threema users at large, but left at-risk individuals vulnerable to targeted attacks.

Mitigating the Compression Side-Channel: Threema should avoid using compression. To avoid an explosion in server-side storage requirements, a binary serialization format such as CBOR [67] could be used instead of JSON.

Threema disabled compression in backups. Attack 7 has serious implications, since observing the size of past Threema backups provides side-channel information on a user's secret key, but does require targeting a specific user.

2.6 CONCLUSIONS

Threema is marketed as a secure app, suitable for both individuals and companies. It is used by government departments and high-profile politicians. On the other hand, the seven attacks we have presented highlight fundamental weaknesses in the design of Threema. Indeed, the Threema protocols lacked basic properties that are nowadays considered *de rigueur* for a messenger app to be regarded as secure: forward secrecy with respect to a malicious server, and protection against replay, reflection, and reordering attacks. The cryptography in Threema showed design flaws that needed to be addressed in order to meet the security expectations of its users. We have not analysed Threema again after our original research was carried out. All of our attacks have been mitigated with targeted patches, without changing the Threema architecture in depth, or introducing breaking changes in most of the protocols. While the security analysis of Ibex is a step in the right direction, in the absence of a full security proof, it remains to be seen whether Threema will withstand possible future attacks.

Stepping back from the details of our analysis, we believe that our work presents (at least) three useful lessons for developers when deploying complex cryptographic protocols.

First, *using modern, secure libraries for cryptographic primitives does not on its own lead to a secure protocol design*: It is possible to misuse libraries such as NaCl and libsignal when building a more complex protocol and developers must be wary not to be lulled into a false sense of security. A pertinent example is the C2S protocol in Threema, which has many issues with ephemeral keys. Another recent example is Bridgefy's mis-integration of libsignal [13]. In some cases, it is the API of the libraries themselves that encourages bad cryptographic practice. For instance, NaCl's authenticated encryption routines do not allow for associated data. This was the reason for the Threema's bespoke metadata box construction. To their credit, Threema's developers did (largely) avoid "rolling their own crypto" but this advice should be extended to "don't roll your own cryptographic protocol". Of course this advice is only useful if good alternatives are available. In the particular case of the C2S protocol, Threema could have used the Noise IK protocol or TLS.

Second, *beware of cross-protocol interactions*: A standalone protocol may appear to be secure, but cross-protocol interactions can undermine security guarantees, as we have shown in Threema with the vouch box and Kompromat attacks. So developers should be careful not to introduce such vulnerabilities. Every message should be cryptographically bound to its protocol and thereby rejected whenever it is used in a different protocol. This can be done by strictly following the key separation principle

– use different keys for different purposes. Notably, a recent analysis of the secure messaging protocol Matrix also exploited cross-protocol interaction to violate the confidentiality of messages [16].

Third, *proactive, not reactive security*: Our inability to find an attack on a protocol does not imply it is secure: new attacks could be found at any moment and known attacks only get stronger over time. Often, software releases follow a *design-release-break-patch* process (a *reactive* approach [227]). This can be inconvenient for users and may also lead to the need to maintain backwards compatibility, opening the way to possible downgrade attacks [172, 51, 9, 32]. Developers should adopt a *proactive* approach, where the protocol is formally analysed during the design stage. Ideally, different forms of analysis would be carried out, e.g. using symbolic approaches or by producing computational security proofs. While this cannot exclude every attack possibility, it can help improve confidence in the protocol's security. Furthermore, protocols should be designed in such a way that this formal analysis can be more easily carried out. In the present setting, because the C2S protocol uses its session key directly in the handshake (to derive the metadata key), it is impossible to prove session key security, a standard property targeted in security proofs for key exchange protocols.

None of these lessons is fundamentally new, but given the results of our analysis of Threema, and the recent works in the field of secure protocol analysis [16, 13, 20], they apparently bear repeating.

Chapter 3

MFKDF: Multiple Factors Knocked Down Flat

This chapter has been previously published as:

> **Scarlata, M.**, Backendal, M., Haller, M. "MFKDF: Multiple Factors Knocked Down Flat." *33rd USENIX Security Symposium (USENIX Security 2024), pp. 4301–4318. USENIX Association, August 2024.*
> DOI: https://doi.org/10.3929/ethz-c-000682942

For this chapter, I have made minor modifications from the original publication, including extended details that were present in the full version [260] or previously unpublished.

Author contributions
This chapter is based on joint work of all co-authors. This work was born out of a discussion between me, MB and MH in the *hallway track* of Crypto '23, Santa Barbara, as we went from excitement about using MFKDF to dismay about its weaknesses. MB formalized MFKDF and came up with a new syntax for factor constructions; in the process, she discovered several attacks. MH discovered the HOTP bias attack, carried out its analysis and implemented simulations. My main contributions were: realizing that one-timeness was a fundamentally unattainable goal for key derivation; carrying out a full analysis of the reference implementation of MFKDF; discovering most of the initial attacks; implementing all of the PoCs.

Summary In this chapter, we show that the Multi-Factor Key Derivation Function (MFKDF) framework proposed by Nair and Song at USENIX Security 2023[220] falls short of their stated security goals. MFKDF allows users to derive cryptographic keys based on multiple inputs: factors such as passwords and one-time codes, and a public state. Underspecified cryptographic primitives and the lack of integrity of the public state lead to several attacks, ranging from full key recovery when an HOTP factor is compromised, to bypassing factors entirely or severely reducing their entropy. We reflect on the different threat models for key-derivation and authentication, and conclude that MFKDF is always weaker than plain PBKDF and multi-factor authentication in each setting.

3.1 INTRODUCTION

Chapter Overview:

3.1 Introduction 42
3.2 Background 47
3.3 Attacks 57
3.4 Mitigations 72
3.5 Discussion 74
3.6 Conclusions 76

With more and more data stored online or distributed across multiple devices, an increasing number of security-sensitive applications face the challenge of combining data availability with user-friendly key management. The traditional solution is passwords, for both authentication and key derivation. Password-based systems are relieved from some of the difficulties of key management, since users do not have to handle cryptographic keys. In return, allowing (and even encouraging) human-memorable secrets means that passwords often have low entropy, come from a small and predictable "dictionary" and may be highly correlated with one another.

In response to a wide variety of attacks stemming from these issues, password-only authentication to web services is being phased out. Instead, users are offered a two step verification process, where they need to provide a second "factor" in addition to their password. This is known as Two-Factor Authentication (2FA), or more generally, Multi-Factor Authentication (MFA). Examples of such second factors include Time-Based and HMAC-Based One-Time-Passwords (T/HOTP), Out-Of-Band (OOB) authentication and codes generated by hardware tokens such as YubiKeys. Surveys show a sharp increase in the adoption of MFA among users [79], and cybersecurity advisory bodies such as CISA now advise the use of MFA in their recommended best practices [88].

In contrast, passwords are still commonly used as the sole source of entropy for key derivation, as input to Password-Based Key Derivation Functions (PBKDFs), in systems performing client-side encryption. Examples of this include full disk encryption, end-to-end encrypted cloud storage, password managers and cryptocurrency wallets. In all of these settings, keys derived from passwords remain as (in)secure as the passwords from which they stem.

MFKDF. In an attempt to remedy this, Nair and Song [220] (NS) propose the concept of a "Multi-Factor Key Derivation Function" (MFKDF) as a way to integrate the advantages of multi-factor authentication into password-based key derivation. MFKDF extracts entropy from existing authentication factors, such as T/HOTP, and combines them with the password before applying a PBKDF to derive a key. The goal is to derive a key that has the entropy of all the factors and password combined, and to retain security even against partial compromise of the source key material (i.e., the password and the authentication factors). The security guarantees claimed for MFKDF state that attackers should not be able to separately guess individual MFKDF factors, and that the difficulty of correctly guessing all factors simultaneously gives an "exponential security improvement over password-based key derivation" [220].

NS propose to use MFKDF-derived keys directly in other applications requiring a symmetric key, such as database encryption in a password manager. We will refer to this setting as the *key derivation* setting. Additionally, the authors state that MFKDF-derived keys can be used to authenticate users, as a replacement for MFA, since an MFKDF key im-

plicitly verifies the user's authentication factors. We will refer to this setting as the *authentication* setting.

The concept of MFKDF was presented at USENIX Security 2023 [220], and is accompanied by a Javascript reference implementation [217].

3.1.1 Our Contributions

We analysed MFKDF and discovered several issues, ranging from a fundamental problem with the idea of using authentication factors for key derivation (Attack 3.3.1), to vulnerabilities in the proposed constructions (Attacks in 3.3.2–3.3.3). We briefly outline the attacks below.

1. The **Dynamic Factor Attack** (3.3.1) surfaces a critical mismatch between the goals of key derivation and authentication, and shows that the most common authentication factors are unsuitable for key derivation. By construction, key derivation functions are deterministic; they need to always re-derive the same key on the same input. Hence, MFKDF extracts *static* key material from so called *dynamic* authentication factors, such as T/HOTP. When used for authentication, dynamic factors produce different responses to authentication challenges. Leakage of such one-time codes does not result in a permanent degradation of security: a later authentication attempt would use a different code, which cannot be derived from the leaked one. This attack shows that an adversary can recompute this static key material from a single one-time code from the authentication factor, thereby permanently degrading the security of MFKDF. This is in sharp conflict with the original security models of TOTP and HOTP.

The threat models for authentication and key derivation allow for different kinds of leakage of secret values. In particular, "one-timeness" of authentication codes is not reflected in the confidentiality requirement of KDF inputs. So while the idea of using authentication factors to strengthen password-derived keys is praiseworthy, our first attack shows that—because of these disparate threat models—the use of existing authentication factors for key derivation is problematic.

We present many other attacks, showcasing vulnerabilities in the MFKDF construction that undermine the claimed security guarantees. These attacks fall into two main categories:

2. The **State Integrity Attacks** (3.3.2) leverage the fact that a malicious server can modify the MFKDF state: this allows the adversary to control the output of the key derivation, to bypass factors, and even to recover derived keys when users interact with the MFKDF construction.

3. The **Specification Attacks** (3.3.3) exploit incorrect or missing assumptions in the specification of MFKDF. In particular, we show that, as specified in the version we analysed, MFKDF built with

H/TOTP factors is insecure. Additionally, MFKDF relies on properties of symmetric cryptography that do not follow from traditional security notions such as IND-CPA, and on the (lacking) information theoretical security of the many-time pad. These mistakes, and others, invalidate the security theorems presented by NS.

Most of the above attacks impact not only the formal specification of MFKDF, but also the reference implementation. This highlights how mistakes at the specification level transform into practically exploitable vulnerabilities in code.

After presenting the attacks, we propose countermeasures for the issues which can be mitigated, and draw wider conclusions from the remaining fundamental problems. We argue that the proposed combination of MFA and PBKDF fails to recognize the distinct threat models for authentication and key derivation, and is therefore irrecoverably flawed.

In the authentication setting, we show that MFKDF is always weaker than corresponding traditional MFA choices. Additionally, we believe that the statement that MFKDF can be implemented with "no noticeable change to the user experience" may lead to careless handling of one-time codes, which are much more security critical when used for MFKDF than for traditional MFA. We hence caution against the use of MFKDF in this setting.

In the key derivation setting, our attacks show that the usage of MFKDF can annul the security of a strong password. We propose mitigations to these issues, but show that even after patching the vulnerabilities, MFKDF does not provide a significant advantage over a naïve PBKDF construction which derives the key from the concatenation of multiple static secrets.

3.1.2 Related Work

MFA and Key Derivation. The state of the art in multi-factor authentication includes WebAuthn [160], a part of the FIDO2 Project [25]. WebAuthn enables completely passwordless authentication, thus sidestepping the problem of user passwords. WebAuthn credentials, also known as Passkeys [73], leverage public-key cryptography for authentication, and can either be bound to a hardware authenticator (such as a YubiKey [154]), or synchronized across user devices.

For local authentication and key derivation, modern approaches include relying on secure hardware to achieve higher security guarantees. Examples include the use of Trusted Platform Modules (TPMs) [214, 167] for disk encryption and smartphones leveraging secure chips [167, 287] to defend against brute-force attacks on PINs and passcodes.

In the online setting, Oblivious Pseudo-Random Functions [222, 123, 174] (OPRFs) let a client derive a key as a function of its password and a server secret, without the server learning the password or the derived key. Combining the entropy of the password with the server secret increases the cost of offline dictionary attacks on the password for external adversaries, making OPRFs a promising alternative to PBKDFs.

KDFs that combine multiple inputs have emerged in the literature surrounding hybrid KEMs [106, 132, 55] and hybrid key exchange [114], in order to support entropy contributions from both classical and post-quantum primitives; key derivation for messaging may also require multiple key inputs, such as the combination of pre-shared keys in MLS groups [39] and the combination of different Diffie-Hellman secrets in Signal's X3DH [206].

KeeChallenge and OtpKeyProv. KeePass [245] is a password manager that supports authentication factors, such as HOTP and HMAC challenge-response authentication [290], in addition to a password to encrypt the application database. This integration is implemented through two plugins, KeeChallenge [256] and OtpKeyProv [246], which both predate MFKDF, as shown from historical snapshots of the plugin list on the KeePass website [199]. While an in-depth analysis of the security of these plugins is outside of the scope of this chapter, we note that these plugins also suffer from some of the same problems as MFKDF. The dynamic factor attack applies here as well: the plugins need to extract static key material from the one-time codes. In addition, we show in Attack 3.3.3.2 that HOTP codes are not uniformly distributed: unless the plugins account for biases in the distribution, their security would also be degraded.

MF*. MFKDF also appears in other works by NS. The custodial wallet design [218] is based on MFKDF and inherits its vulnerabilities, as does their follow-up implementation of an MFKDF-based Ethereum wallet [266]. The MFCHF credential hashing function [219] is vulnerable to our attacks on factor constructions.

Attacks. There is a flourishing variety of attacks on cryptosystems in the wild. Key-overwriting attacks in the style of our OOB attack began with Klima and Rosa [183], and have seen a recent resurgence in modern cryptanalysis [72, 38, 19]. Lack of integrity in metadata has been exploited in practice in recent attacks in messaging applications [20, 229], but its perils have been long known by the community [231].

Non-uniform distributions were exploited in the contexts of biased ECDSA nonces [69], randomness failures in RSA key generation [158], and RC4 [202]. While RFC 4226 [197, A.4] discusses why HOTP biases are negligible in the authentication setting, to the best of our knowledge we are the first to construct and evaluate an attack that exploits the bias when HOTPs are used for encryption.

PBKDF constructions have been analyzed before [26, 288], but MFKDF aspires to be an entirely novel primitive, warranting its own analysis.

Follow-ups. The theoretical foundations of KDFs that take multiple inputs have been first systematically studied by Backendal et al. in [36], which lays down security definitions and potential constructions for what they dub *n-KDF*s. After our research was carried out, Roberts, Nair and Song went on to define MFKDF2 [253], a follow up to MFKDF that aimed to patch all the security vulnerabilities we identified with the original work. A security audit of the MFKDF2 implementation was

carried out by Scarlata [252], highlighting some security issues that were fixed in later releases of the library. At the time of writing this thesis, a formal security analysis of MFKDF2 is still missing. In parallel, Boyen et al. [68] proposed the notion of "Two-Factor Authenticated Key Exchange", asymmetric Password-Authenticated Key Exchanges (PAKEs) where the password is augmented with a t-bit one-time code computed by a second device. The authors also propose two efficient[1] instantiations, and prove their security. This work does not contradict our fundamental attack, since the PAKE does not derive a deterministic key, but shows that cryptographically secure multi-factor constructions are possible.

[1] These protocols remain heavyweight, due to the reliance on homomorphic encryption, zero-knowledge proofs, and non-standard one-way functions with algebraic properties. No concrete implementations or benchmarks have been published.

Iliescu's MSc thesis [165], supervised by Scarlata, Backendal and Paterson, explores the use of OPRF evaluation as a way to implement second factors for key derivation. For instance, an OPRF key can be placed on a smartphone, and used to interactively evaluate the OPRF on input a static secret like a password. With such a design, even with access to all the messages of the OPRF flow, an adversary would need to compromise both the password and the OPRF key in order to derive the final key. This would improve security in the setting where the software running on the user's client (e.g. their laptop) is not compromised, but all of the inputs are (e.g. because of a hardware keylogger and network interception).

3.1.3 *Ethical Considerations*

We disclosed our findings to Nair and Song on 01/17/2024 and they acknowledged receiving them on 01/22/2024. While NS argued that the underspecification exploited by some of our attacks is intentional for "flexibility and future-proofness" of MFKDF, they confirmed the vulnerabilities in their implementation, as well as that Attack 3.3.1 is unavoidable. We discussed the mitigations from Section 3.4 with NS. They intended to modify the MFKDF specifications to highlight the importance of protecting the integrity of the MFKDF state and the use of secure encryption primitives.[2] They clarified that the MFKDF implementation was intended as a proof of concept; they were unaware of any deployment of their code, and they would implement any necessary changes prior to use in production applications.

[2] This has been partially realized with the release of MFKDF2 [253].

3.1.4 *Chapter Structure*

We start by defining a formal syntax for the MFKDF constructions and primitives in Section 3.2, adapting it slightly from the syntax presented in [220] for the sake of clarity.[3] Section 3.3 presents our attacks against the MFKDF construction and reference implementation. Section 3.4 follows with the possible mitigations. Finally, Section 3.5 discusses the inherent flaws in MFA-based key derivation functions, and the different threat models of key derivation and multi-factor authentication. We conclude and propose future directions in Section 3.6.

[3] Note that none of our attacks are affected by this change in syntax; we deviate from the original MFKDF description only in the interests of clarity and internal consistency.

3.2 BACKGROUND

Notation and Conventions. We denote assigning value x to variable a by $a \leftarrow x$, where x might be the output of an algorithm. Similarly, $a \leftarrow\!\!\$\ \mathcal{X}$ denotes sampling uniformly at random from the set $\mathcal{X}$ and assigning the result to a. We write $[0, 10^6)$ for the right-open interval of all integers from 0 (inclusive) to 10^6 (exclusive). The symbol $\|$ denotes string concatenation and $\oplus$ denotes XOR. For an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$, $\mathsf{Enc}(k, m)$ denotes the encryption of message m under key k and $\mathsf{Dec}(k, c)$ the decryption of ciphertext c.

3.2.1 *Authentication Factors*

Following [220], we define a *factor*, F, to be a two-valued probability distribution (σ, α) generated by an efficient probabilistic algorithm, which we also refer to as F. The *factor material* σ represents the secret part of the factor (for example, a password, or a key for the HOTP algorithm) whereas the *factor parameters* α represent the public knowledge about σ.

An authentication factor additionally specifies a protocol between a prover and a verifier, consisting of a *witness generation* algorithm and a *verification* algorithm. The witness generation algorithm, run by the prover, takes the factor material σ and some optional additional *witness input* inp to produce a witness W. Depending on the protocol, the optional input inp may be generated by the verifier and sent to the prover (e.g., in challenge-response protocols), or be part of the state kept by both parties (e.g., a counter). The verification algorithm takes as input σ, inp (if present) and W, and checks if W is a valid witness.

As an example, in the case of passwords the witness W is often the password σ itself. For T/HOTP, W is a one-time password, and inp the current timestamp or counter value used to generate it. The secret factor material σ is assumed to be shared (for example, during a setup phase) between the prover and the verifier, such that at the time of authentication, only the witness needs to be communicated. We say that an authentication factor is *dynamic* if the witness changes with every use of the factor, and learning W does not allow an adversary to compute σ.

Below, we review the dynamic factors used in MFKDF.

HOTP and TOTP. The HOTP algorithm, defined in RFC4226 [197] and shown in Algorithm 1, takes as input a static symmetric key k_h (the secret factor material σ) and a counter ctr (as the witness input inp). It first generates a 20-byte digest hs by computing HMAC on the key and counter, then truncates the digest to 31 bits using a dynamic truncation algorithm that we call $\mathsf{DynTrunc}$.[4] Finally the truncated digest is transformed to an integer using the function $\mathsf{BitsToInt}$, which on input a bitstring $b_n b_{n-1} \ldots b_0$ returns the integer $s_{int} = \sum_{i=0}^{n} b_i \cdot 2^i$. The reduction of s_{int} to d digits is the resulting one-time password $\mathsf{HOTP}(k_h, ctr)$, which constitutes the witness W.

The verification algorithm simply recomputes the one-time password and checks that $\mathsf{HOTP}(k_h, ctr) = W$.

[4] This algorithm is defined on page 7 of RFC4226 [197], but the details do not matter for our purposes.

Algorithm 1 HMAC-based One-Time Password

procedure $\mathsf{HOTP}(k_h, ctr)$ ▷ Witness generation
  $hs \leftarrow \mathsf{HMAC\text{-}SHA\text{-}1}(k_h, ctr)$
  $s_{bits} \leftarrow \mathsf{DynTrunc}(hs)$ ▷ $|s_{bits}| = 31$ bits
  $s_{int} \leftarrow \mathsf{BitsToInt}(s_{bits})$
  return $s_{int} \bmod 10^d$ ▷ Witness W

[5] Additionally, TOTP may use a different hash function than HOTP.

TOTP [198] is defined as $\mathsf{TOTP} := \mathsf{HOTP}(k_h, T)$[5] where the counter ctr is replaced with T, a coarse-grained integer representation of the current time. Note that for both HOTP and TOTP, the PRF security of HMAC [44, 43] implies that access to old one-time codes does not allow an attacker to produce new ones or recover k_h, making these factors dynamic.

Out-Of-Band Codes. A popular non-cryptographic way to implement a dynamic factor is to send one-time codes over an "out of band" channel (e.g., SMS or email). In this case, there is no shared secret material, i.e., σ is empty. Instead, the server generates a random one-time code as the witness input inp, and sends it to the user over the OOB channel. The user then sends this value back to the server over the main channel. Hence, the witness is directly the input inp. The server verifies that $inp = W$, i.e., that the user sent back the same code that the server generated. This kind of factor lacks cryptographic guarantees, and assumes the OOB channel to be confidential and exclusive to the user.

HMAC Hardware Tokens. Some hardware tokens such as YubiKeys use HMAC-SHA-1 evaluation for authentication. This is equivalent to HOTP, except that the input to the witness generation algorithm is an arbitrary challenge rather than a counter, and the output is the full-length HMAC tag.

3.2.2 MFKDF

The core idea of MFKDF is to combine MFA with PBKDF. That is, the goal is to use some combination of authentication factors to derive a static symmetric key. The MFKDF paradigm consists of two parts: (1) a factor construction part, which takes the factor material from an authentication factor and turns it into *source key material*, and (2) the KDF part, in which the source key material from multiple authentication factors is turned into a key. See Figure 3.1 for an illustration.

In the following, we briefly describe the syntax of a factor construction, and the constructions proposed by NS for some dynamic factors. Next, we review the syntax of a key derivation function and describe how factor constructions are combined to build MFKDF, describing the threshold setting in Section 3.2.2.2 and the n-out-of-n derivation setting in Section 3.2.2.3.

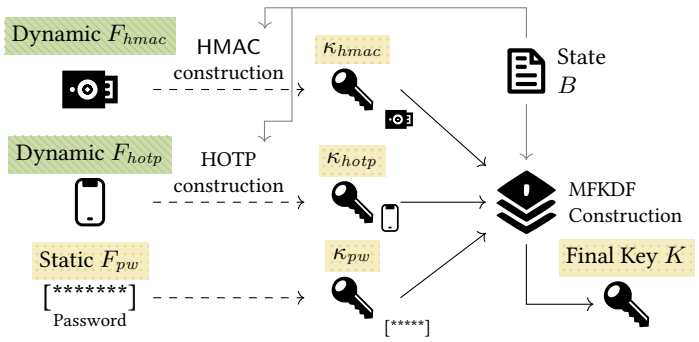


Figure 3.1: MFKDF combining authentication factors (F_{hotp} and F_{hmac}) with a password to derive a key. Dynamic factors are colored green and striped, static ones yellow and dotted.

3.2.2.1 *Factor Constructions*

A factor construction for a factor F consists of algorithms Setup and Derive. Via $(\kappa_F, \beta_0) \leftarrow \mathsf{Setup}(\sigma)$, the randomized setup algorithm on input the factor material σ "constructs" the static source key material κ_F and initial state β_0.[6] Then, the randomized and stateful algorithm Derive, on input a witness W_i for F and (public) state β_i, enables continued evaluation by reconstructing the key material κ_F and producing an updated state β_{i+1}. That is, $(\kappa_F, \beta_{i+1}) \leftarrow \mathsf{Derive}(W_i, \beta_i)$.

For dynamic factors, the witnesses change over time, making it nontrivial to derive the same static κ_F from every witness. NS solve this by storing the secret factor material σ in β, thereby allowing future witnesses to be computed within algorithm Derive and the changes offset by updates to the state. To protect σ, it is encrypted under the MFKDF-derived key K. This establishes a circular "feedback mechanism" where K is first derived from all factors, and then used to encrypt the factor materials. In order for the feedback mechanism to work, the setup and derive algorithms must be run in parallel for all factors used in MFKDF, and intermediate outputs from each factor construction influence the computations of others. To model this, we deviate from the syntax presented in [220] and split algorithms Setup and Derive into two stages each: Setup_1, Setup_2, Derive_1, and Derive_2. Stage one reconstructs κ_F, which is combined with other factors' key material to derive K. Stage two uses K and κ_F to create β_{i+1} for the next iteration. We thus define a factor construction as follows.

[6] Note that in [220], NS refer to the source key material κ_F as "factor material" and the public parameters β as "factor parameters" (and denote them σ and α, respectively). However, not all factor constructions use the factor material σ as the source key material κ_F (for example, the HOTP construction uses a fresh static secret as κ_F, not the HMAC key $\sigma = k_h$). Hence, we choose to separate the notation to avoid confusion.

Definition 1 (Factor Construction)**.** A factor construction for factor F with factor material σ is a tuple of algorithms

$$\begin{aligned} \kappa_F &\leftarrow \mathsf{Setup}_1(\sigma), \\ \beta_0 &\leftarrow \mathsf{Setup}_2(K, \sigma, \kappa_F), \\ \kappa_F &\leftarrow \mathsf{Derive}_1(W_i, \beta_i), \text{ and} \\ \beta_{i+1} &\leftarrow \mathsf{Derive}_2(K, \kappa_F, \beta_i), \end{aligned}$$

where K is the final key output by MFKDF (see Section 3.2.2.2).

For factors that do not employ the feedback mechanism, we will simply refer to the combined algorithms $(\kappa_F, \beta_0) \leftarrow \mathsf{Setup}(\sigma)$ and $(\kappa_F, \beta_{i+1}) \leftarrow \mathsf{Derive}(W_i, \beta_i)$.

Algorithm 2 HOTP Factor Construction for HOTP key k_h

procedure $\mathsf{Setup}_1(k_h)$
  $\kappa_{hotp} \leftarrow\!\!\$\ [0, 10^d)$
  return κ_{hotp}
procedure $\mathsf{Setup}_2(K, k_h, \kappa_{hotp})$
  $W_0 \leftarrow \mathsf{HOTP}(k_h, 0)$
  $c_{\kappa,0} \leftarrow (\kappa_{hotp} - W_0) \bmod 10^d$
  $c_\sigma \leftarrow \mathsf{SK.Enc}(K, k_h)$
  return $(0, c_{\kappa,0}, c_\sigma)$
procedure $\mathsf{Derive}_1(W_i, \beta_i)$
  $(i, c_{\kappa,i}, c_\sigma) \leftarrow \beta_i$
  $\kappa_{hotp} \leftarrow (c_{\kappa,i} + W_i) \bmod 10^d$
  return κ_{hotp}
procedure $\mathsf{Derive}_2(K, \kappa_{hotp}, \beta_i)$
  $(i, c_{\kappa,i}, c_\sigma) \leftarrow \beta_i$
  $k_h \leftarrow \mathsf{SK.Dec}(K, c_\sigma)$
  $W_{i+1} \leftarrow \mathsf{HOTP}(k_h, i+1)$
  $c_{\kappa,i+1} \leftarrow (\kappa_{hotp} - W_{i+1}) \bmod 10^d$
  return $(i+1, c_{\kappa,i+1}, c_\sigma)$

In the following paragraphs, we review the factor constructions proposed by NS in [220] relevant to our attacks.

HOTP Factor Construction. Recall that for an HOTP factor F_{hotp}, the private factor material σ is the HMAC key k_h, and the witness is a one-time password $W_i = \mathsf{HOTP}(k_h, i)$, where i is a counter. In the factor construction for MFKDF, shown in Algorithm 2, the source key material κ_{hotp} consists of a d-digit integer (called the "target" in [220]), which is sampled uniformly at random from $[0, 10^d)$.[7] The source key material κ_{hotp} is encrypted in a stream-cipher fashion by subtracting modulo 10^d the next one-time password W_{i+1}, generating a ciphertext $c_{\kappa,i+1}$ which is stored in the factor state β_{i+1}. This allows κ_{hotp} to be (re-)derived by adding the witness back to $c_{\kappa,i+1}$.

[7] In practice, $d = 6$ is commonly used for HOTP tokens.

To access the next witness W_{i+1} used to encrypt κ_{hotp}, NS use the feedback mechanism described above: the HMAC key k_h is stored encrypted in the state using the final MFKDF key K. NS do not further specify the encryption scheme used in their HOTP (and TOTP) factor construction. In particular, they do not demand that the scheme should be randomized, nor do they define the expected security properties. As we will see later, this under-specification makes the HOTP (and TOTP) factor construction of MFKDF vulnerable to attacks.

This feedback mechanism adds an element of circular security to the factor construction: the source key material κ_{hotp} is encrypted with the witnesses derived from k_h, and k_h itself is encrypted under the key K derived from the source key material. This is true for all factor constructions proposed by NS that make use of the feedback mechanism. A

Algorithm 3 HMAC Factor Construction for HMAC key k_h

procedure Setup(k_h)
  $\kappa_{hmac} \leftarrow k_h$; $c_0 \leftarrow\!\!\$\ \{0,1\}^{160}$
  $W_0 \leftarrow$ HMAC-SHA-1(κ_{hmac}, c_0)
  return $(\kappa_{hmac}, (c_0, \kappa_{hmac} \oplus W_0))$
procedure Derive(W_i, β_i)
  $(c_i, c_{\kappa,i}) \leftarrow \beta_i$
  $\kappa_{hmac} \leftarrow c_{\kappa,i} \oplus W_i$
  $c_{i+1} \leftarrow\!\!\$\ \{0,1\}^{160}$
  $W_{i+1} \leftarrow$ HMAC-SHA-1(κ_{hmac}, c_{i+1})
  return $(\kappa_{hmac}, (c_{i+1}, \kappa_{hmac} \oplus W_{i+1}))$

formal proof of security of MFKDF would hence be limited to what can be achieved in this setting of key-dependent messages [60].

TOTP Factor Construction. The TOTP factor construction is analogous to the HOTP construction, except that the ciphertext $c_{\kappa,i}$ is replaced by a vector $\overline{c_{\kappa,i}}$ which contains the encryption of κ_{hotp} under all OTP witnesses for a window of future times, such that decryption is possible using any witness generated during a time in the window. For details, see [220, Appendix A].

HMAC Factor Construction. This is similar in spirit to the HOTP construction, but this time the HMAC key k_h is directly used as the source key material κ_{hmac}. Challenges are random 160-bit values, and the witnesses W_i from the authenticator are used as one-time pads to encrypt κ_{hmac}. Algorithm 3 shows the HMAC factor construction used in MFKDF to enable support for hardware authenticators, such as YubiKeys, via the HMAC-SHA-1 challenge-response mechanism. In contrast to the usual authentication protocol of an HMAC hardware token, where the authenticator is used to compute the response to a freshly sampled challenge, here the next challenge c_{i+1} is derived ahead of time and stored in the public state β_{i+1}. The state also contains the XOR of the HMAC key k_h with the next witness W_{i+1} computed as response to c_{i+1}. When the client authenticates, it uses the hardware authenticator to re-derive the witness W_{i+1} in response to the stored challenge and can hence recover κ_{hmac}.

OOB Factor Construction. Recall that OOB authentication factors rely on a secure channel between the verifier and the prover, such that the verifier can send a one-time password (OTP) to the prover when they want to authenticate. For MFKDF, NS assume a particular instantiation of this channel, based on S/MIME: the users owns a key pair (sk, pk) for an associated public-key encryption scheme (PK.Enc, PK.Dec). The key material κ_{oob} is sampled uniformly at random in $[0, 10^6)$, and encrypted in a stream-cipher fashion (like in the HOTP construction) with a random OTP $\in [0, 10^6)$ generated by the user. Finally, the user encrypts the OTP to themselves and sends it to the server for storage. At factor derivation time, the user can fetch the OTP and decrypt it. Algorithm 4 shows the

Algorithm 4 OOB Factor Construction

Require: Let $(\mathsf{PK.Enc}, \mathsf{PK.Dec})$ be a public-key encryption scheme with associated keys (pk, sk).

procedure Setup()
  $\kappa_{oob} \leftarrow\!\!\$\ [0, 10^6)$
  $W_0 \leftarrow\!\!\$\ [0, 10^6)$
  $c_{\kappa,0} \leftarrow (\kappa_{oob} - W_0) \bmod 10^6$
  $c_{W,0} \leftarrow \mathsf{PK.Enc}(pk, W_0)$
  return $(\kappa_{oob}, (pk, c_{\kappa,0}, c_{W,0}))$
procedure Derive(W_i, β_i) ▷ The user has derived W_i using their sk by $W_i \leftarrow \mathsf{PK.Dec}(sk, c_{W,i})$.
  $(pk, c_{\kappa,i}, c_{W,i}) \leftarrow \beta_i$
  $\kappa_{oob} \leftarrow (c_{\kappa,i} + W_i) \bmod 10^6$
  $W_{i+1} \leftarrow\!\!\$\ [0, 10^6)$
  $c_{\kappa,i+1} \leftarrow (\kappa_{oob} - W_{i+1}) \bmod 10^6$
  $c_{W,i+1} \leftarrow \mathsf{PK.Enc}(pk, W_{i+1})$
  return $(\kappa_{oob}, (pk, c_{\kappa,i+1}, c_{W,i+1}))$

setup and derive procedures for the OOB factor construction proposed by Nair and Song. Here, the witness is the OTP from the OOB authentication factor, but generated by the client rather than the server. Note that the decryption of the OTP witness W_i from the outsourced ciphertext $c_{W,i}$ is part of the normal authentication protocol for the OOB authentication factor, and hence assumed to happen outside of algorithm Derive.

3.2.2.2 Combining Factors - Threshold

Here we review how MFKDF uses the source key material from the factor constructions to derive a key. NS present two variants of MFKDF: a plain one, where all the factors are necessary to derive the final MFKDF key K, and a threshold variant, where K can be derived using only t-out-of-n factors. We focus here on the threshold construction in order to provide the necessary background for Attacks 3.3.2.2 and 3.3.3.4. The details of the plain construction are given in Section 3.2.2.3.

The threshold MFKDF presented in [220, App. A] is built from a PBKDF and a Shamir secret sharing scheme [265]. We review the syntax of both as used in MFKDF.

Definition 2 (Password-Based Key Derivation Function)**.** A key derivation function $\mathsf{PBKDF}(\kappa, salt, \ell) \mapsto K$ takes as input source key material κ, a salt $salt \in \{0,1\}^*$ and a length $\ell \in \mathbb{N}$ to produce a key $K \in \{0,1\}^\ell$.

Definition 3 (Shamir Secret Sharing)**.** A Shamir secret sharing scheme consists of algorithms

$$\begin{aligned}(\mathrm{s}_1, \ldots, \mathrm{s}_n) &\leftarrow \mathsf{Share}(\kappa, t, n),\\ \kappa &\leftarrow \mathsf{Comb}((\mathrm{s}_{u_1}, \ldots, \mathrm{s}_{u_t}), t, n), \text{ and}\\ \mathrm{s}_j &\leftarrow \mathsf{Rec}(\kappa, t, n, j).\end{aligned}$$

Share splits secret κ into n shares, out of which t suffice to reconstruct the secret through algorithm Comb. We refer to [265] for the correctness and security properties. Algorithm Rec is introduced by Nair and Song, who do not describe its properties explicitly. From its use, we deduce that given the secret κ and the sharing parameters, it recovers the j^{th} share.

Similarly to factor constructions, both the plain and threshold MFKDF schemes define algorithms Setup and Derive. Threshold MFKDF additionally defines an algorithm Recover, which is used in case a factor is lost or needs to be replaced.

Definition 4 (Threshold MFKDF Construction)**.** Let $F_1, F_2, \ldots, F_n$ be authentication factors and let $\overline{\sigma} = (\sigma_{F_1}, \sigma_{F_2}, \ldots, \sigma_{F_n})$ be a vector with the corresponding factor materials. The t-out-of-n threshold MFKDF construction for factors $F_1, F_2, \ldots, F_n$ consists of algorithms

$$\begin{aligned}(K, B_0) &\leftarrow \mathsf{Setup}(\overline{\sigma}, t, \ell),\\ (K, B_{i+1}) &\leftarrow \mathsf{Derive}(\overline{W}_i, B_i), \text{ and}\\ B_{i+1} &\leftarrow \mathsf{Recover}(\overline{W}_i, B_i, x, \sigma_{F'_x}).\end{aligned}$$

Algorithm Setup takes as input $\overline{\sigma}$, the threshold parameter t and $\ell \in \mathbb{N}$. It returns the derived key $K \in \{0,1\}^\ell$ and the initial public state B_0. Derive takes as input a vector of witnesses for a subset of t factors $\overline{W}_i = (W_{i,u_1}, \ldots, W_{i,u_t})$ and the i^{th} public state B_i (containing the state of all n factors). It returns the key K and the updated state. Algorithm Recover takes as input $\overline{W}_i$, the public state B_i, the index x of the factor F_x to be replaced, and the new factor material $\sigma_{F'_x}$ of factor F'_x replacing it.

Here, we recall a high-level overview from [220], with the full details of Setup, Derive, and Recover in Algorithm 5. Algorithms (Share, Comb, Rec) are part of a Shamir secret sharing scheme (see Definition 3).

$\underline{\mathsf{Setup}((\sigma_{F_1}, \ldots, \sigma_{F_n}), t, \ell)}$

1. Sample κ and $salt$ randomly from $\{0,1\}^\ell$.
2. Split κ into n shares using $(\mathrm{s}_1, \ldots, \mathrm{s}_n) \leftarrow \mathsf{Share}(\kappa, t, n)$.
3. For each factor F_j, compute the key material κ_j using algorithm F_j.Setup associated to the factor construction.
4. Expand each κ_j to pad_j of length ℓ using HKDF-Expand.
5. Encrypt each share s_j with the pad derived from the expanded factor via $c_{\mathrm{s}_j} \leftarrow \mathrm{s}_j \oplus \mathsf{pad}_j$.
6. Run $K \leftarrow \mathsf{PBKDF}(\kappa, salt, \ell)$ to derive $K \in \{0,1\}^\ell$.
7. Compute $\beta_{0,j}$ for each factor (using K for the factors that use the feedback mechanism).
8. Let $B_0 \leftarrow (t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{0,1}, \ldots, \beta_{0,n}))$.
9. Return K and the public state B_0.

$\underline{\mathsf{Derive}((W_{i,u_1}, \ldots, W_{i,u_t}), B_i)}$

1. For each of the t witnesses, derive the source key material κ_{u_j} from witness W_{i,u_j} using algorithm F_{u_j}.Derive.

2. Expand each κ_{u_j} to pad_{u_j} of length ℓ using HKDF-Expand.
3. Decrypt share s_{u_j} via $\mathrm{s}_{u_j} \leftarrow c_{\mathrm{s}_{u_j}} \oplus \mathsf{pad}_{u_j}$.
4. Combine the shares into κ using $\mathsf{Comb}((\mathrm{s}_{u_1}, \ldots, \mathrm{s}_{u_t}), t, n)$.
5. Use PBKDF on κ to produce K.
6. Compute β_{i+1,u_j} for each factor (using K for the factors that use the feedback mechanism) and update B_{i+1}.
7. Return K and the updated public state B_{i+1}.

$\underline{\mathsf{Recover}(\overline{W}_i, B_i, x, \sigma_{F'_x})}$

1. Compute κ as in steps 1–4 of Derive.
2. Set up the new factor F'_x, computing κ'_x and pad'_x as in steps 3 and 4 of Setup.
3. Recover the share of the lost factor via $\mathrm{s}_x \leftarrow \mathsf{Rec}(\kappa, t, n, x)$.
4. Update c_{s_x} to $c'_{\mathrm{s}_x} \leftarrow \mathrm{s}_x \oplus \mathsf{pad}'_x$.
5. Compute the inital state for F'_x and update the state of each of the t factors used in the recovery.
6. Update B_{i+1} with the new states and c_{s_x} and return it.

3.2.2.3 Combining Factors - Plain

Definition 5 (Plain MFKDF Construction)**.** Let $F_1, F_2, \ldots, F_n$ be authentication factors and let $\overline{\sigma} = (\sigma_{F_1}, \sigma_{F_2}, \ldots, \sigma_{F_n})$ be a vector with the corresponding factor materials. The MFKDF construction for factors $F_1, F_2, \ldots, F_n$ consists of algorithms

$$(K, B_0) \leftarrow \mathsf{Setup}(\overline{\sigma}, \ell), \text{ and}$$
$$(K, B_{i+1}) \leftarrow \mathsf{Derive}(\overline{W}_i, B_i).$$

Here ℓ is the desired length of the derived key and $\overline{W}_i = (W_{i,F_1}, \ldots, W_{i,F_n})$ is a vector consisting of the i^{th} witness for each factor in the construction. K is the final key output by MFKDF and B_i is the i^{th} public state.

For each factor F_j, let F_j.Setup and F_j.Derive be the setup and derive algorithm, respectively, associated to F_j by the factor construction. The "plain" (n-out-of-n) MFKDF construction for factors $F_1, F_2, \ldots, F_n$ is given in Algorithm 6. Unlike the Threshold version, it is not based on Shamir sharing, but rather on deriving the final key from a simple concatenation of the factor key materials.

Algorithm 5 t-out-of-n Threshold MFKDF Construction

Require: Let PBKDF be a memory-hard PBKDF, let HKDF.Exp be HKDF-Expand [186], and let (Share, Comb, Rec) be Shamir secret sharing [265].

procedure Setup($(\sigma_{F_1}, \ldots, \sigma_{F_n}), t, \ell$)
  $\kappa \leftarrow\$ \{0,1\}^\ell$; $salt \leftarrow\$ \{0,1\}^\ell$
  $K \leftarrow \mathsf{PBKDF}(\kappa, salt, \ell)$
  $(\mathrm{s}_1, \ldots, \mathrm{s}_n) \leftarrow \mathsf{Share}(\kappa, t, n)$
  for $j = 1$ to n **do**
    $\kappa_j \leftarrow F_j.\mathsf{Setup}_1(\sigma_{F_j})$ ▷ If F_j uses feedback
    $(\beta_{0,j}, \kappa_j) \leftarrow F_j.\mathsf{Setup}(\sigma_{F_j})$ ▷ Else
    $\mathsf{pad}_j \leftarrow \mathsf{HKDF.Exp}(\kappa_j, \varepsilon, \ell)$ ▷ Expand κ_j to length ℓ with empty context
    $c_{\mathrm{s}_j} \leftarrow \mathsf{pad}_j \oplus \mathrm{s}_j$
  for $j = 1$ to n **do**
    $\beta_{0,j} \leftarrow F_j.\mathsf{Setup}_2(K, \sigma_{F_j}, \kappa_j)$ ▷ If feedback
  $B_0 \leftarrow (t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{0,1}, \ldots, \beta_{0,n}))$
  return (K, B_0)
procedure Derive($\overline{W}_i, B_i$) ▷ Using t factors $(F_{u_1}, \ldots, F_{u_t})$ of the n original factors
  $(W_{i,u_1}, \ldots, W_{i,u_t}) \leftarrow \overline{W}_i$
  $(t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{i,1}, \ldots, \beta_{i,n})) \leftarrow B_i$
  for $j = 1$ to t **do**
    $\kappa_{u_j} \leftarrow F_{u_j}.\mathsf{Derive}_1(W_{i,u_j}, \beta_{i,u_j})$ ▷ If feedback
    $(\kappa_{u_j}, \beta_{i+1,u_j}) \leftarrow F_{u_j}.\mathsf{Derive}(W_{i,u_j}, \beta_{i,u_j})$ ▷ Else
    $\mathsf{pad}_{u_j} \leftarrow \mathsf{HKDF.Exp}(\kappa_{u_j}, \varepsilon, \ell)$
    $\mathrm{s}_{u_j} \leftarrow \mathsf{pad}_{u_j} \oplus c_{\mathrm{s}_{u_j}}$
  $\kappa \leftarrow \mathsf{Comb}((\mathrm{s}_{u_1}, \ldots, \mathrm{s}_{u_t}), t, n)$
  $K \leftarrow \mathsf{PBKDF}(\kappa, salt, \ell)$
  for $j = 1$ to n **do**
    if $j \in \{u_1, \ldots, u_t\}$ **then**
      $\beta_{i+1,j} \leftarrow F_j.\mathsf{Derive}_2(K, \kappa_j, \beta_{i,j})$ ▷ If F_j uses feedback
    else $\beta_{i+1,j} \leftarrow \beta_{i,j}$
  $B_{i+1} \leftarrow (t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{i+1,1}, \ldots, \beta_{i+1,n}))$
  return (K, κ, B_{i+1}) ▷ If called by Recover
  return (K, B_{i+1}) ▷ Else
procedure Recover($\overline{W}_i, B_i, x, \sigma_{F'_x}$) ▷ Recover share x using t factors
  ▷ Recover the secret and update existing factors:
  $(K, \kappa, B_{i+1}) \leftarrow \mathsf{Derive}(\overline{W}_i, B_i)$
  $(t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{i+1,1}, \ldots, \beta_{i+1,n})) \leftarrow B_{i+1}$
  ▷ Replace the lost factor:
  $\kappa'_x \leftarrow F'_x.\mathsf{Setup}_1(\sigma_{F'_x})$ ▷ If F'_x uses feedback
  $\beta_{i+1,x} \leftarrow F'_x.\mathsf{Setup}_2(K, \sigma_{F'_x}, \kappa'_x)$
  $(\beta_{i+1,x}, \kappa'_x) \leftarrow F'_x.\mathsf{Setup}(\sigma_{F'_x})$ ▷ Else
  $\mathsf{pad}'_x \leftarrow \mathsf{HKDF.Exp}(\kappa'_x, \varepsilon, \ell)$
  $\mathrm{s}_x \leftarrow \mathsf{Rec}(\kappa, t, n, x)$
  $c_{\mathrm{s}_x} \leftarrow \mathsf{pad}'_x \oplus \mathrm{s}_x$
  $B_{i+1} \leftarrow (t, \ell, salt, (c_{\mathrm{s}_1}, \ldots, c_{\mathrm{s}_n}), (\beta_{i+1,1}, \ldots, \beta_{i+1,n}))$
  return B_{i+1}

Algorithm 6 Plain MFKDF Construction

Require: Let PBKDF be a memory-hard KDF.

procedure $\mathsf{Setup}((\sigma_{F_1}, \ldots, \sigma_{F_n}), \ell)$
  for $j = 1$ to n **do**
    $\kappa_{F_j} \leftarrow F_j.\mathsf{Setup}_1(\sigma_{F_j})$ ▷ If F_j uses feedback
    $(\kappa_{F_j}, \beta_{0,F_j}) \leftarrow F_j.\mathsf{Setup}(\sigma_{F_j})$ ▷ Else
  $\kappa \leftarrow \kappa_{F_1} \| \kappa_{F_2} \| \ldots \| \kappa_{F_n}$; $salt \leftarrow\!\!\$\ \{0,1\}^{\ell}$
  $K \leftarrow \mathsf{PBKDF}(\kappa, salt, \ell)$
  for $j = 1$ to n **do** ▷ If F_j uses feedback
    $\beta_{0,F_j} \leftarrow F_j.\mathsf{Setup}_2(K, \sigma_{F_j}, \kappa_{F_j})$
  $B_0 \leftarrow (\ell, salt, (\beta_{0,F_1}, \ldots, \beta_{0,F_n}))$
  return (K, B_0)

procedure $\mathsf{Derive}(\overline{W}_i, B_i)$
  $(W_{i,F_1}, \ldots, W_{i,F_n}) \leftarrow \overline{W}_i$
  $(\ell, salt, (\beta_{i,F_1}, \ldots, \beta_{i,F_n})) \leftarrow B_i$
  for $j = 1$ to n **do**
    $\kappa_{F_j} \leftarrow F_j.\mathsf{Derive}_1(W_{i,F_j}, \beta_{i,F_j})$ ▷ If feedback
    $(\kappa_{F_j}, \beta_{i+1,F_j}) \leftarrow F_j.\mathsf{Derive}(W_{i,F_j}, \beta_{i,F_j})$ ▷ Else
  $\kappa \leftarrow \kappa_{F_1} \| \kappa_{F_2} \| \ldots \| \kappa_{F_n}$
  $K \leftarrow \mathsf{PBKDF}(\kappa, salt, \ell)$
  for $j = 1$ to n **do** ▷ If F_j uses feedback
    $\beta_{i+1,F_j} \leftarrow F_j.\mathsf{Derive}_2(K, \kappa_{F_j}, \beta_{i,F_j})$
  $B_{i+1} \leftarrow (\ell, salt, (\beta_{i+1,F_1}, \ldots, \beta_{i+1,F_n}))$
  return (K, B_{i+1})

Table 3.1: Summary of the attacks, listing the type of attack (**fund**amental, state **int**egrity, **impl**ementation-related, or caused by bad assumptions or insufficient detail in the **spec**ification), and for each attack its prerequisites (*read/write* access to the MFKDF state B; *leak* of a witness W_i or the factor material σ; *user actions* such as key derivation and replacing a factor), and its impact (recover the final key K, the source key material κ_F, or violate security theorems) and whether it can be mitigated (from 'no mitigations possible' ○, to 'mitigations unclear' ◑, and 'mitigations provided' ●).

		Read State	Write State	Leak One Factor Witness	Leak Factor Material	User Action	Recover Final Key	Recover Src. Key Material	Break *'Safety'*	Break *'Entropy'*	Break Exp. Security	
Dynamic Factor (3.3.1)	fund	✓	✗	✓	✗	✗	✗	✓	✗	✓	✓[c]	○
OOB Overwriting (3.3.2.1)	int	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓[c]	◑
Parameter Tampering (3.3.2.2)	int	✓	✓	✗	✗	✓	✓	✗	✗	✓	✗	◑
Share Dilution (3.3.2.3)	int, impl	✓	✓	✗	✗	✓	✓[a]	✗	✓	✓	✗	◑
HOTP Compromise (3.3.3.1)	spec, impl	✓	✗	✗	✓	✗	✓	–	✗	✓	✓	◑
HOTP Bias (3.3.3.2)	spec	✓	✗	✗	✗	✗	✗	✓	✗	✓	✓[c]	●
Share Recovery (3.3.3.3)	spec	✓	✗	✓[b]	✗	✓	✗	✓	✓	✓	✓[c]	●
Share Format (3.3.3.4)	spec	✓	✗	✗	✗	✗	✓[a]	✓	✗	✓	✓	●
	Type	Prerequisites					Impact					Mitigations

[a] Depending on the minimum entropy of the remaining factors in the target MFKDF construction. [b] Witness of a replaced factor. No witness needed if the factor has low entropy. [c] As a consequence of recovering the source key material for the target factor.

3.3 ATTACKS

In this section, we present our attacks against the MFKDF construction and its reference implementation (release v1.4.7) [217]. Table 3.1 provides an overview of our attacks, categorized as fundamental (Section 3.3.1), attacking unauthenticated state (Section 3.3.2) or exploiting underspecifications and bad assumptions (Section 3.3.3). It also shows the attack prerequisites, and whether each attack can be mitigated.

No concrete threat model is specified by NS [220]: the authors are mostly concerned with attacks such as "credential stuffing", where an attacker uses a user's leaked password from one account to access their other accounts, in the hope that the user reused the password across multiple accounts. We consider a more extensive threat model, which includes access to the MFKDF state and user actions: the prerequisites allow us to specify the fine-grained adversarial capabilities needed for each of the attacks.

Under the impact column, we highlight the security goals of MFKDF contradicted by our attacks. In particular, all attacks break the "Entropy" goal [220, §3]:

> Attacking the derived key should be as hard as attacking the weakest set of allowed factors.

Some attacks additionally contradict "Safety" [220, §3]:

> Providing an invalid set of factors should be highly unlikely to derive the correct key.

Together, they invalidate the MFKDF security theorem [221, App. B, Thm 4]. The theorem implicitly assumes that the combination of dif-

ferent factors results in each factor contributing to the entropy of the final key without any leakage—our attacks show that this is not the case.

We also falsify the claim of "exponential security" by showing attacks that recover the final key or the source key material of factor constructions:

> [A]ttackers cannot separately guess individual MFKDF factors, and must simultaneously correctly guess all factors to derive a key. [...] Therefore, given n available factors with a mean factor entropy of 2^{-m} bits, the crack time t will be $t \propto 2^m$ for PBKDFs and $t \propto 2^{mn}$ for MFKDF. We thus claim that n-factor MFKDF provides an exponential security improvement over PBKDFs. [220, § 6]

Many of our attacks additionally lead to practically exploitable vulnerabilities in the reference implementation of MFKDF.

3.3.1 Dynamic Factor Attack

The H/TOTP, HMAC and OOB factor constructions convert dynamic authentication factors into static source key material. We show that these constructions inherently degrade the security compared to what is expected of a dynamic factor: namely, that access to an old witness does not compromise the security of the factor. When used for MFKDF, leakage of a single witness to an adversary with access to the public factor state instead compromises the factor permanently. As we discuss in Section 3.5, this represents a fundamental issue with MFKDF that cannot be mitigated.

Threat Model. The attack assumes an adversary with access to an authentication witness for the dynamic factor. The witness does not have to be "fresh", in the sense that it would be valid if the factor was used for authentication. For example, an already used or outdated H/TOTP or OOB code suffices.

This threat model matches the standard expectation that "one-time" codes are ephemeral, in the sense that leakage of an old code does not compromise the security of the factor. It is also formally supported by the threat model for HOTP described in RFC 4226 [197]. As a real-world example, an adversary that captures a one-time code as it is being used (e.g., by looking over the shoulder of the victim as they type it in, or through a key-logger on a public computer) can mount this attack, even after the code has been used or expired.

Additionally, the attack assumes read access to the MFKDF state. Recall that the MFKDF state contains all factor construction states (see Line 8 of the Setup algorithm of Definition 4). As described in [220], this state does not have to be kept confidential, and could even "be stored on a public blockchain" [220, p. 4]. This threat model is further supported by the proposed deployments of MFKDF as an online password manager [220, §10.1] or a decentralized cryptocurrency wallet [220, §10.2], where the state is outsourced by the client and available to the adversary.

Attack Overview. Recall from Figure 3.1 that the source key material κ_F associated to a factor construction F is derived from the i-th witness W_i, and the public factor state β_i via

$$\kappa_F \leftarrow \mathsf{Derive}_1(W_i, \beta_i) \ .$$

This means that an adversary with access to the public state β_i can recover κ_F as soon as a witness is observed, thereby reducing the entropy of the final MFKDF key by the amount provided by factor F. Furthermore, since the adversary can record all past public states $\beta_0, \ldots, \beta_i$, κ_F can be compromised as soon as any previous witness is leaked. As stated above, this removes the security guarantee of dynamic factors which is normally provided by the ephemerality of the witnesses.

Impact. The consequence of this attack is that dynamic factors are no more secure than static factors with the same entropy. For instance, a 6 digit HOTP code from a software authenticator is no stronger than a six digit PIN which the user could store in a password manager or memorize. More troublesome, the gap between the expected security of one-time codes (that is, that exposure of old codes is harmless) and their usage in MFKDF likely leads to a false sense of security for users, who, unaware of the implications for MFKDF, might not protect their old one-time codes as they would a PIN or a password, thereby inadvertently risking the security of the source key material. In conclusion: in MFKDF, one-time codes are not one-time.

3.3.2 State Integrity Attacks

Beside the inherent issues with using dynamic authentication factors, the public state B of MFKDF provides a large attack surface. This state is accessed and updated in both the plain and threshold constructions (see algorithm Derive in Definition 4 and 5). In [220], NS do not consider whether the state needs to be authenticated or protected in any way. We show that the lack of integrity of the state leads to practical attacks against MFKDF. In Section 3.4, we argue that mitigations to this problem are non-trivial, because of its circularity: the integrity of the state should be verified prior to its usage, but before using the state, an MFKDF user has no key that can be used for this verification.

The attacks presented here assume an adversary with read and write access to the MFKDF state. This is the standard threat model for end-to-end encrypted systems where users outsource the state to the server. For instance, if MFKDF is used in an online password manager with an untrusted server, as proposed by NS, the service provider or an adversary controlling the server could perform these attacks.

3.3.2.1 OOB Overwriting Attack

This attack exploits the fact that users fetch their S/MIME public key from the unauthenticated MFKDF state. Due to the lack of integrity of the outsourced key, an attacker with access to the public state can

overwrite the user's public key pk with a malicious key, such that the user encrypts their next OOB witness to a public key controlled by the adversary. This allows the attacker to recover the source key material κ_{oob}, since it is stored in the public state "encrypted" under the OOB witness.

Attack Overview. Let $\beta_i = (pk, c_{\kappa,i}, c_{W,i})$ be the (honest) OOB factor state after the i-th use of the OOB factor, consisting of the user's public key pk, the encrypted key material $c_{\kappa,i}$ and the encryption of the current one-time code $c_{W,i}$.

The attacker overwrites the state with $\beta_i^* = (pk^*, c_{\kappa,i}, c_{W,i})$, where pk^* is the public key of an S/MIME key pair for which the adversary knows the corresponding private key sk^*. Upon the next use of the OOB factor, the user fetches β_i^* from the state and runs:

$\underline{\mathsf{Derive}(W_i, \beta_i^*)}$
$(pk^*, c_{\kappa,i}, c_{W,i}) \leftarrow \beta_i^*$
$\kappa_{oob} \leftarrow (c_{\kappa,i} + W_i) \bmod 10^6$
$W_{i+1} \leftarrow\!\!\$\ [0, 10^6)$
$c_{\kappa,i+1} \leftarrow (\kappa_{oob} - W_{i+1}) \bmod 10^6$
$c^*_{W,i+1} \leftarrow \mathsf{PK.Enc}(pk^*, W_{i+1})$
Return $(\kappa_{oob}, (pk^*, c_{\kappa,i+1}, c^*_{W,i+1}))$

In particular, the key pk^* of the adversary is used to encrypt the next OOB code W_{i+1}. Since the attacker has access to the updated public state $\beta_{i+1} = (pk^*, c_{\kappa,i+1}, c^*_{W,i+1})$, it can recover the code as $W_{i+1} \leftarrow \mathsf{PK.Dec}(sk^*, c^*_{W,i+1})$. After this, it is trivial to recover the source key material as $\kappa_{oob} \leftarrow (c_{\kappa,i+1} + W_{i+1}) \bmod 10^6$.

Impact. A malicious server can use this attack to compromise the OOB source key material as soon as the OOB factor is used, removing the entropy from this factor from the MFKDF key. This compromise is invisible to the user: after obtaining the source key material, the adversary can overwrite β^*_{i+1}, either with the previous honest state β_i, or by replacing $c^*_{W,i+1}$ with $c_{W,i+1} \leftarrow \mathsf{PK.Enc}(pk, W_{i+1})$, ensuring that the user can successfully decrypt the witness ciphertext with their S/MIME private key the next time the OOB factor is used.

3.3.2.2 Parameter Tampering Attacks

The public MFKDF state B includes information on the key derivation function and its parameters, such as the number of iterations for PBKDF and the length of the derived key. Tampering with this information will, in most cases, result in deriving a bogus key. Nonetheless, the resulting key could be used to infer information about the factors, or can be easy to guess. The attacks we present illustrate the danger of storing this "KDF metadata" unauthenticated in the MFKDF state, rather than having the MFKDF client use a fixed KDF.

For this attack, we additionally assume that the MFKDF key is used to, for example, encrypt data with an AEAD, or to authenticate to a service,

and that the adversary can see the resulting ciphertexts or authentication messages. If MFKDF is used to derive a key for a password manager or outsourced file storage, an attacker in the malicious or compromised service provider threat model can perform this attack. Similarly, if MFKDF is used for authentication with the ISO/IEC 9798-2 Unilateral Authentication protocol, as proposed by NS[221, §7], the verifier can perform this attack.

1: Short Key Attack. Consider MFKDF with key length ℓ chosen by the user. (The default length is 32 bytes.) The adversary can overwrite ℓ, picking a new arbitrary length: let's assume 4 bytes. If this key is used to encrypt user data or to produce authentication messages, the adversary can easily guess the 4-byte key by brute-force.

This trivially violates the confidentiality of the encrypted data, but also has further consequences. Note that some of the PBKDFs proposed for use in MFKDF (such as PBKDF2) are actually Extendable Output Functions (XOFs), meaning that the 4-byte derived key will be a prefix of the original key K. The adversary would thus learn the first four bytes of K. The attack can then continue iteratively: the adversary sets the key length to 8 bytes, and guesses the remaining unknown 4 bytes. This process can be iterated until the full key is recovered.

2: Weak PBKDF Attack. The recommended choice of PBKDF for MFKDF is Argon2, a memory-hard PBKDF. The use of Argon2 slows down a brute-force attack on the MFKDF factors, by limiting the effectiveness of parallelization [56, 234].

However, since the choice of PBKDF is stored in the public state B, an adversary can overwrite B, replacing Argon2 with a weak KDF such as, e.g., PBKDF2 with a single iteration of HMAC. When MFKDF is evaluated, a different key $K' \neq K$ will be derived from the same factors. If the adversary observes cryptographic operations where K' is used, like encryption or authentication, it can mount a brute-force attack to guess the source key material of the factors used to derive K', with a high degree of parallelism and fast computation for each guess. A successful attack would expose the source key material, enabling the adversary to recover the original key K. Hence, the benefits of using a memory-hard PBKDF are void against an attacker with write access to the state.

Impact. The attacks significantly reduce the computation cost of brute-force attacks on MFKDF. For the weak PBKDF attack, an analysis by Percival [234] estimates the cost of guessing a password with 56 bits entropy to $10M for PBKDF2, compared to $210B for a memory-hard KDF. For the short key attack, recovering a 32 byte key would require only $8 \cdot 2^{32}$ guesses when iterating the attack 4 bytes at a time, compared to 2^{256} guesses for a brute-force search.

3.3.2.3 Share Dilution Attack

The t-out-of-n threshold MFKDF construction (Section 3.2.2.2) uses Shamir secret sharing to allow the MFKDF key to be derived from any subset

of at least t out of the n factors. Like for the parameter tampering attacks, the threshold value t is stored in the unauthenticated MFKDF state. This attack effectively removes factors from MFKDF, exploiting the lack of integrity of the state, together with the fact that the concrete implementation of algorithm Recover [217] in the reference implementation deviates from what NS describe in [220], and that we report in Algorithm 5.

Attack Overview. Instead of recovering the share of the lost factor with Rec, as specified in [220] (see Algorithm 5), the implementation of algorithm Recover runs $\mathsf{Share}(\kappa, t, n)$ to generate fresh shares for all factors, where t is fetched from the MFKDF state B.[8] This is equivalent to running the share setup algorithm again with the same secret. Since B is not integrity-protected, an adversary can overwrite t with an arbitrary value $t' < t$. This change has no effect until the user runs algorithm Recover (e.g., to change their password). At that point, the adversary-chosen threshold t' is used instead of t as input to Share, with the result that t' of the new shares suffice to derive the MFKDF key.

[8] See src/classes/MFKDFDerivedKey/reconstitution.js, line 365

Impact. This attack impacts the security of the t-out-of-n threshold MFKDF construction by reducing the threshold needed to compute the final key, hence "diluting" the entropy provided by multiple factors. As an example, a 3-out-of-4 construction with HOTP, HMAC, password and OOB as factors becomes trivially broken when t' is set to 2: an exhaustive key search for the ~40 bits of entropy provided by the OOB and HOTP factors would take seconds on a laptop.

Limitations. The MFKDF reference implementation is only vulnerable to our attack if the attacker sets $t' > 1$, as Shamir secret sharing is not used for thresholds of 1.

3.3.3 *Specification Attacks*

Underspecification in cryptographic systems can lead to attacks, as showcased by for example JWT [267]. We now turn to attacks on MFKDF which fall into this category. They either stem from a lack of rigor in the specification of cryptographic primitives, or from incorrect assumptions on the security of these primitives. Most of these attacks are also reflected in the reference implementation of MFKDF itself. The attacks in this section all assume read-only access to the public MFKDF state.

3.3.3.1 *HOTP Compromise Attack*

This attack allows an adversary who already compromised the H/TOTP factor material (i.e. the HMAC key)[9] to recover the MFKDF key if H/TOTP factor constructions are used, bypassing other factors entirely. For both HOTP and TOTP, the HMAC key is stored in encrypted form in the public factor state. This encryption is performed using a "symmetric encryption scheme", but no security requirements of the scheme are specified. In their concrete implementation of the H/TOTP factor constructions, Nair and Song use the one-time pad (i.e. XOR) as the symmetric encryption scheme.

[9] This HMAC key can be commonly extracted from authenticator apps for smartphones by using the backup functionality [147, 31, 41].

Attack Overview. In the reference implementation of MFKDF, the feedback encryption mechanism using the final MFKDF key K creates the factor material ciphertext on line 7 of Algorithm 2 as $c_\sigma \leftarrow \mathsf{SK.Enc}(K, k_h)$, where

$$\mathsf{SK.Enc}(K, k_h) := K \oplus k_h \ .$$

The ciphertext c_σ is part of β_i, contained in the public MFKDF state. Hence, an attacker that compromises the factor material k_h of a *single* HOTP or TOTP factor can recover the final MFKDF key as $K \leftarrow c_\sigma \oplus k_h$.

Impact. This attack violates the MFKDF security guarantee that the derived key can only be recovered after *all* factors have been compromised. While the leakage of the HMAC key is a strong prerequisite, the consequence is that this attack completely voids the security of MFKDF in the presence of a *single* compromised factor.

3.3.3.2 HOTP Bias Attack

This attack exploits a small bias in HOTP witnesses to recover the source key material κ_{hotp} from sufficient observations of the public state in the HOTP factor construction.

The reduction of the 31-bit integer s_{int} modulo 10^d on Line 5 of HOTP in Algorithm 1 adds a small bias to the output, favouring values closer to zero. While this bias is too small to be an issue for authentication [197, A.4.1], it can be exploited for MFKDF because NS use the HOTP witnesses to protect the confidentiality of a static secret. This connects different samples (despite independent witnesses) and allows an adversary to exploit the small bias in each observation to accumulate information about the hidden secret.

Attack Overview. The HOTP factor construction uses a d-digit long witness to hide the static source key material $\kappa_{hotp} \in [0, 10^d)$: it computes the next witness W_{i+1} and encrypts κ_{hotp} via $c_{\kappa,i+1} = \kappa_{hotp} - W_{i+1} \bmod 10^d$ (line 17 of Algorithm 2). The resulting ciphertext $c_{\kappa,i+1}$ is called the "offset" in [220]. The adversary may observe many such offsets, since they are part of the public state.

Contrary to the assumption of NS, the HOTP values specified in RFC 4226 [197] are not uniformly random in $[0, 10^d)$: they have a slight bias towards smaller values. Algorithm 1 shows that HOTP produces the witness by generating 31 random bits,[10] interpreting them as an integer, and taking the result modulo 10^d to get a d-digit witness. This results in a bias, since the largest number, $2^{31} - 1$, is not divisible by 10^d. Hence, for $\gamma = (2^{31} - 1) \bmod 10^d$, values in $[0, \gamma]$ are slightly more likely to be output than values in $(\gamma, 10^d)$. While the HOTP factor construction would be provably secure with truly uniformly random witnesses (as alleged by NS in the security theorem for the H/TOTP factor construction [220, §5.2.2]), the actual, biased, witnesses can be exploited to recover κ_{hotp}.

As above, let γ denote the largest witness which occurs more frequently than average. Figure 3.2 illustrates the bias for a simplified example with 1-digit ($d = 1$) HOTP values.[11] Since the offsets $c_{\kappa,i} =$

[10] The bits are selected from the HMAC digest. However, for our purpose and due to the PRF security of HMAC, we can consider the bits to be random.

[11] Our attack targets full-sized HOTP codes of at least 6 digits, as specified in RFC 4226 [197]. Figure 3.2 is only an illustrative example.

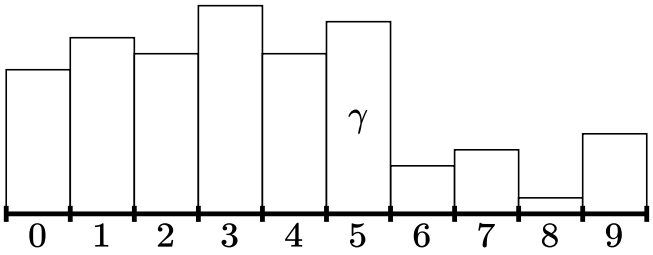


Figure 3.2: Bias of 1-digit HOTP witnesses for $n = 10$ bins.

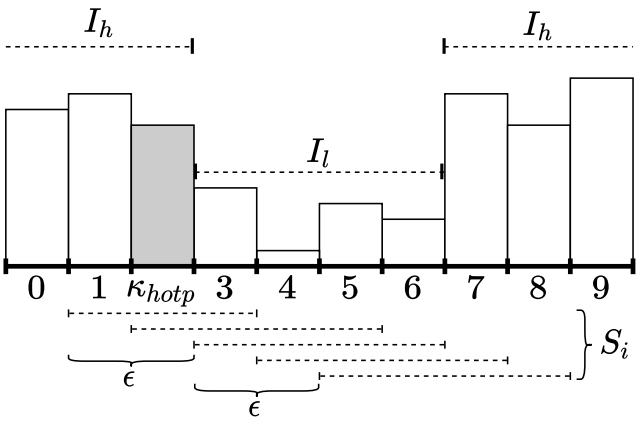


Figure 3.3: Bias of offsets, shifting the HOTP witness bias depending on the hidden secret κ_{hotp}.

$\kappa_{hotp} - W_i \bmod 10$ are generated by subtracting the biased factor witnesses W_i from the secret, offsets in the interval $[\kappa_{hotp} - \gamma, \kappa_{hotp}]$ (modulo 10^d) will appear more frequently than the rest. Hence, κ_{hotp} can be recovered by finding the upper endpoint of this interval.

Figure 3.3 shows a possible distribution of the biased offset for the static secret $\kappa_{hotp} = 2$. By I_h we denote the interval of more frequent offsets, and I_l the less frequent. In the previous example, we have $I_l = [3, 6]$ and $I_h = [0, 2] \cup [7, 9]$.

Given m offsets $c_{\kappa,i}$, an attack strategy to locate interval I_h is to use the maximum likelihood estimator (MLE). We aim to identify the consecutive interval (with wraparound at 10^d) that has the maximal sum of per-bin counts (i.e. the largest number of offsets). The secret κ_{hotp} is the last value of I_h before the less frequent offsets.

Concretely, fix a candidate secret t and recall that I_h consists of the γ values ending at t. Each offset independently falls inside this interval with probability p_h and outside it with probability $p_l < p_h$. Writing $S(t)$ for the number of the m observed offsets that fall in the interval, the likelihood and log-likelihood are

$$\begin{aligned} L(t) &= p_h^{S(t)}\, p_l^{m-S(t)}\,, \\ \log L(t) &= S(t) \log p_h + (m - S(t)) \log p_l \\ &= m \log p_l + S(t)\,(\log p_h - \log p_l)\,. \end{aligned}$$

Since $p_h > p_l$, this is increasing in $S(t)$, so maximising the likelihood amounts exactly to selecting the interval with the most offsets (breaking ties arbitrarily). The only assumption is that the individual offsets $c_{\kappa,i}$

are independent given κ_{hotp} – the witnesses are evaluated at distinct counters and are independent by PRF security of HMAC.

We generalize the problem of finding the exact end point of I_h to finding a value that is at most ϵ values away from the correct secret κ_{hotp}, as the adversary may have external means to test multiple guesses for κ_{hotp}. For instance, it may be feasible to try some number of HOTP values for every password guess, and perform trial decryption with the resulting MFKDF key to verify the correctness of the guess.

It is instructive to model MFKDF offsets as a "balls into bins" problem with biased probabilities, where samples correspond to balls and integers in $[0, 10^d)$ to bins, to gain a more rigorous understanding of our attack. As discussed below, we can express the number of balls in an interval of bins as the sum of Bernoulli random variables and use Chernoff bounds to estimate the worst case number of samples needed to get within ϵ distance of κ_{hotp}.

Formal Analysis. There are $n = 10^d$ bins (one for every possible offset) and m balls (corresponding to $c_{\kappa,i}$). The interval I_h of more likely offsets has $\gamma = 2^{31} \bmod 10^d$ elements (e.g., in Figure 3.3, $\gamma = 6$), and $|I_l| = 10^d - \gamma$. We now compute the required number of balls to recover κ_{hotp} with good probability.

Let X_i^j be a Bernoulli random variable that is 1 if ball i falls into bin j and 0 otherwise. The biased probabilities p_h for likely bins, respectively p_l for unlikely bins, are as follows:

$$p_h := \Pr[X_i^j = 1 | j \in I_h] = \frac{\lfloor 2^{31}/10^d \rfloor + 1}{2^{31}}$$
$$p_l := \Pr[X_i^j = 1 | j \in I_l] = \frac{\lfloor 2^{31}/10^d \rfloor}{2^{31}}$$

Let $B_j = \sum_1^m X_i^j$ be the random variable counting the number of balls in bin j. Since $|I_h| = \gamma$, we define a random variable for the sum of γ consecutive bins as follows, ending with bin i:

$$S_i = \sum_{j=0}^{\gamma-1} B_{i-j \bmod 10^d}$$

Our attack is successful if the interval I_h (which ends with the bin κ_{hotp}) has the largest overall sum, i.e., $\Pr[S_{\kappa_{hotp}} > \max_{j \neq \kappa_{hotp}} S_j]$.

In the generalized version of the problem, we consider the attack to be successful if any bin with distance at most ϵ from κ_{hotp} has the maximal sum for $i, j \in [0, 10^d)$:

$$\Pr\left[\max_{|i-\kappa_{hotp}|\leq\epsilon} S_i > \max_{|j-\kappa_{hotp}|>\epsilon} S_j\right]$$
$$= \bigvee_{|i-\kappa_{hotp}|\leq\epsilon} \bigwedge_{|j-\kappa_{hotp}|>\epsilon} \Pr[S_i > S_j] \tag{3.1}$$

We will now compute $\Pr[S_i > S_j]$ for some i, j satisfying $|i - \kappa_{hotp}| \leq \epsilon$ and $|j - \kappa_{hotp}| > \epsilon$, by applying two Chernoff bounds for

independent Poisson trials for some $0 < \delta_1, \delta_2 < 1$:

$$\Pr[S_i > (1-\delta_1)\mu_i] \geq 1 - e^{-\delta_1^2 \cdot \mu_i/2}$$
$$\Pr[S_j < (1+\delta_2)\mu_j] \geq 1 - e^{-\delta_2^2 \cdot \mu_j/3}$$

where μ_k denotes the expected value of S_k for $k \in \{i, j\}$, computed explicitly below. If we carefully pick δ_1, δ_2 such that $(1-\delta_1)\mu_i > (1+\delta_2)\mu_j$, then we can combine the above probabilities to estimate $\Pr[S_i > S_j]$. Although S_i and S_j are clearly not independent, we find that the empirically measured attack success reasonably matches the values computed here.

Before we can pick δ_1, δ_2, we compute the mean μ_k for some $k \in [0, 10^d)$ with the straightforward computation:

$$\mu_k = \mathbb{E}[S_k] = \sum_{j=0}^{\gamma-1} \sum_{i=1}^{m} \mathbb{E}[X^i_{k-j \bmod 10^d}] = m \cdot ((\gamma - \alpha_k)p_h + \alpha_k p_l)$$

for $\alpha_k = \min(|k - \kappa_{hotp}|, \min(n-\gamma, \gamma))$, the number of unlikely bins among the ones that S_k sums over (which can be at most the total number of unlikely bins $n - \gamma$, and not more than all bins of S_k).

We derive the following limits for δ_1, δ_2:

$$0 < \delta_1 < 1 - (1+\delta_2)\frac{\mu_j}{\mu_i}$$
$$0 < \delta_2 < \frac{\mu_i}{\mu_j} - 1$$

For the optimal bounds, we would set the values δ_1, δ_2 such that they maximise $\Pr[S_i > (1-\delta_1)\mu_i] \cdot \Pr[S_j < (1+\delta_2)\mu_j]$.[12] For simplicity, we just pick the middle of the interval for δ_2 (which is not significantly worse than the optimal choice):

$$\delta_2 = \frac{\mu_i - \mu_j}{2\mu_j}, \delta_1 = \frac{\mu_i - \mu_j}{2\mu_i}$$

We can estimate the worst case number of samples needed to identify the secret (up to an error ϵ) by picking the desired success probability t for $\Pr[S_i > S_j]$, for a reasonable choice of i, j, and then solve for m. We approximate

$$t = \Pr[S_i > S_j] \geq (\Pr[S_i > (1-\delta_1)\mu_i])^2$$

since we pick δ_1 and δ_2 to roughly balance the two Chernoff bounds. Then, we compute the number of samples m as

$$m \leq \frac{-2\ln(1-\sqrt{t})}{\delta_1^2 \cdot \gamma \cdot p_h}.$$

For example, at most $m \approx 10^{19}$ samples are required to recover κ_{hotp} uniquely for 6-digit HOTP witnesses.[13]

[12] To maximise the product of these probabilities, we would need to balance both terms, i.e., have $\delta_1^2 \cdot \mu_i/2 \approx \delta_2^2 \cdot \mu_j/3$. Hence, we would have to pick δ_1, δ_2 such that $\frac{\delta_1}{\delta_2} \approx \sqrt{\frac{2\mu_j}{3\mu_i}}$.

[13] Using $t = 0.81$, $\epsilon = 0$ (the answer needs to match κ_{hotp}), $i = \kappa_{hotp}$ and $j = \kappa_{hotp} + 1 \bmod 10^d$ since distinguishing S_j from S_i is maximally challenging because they only differ by one element. For these choices, we have the expectations $\mu_i = m \cdot \gamma \cdot p_h$ and $\mu_j = m \cdot ((\gamma - \epsilon - 1) \cdot p_h + (\epsilon + 1) \cdot p_l)$.

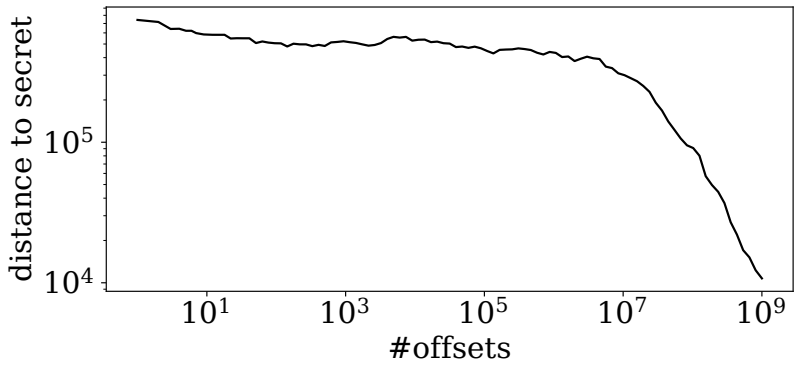


Figure 3.4: Simulated attack on HOTP factor constructions with up to 10^9 samples, averaged over 100 random targets κ_{hotp}.

Attack Cost. TOTP factor constructions of MFKDF, which build on HOTP, require the user to precompute many offsets to allow keys to be re-derived in the future. As TOTP witnesses change every 30 seconds, NS suggest storing 86400 offsets, such that users can be inactive for 30 days without losing access to their key material. This is an unusually high number of HOTP codes that are available to the adversary "for free" in the public MFKDF state. To understand the practicality of our attack, we simulate it by generating samples of the bias using 6-digit HOTP codes (conforming to RFC 4226 [197]) to hide different targets κ_{hotp}.[14] Figure 3.4 shows a log-log plot of the attack success, averaged over 100 targets and up to 10^9 samples. We measure the attack success as the position of target κ_{hotp} in the list of possible values i that is sorted in decreasing order by the overall sum of offsets observed in the interval $[i - \gamma, i] \pmod{10^d}$.[15] This distance decreases from around 300,000 for 10^7 samples to 10700 for 10^9 samples.

Impact. This attack recovers the source key material κ_{hotp} from the bias in HOTP witnesses, given enough observations of the offsets stored in the MFKDF state. While our simulation indicates that this HOTP bias attack is not practical,[16] it directly invalidates the security theorem for the H/TOTP factor constructions in [220, §5.2.2]. In particular, the claims that the offsets "reveal no further information" about the source key material are clearly false.

Unfortunately, this attack also means MFKDF cannot use T/HOTP codes and would require witnesses that are actually uniformly random. This significantly weakens the contribution of MFKDF, as one of its major advantages is the

> compatibility with existing systems using PBKDFs; systems should not need to be entirely rearchitected to use MFKDF, and the user experience should not be impacted whatsoever. [220, § 4].

However, T/HOTP codes are widely used for authentication, and we are not aware of any authentication factors with uniform witnesses that are similarly popular.

[14] Running the simulations took 14.5 hours on an Intel® Xeon® CPU E5-2699A v4 @ 2.40GHz with 44 physical cores and 512 GB of memory.

[15] We conservatively measure attack success and count values that have the same overall sum as the target κ_{hotp} to the values in front of κ_{hotp} to get the worst-case distance of the correct answer from the top of the sorted list.

[16] However, it could be dangerous for an application to allow users to be inactive for many years, which leads to it publishing many more samples in the MFKDF state.

3.3.3.3 *Share Recovery Attack*

This attack impacts the security of the t-out-of-n threshold MFKDF construction (Section 3.2.2.2).

Let F_x be a "lost" factor and let F'_x be the new factor that will replace it. Before the recovery process, the public MFKDF state contains $c_{\mathrm{s}_x} = \mathrm{s}_x \oplus \mathsf{pad}_x$, the encryption of s_x under a pad derived from the source key material of F_x. If the algorithm Recover works as described by NS in the paper,[17] then step 3 uses algorithm Rec to recover s_x with help of the other factors. The recovered share is then re-encrypted with the pad corresponding to the new source key material pad'_x, and the ciphertext in the public state is updated to $c'_{\mathrm{s}_x} = \mathrm{s}_x \oplus \mathsf{pad}'_x$. However, the share itself is not updated.

[17] The reference implementation diverges from this description, and is therefore vulnerable to the share dilution attack described in Section 3.3.2.3.

Attack Overview. The share not being updated when Recover is run has two consequences. First, an adversary who compromises the source key material of the *lost factor* can use it to decrypt c'_{s_x} and recover s_x. Hence key material of a lost factor is as critical for security as that of any still valid factor. Second, the same share s_x is xored twice using different strings pad_x and pad'_x, a "two-time pad". This is completely insecure when the pads have small entropy. If the adversary observes the public state before and after the update, it obtains c_{s_x} and c'_{s_x}. It can therefore compute

$$c_{\mathrm{s}_x} \oplus c'_{\mathrm{s}_x} = \mathsf{pad}_x \oplus \mathsf{pad}'_x$$

and start guessing values for pad_x and pad'_x. For factors of entropy m, pad can only take $M = 2^m$ possible values. We can efficiently recover the values of the pads in $\mathcal{O}(M)$ guesses: first we precompute all the M possible values of $\mathsf{pad}_x \oplus c_{\mathrm{s}_x} \oplus c'_{\mathrm{s}_x}$, storing them in a hashtable. Then we iterate over the M possible values of pad'_x, checking the hashtable for matches. This attack succeeds with overwhelming probability if $2^m \ll 2^\ell$, where ℓ is the bit length of the share. We analyse this in more detail below.

Formal Analysis. Recall the setting: we have a lost factor F and are replacing it with a new factor F'. A secret share s is encrypted with the one-time pad under two different pads, pad and pad′, producing ciphertexts $c_{\mathrm{s}} = \mathrm{s} \oplus \mathsf{pad}$ and $c'_{\mathrm{s}} = \mathrm{s} \oplus \mathsf{pad}'$, where

$$\begin{aligned} \mathsf{pad} &\leftarrow \mathsf{HKDF\text{-}Expand}(\kappa, l) \\ \mathsf{pad}' &\leftarrow \mathsf{HKDF\text{-}Expand}(\kappa', l) \end{aligned} \tag{3.2}$$

are the l-bit outputs of HKDF-Expand on (possibly low-entropy) source key materials κ, κ' associated to F and F', respectively. Given c_{s} and c'_{s}, our goal is to recover s.

κ and κ' have, respectively, min-entropy m and m': we have at most $M \leq 2^m$ (resp. $M' \leq 2^{m'}$) possible distinct values of pad (resp. pad′). Let $P = \{\mathsf{p}_i\}_{i \in [0,M]}$ be the set of possible values of pad (resp. $P' = \{\mathsf{p}'_j\}_{j \in [0,M']}$ the set of possible values of pad′). Observe that $P, P' \subseteq 2^l$, as each pad is an l-bit output of HKDF-Expand. We can enumerate P and P' using the public state of the factors by iterating over all possible

witnesses, thereby recovering the source key material as in Attack 3.3.1, and then using Equation 3.2 to compute the corresponding pads.

In order to recover s, we need to recover the value of pad or pad'. To this end, we first compute a table of values $t_i = \mathsf{p}_i \oplus c_\mathrm{s} \oplus c'_\mathrm{s}$ for each $\mathsf{p}_i \in P$. We then iterate over the values of $\mathsf{p}'_j \in P'$, and check if there exists any t_i such that $t_i = \mathsf{p}'_j$. If that is the case, we have found pads $\mathsf{p}_i, \mathsf{p}'_j$ such that

$$\mathsf{p}_i \oplus c_\mathrm{s} \oplus c'_\mathrm{s} = \mathsf{p}'_j \ . \tag{3.3}$$

We claim that the above only holds when $\mathsf{p}_i = \mathsf{pad}, \mathsf{p}'_j = \mathsf{pad}'$, or symmetrically (if $\mathsf{pad}, \mathsf{pad}' \in P \cap P'$), when $\mathsf{p}_i = \mathsf{pad}', \mathsf{p}'_j = \mathsf{pad}$. This relies on two assumptions.

1. The pad for the lost factor is different from the pad for the new factor (i.e. $\mathsf{pad} \neq \mathsf{pad}'$). Note that if $\mathsf{pad} = \mathsf{pad}'$, the attack is not possible—the observed ciphertext does not change, and we learn nothing from it. Let p_{fail_1} denote the probability that $\mathsf{pad} = \mathsf{pad}'$, i.e., the probability that this assumption fails.

   This can happen for one of two reasons. Either there is a collision in the HKDF-Expand outputs (i.e. $\mathsf{pad} = \mathsf{pad}' \wedge \kappa \neq \kappa'$), or the source key material for the lost factor is the same as the source key material for the new factor (i.e. $\kappa = \kappa'$).

   For the first case, we can assume that HKDF-Expand outputs are uniformly distributed in $\{0,1\}^l$, relying on random-oracle assumptions on the underlying HMAC. The probability of a pad collision can then be determined by the birthday bound: for factors with entropy $m \approx m'$ and $2^m \ll 2^l$, it is approximately $1 - e^{-2^{2m-l}}$. For all practical purposes, this is negligible, so we assume that $p_{\mathrm{fail}_1} = 0$ in this case. For example, according to [220, §6], $m = 20$ for T/HOTP and $m = 40$ for passwords, and $l = 256$ by default in MFKDF.

   For the second case, the probability of the two source key materials colliding depends on the factor constructions used. For instance, if a lost password is replaced with a new, different password, the source key material will change. Instead, if a T/HOTP factor is replaced with another T/HOTP factor with the same number of digits, then it is possible that the new source key material, which is sampled uniformly at random, coincides with the previous one. For factors with source key material sampled uniformly at random, we then have $p_{\mathrm{fail}_1} \approx 2^{-m}$, since the factors have entropy 2^m. For example, when a 6-digit TOTP is used as a factor, $p_{\mathrm{fail}_1} \approx 2^{-20}$.

2. The values $y_{i,j} = \mathsf{p}_i \oplus \mathsf{p}'_j$ are all distinct, save for symmetries. If this were not the case, we would have two colliding values y_{i_1,j_1} and y_{i_2,j_2}, where

$$\mathsf{p}_{i_1} \oplus \mathsf{p}'_{j_1} = \mathsf{p}_{i_2} \oplus \mathsf{p}'_{j_2} \ . \tag{3.4}$$

In this case, we cannot differentiate between correct and incorrect guesses for the pads, since they collide to the same observable $c_s \oplus c'_s$ value.

Let's analyse all the cases in which such a collision can happen:

- If $i_1 = i_2 \wedge j_1 \neq j_2$, then $\mathsf{p}'_{j_1} = \mathsf{p}'_{j_2}$. This is never the case, since the values in P' are all distinct.
- If $i_1 \neq i_2 \wedge j_1 = j_2$, then $\mathsf{p}_{i_1} = \mathsf{p}_{i_2}$. This is never the case, since the values of P are all distinct.
- If $i_1 \neq i_2 \wedge j_1 \neq j_2$, then we consider the cases:
  - $\mathsf{p}_{i_1} = \mathsf{p}_{i_2} \wedge \mathsf{p}'_{j_1} = \mathsf{p}'_{j_2}$. As above, this is never the case: the values of P and P' are all distinct.
  - $\mathsf{p}_{i_1} = \mathsf{p}'_{j_1} \wedge \mathsf{p}_{i_2} = \mathsf{p}'_{j_2}$ or $\mathsf{p}_{i_1} = \mathsf{p}'_{j_2} \wedge \mathsf{p}_{i_2} = \mathsf{p}'_{j_1}$. This can only happen if $P \cap P' \neq \emptyset$. For example, when the new factor uses the same factor construction as the old factor, $P = P'$. Observe that the first case contradicts Assumption 1. The second case is the symmetric case: Equation 3.3 holds for both $\mathsf{pad} = \mathsf{p}_i, \mathsf{pad}' = \mathsf{p}'_j$ and $\mathsf{pad} = \mathsf{p}'_j, \mathsf{pad}' = \mathsf{p}_i$. Hence we are still able to recover the pad values, up to their ordering.
  - $\mathsf{p}_{i_1}, \mathsf{p}_{i_2}, \mathsf{p}'_{j_1}, \mathsf{p}'_{j_2}$ all distinct. If HKDF-Expand outputs are uniformly distributed in $\{0,1\}^l$, so are the pad values. From this, it follows the xor of two uniformly distributed $\mathsf{p}_i, \mathsf{p}_j$ is itself uniformly distributed: the probability of $y_{i,j}$ taking any particular value is then 2^{-l}.

To recap: this assumption fails with probability $p_{\text{fail}_2} = 2^{-l}$. Thus the probability p_{fail_2} is negligible when the HKDF-Expand output space is large, as is the case in MFKDF, where by default $l = 256$.

The probability of either assumption failing is $p_{\text{fail}} \leq p_{\text{fail}_1} + p_{\text{fail}_2} = 2^{-m} + 2^{-l}$. We claimed that Equation 3.3 is only satisfied for $\mathsf{p}_i = \mathsf{pad}, \mathsf{p}'_j = \mathsf{pad}'$ or $\mathsf{p}_i = \mathsf{pad}', \mathsf{p}'_j = \mathsf{pad}$. Consider the constant value $c_s \oplus c'_s$. By the first assumption, we know that it is non-zero, and results from a two-time pad. By the second assumption, we know that this value corresponds to a single pair of $\mathsf{p}_i, \mathsf{p}'_j$. Since we know that $\mathsf{pad} \in P$ and $\mathsf{pad}' \in P'$, our claim holds: we can recover the pads and hence the share.

We ran simulations of our attack, by encrypting the same 256-bit secret using two different pads, derived from a factor with 24 bits of entropy. This is sufficient to capture HOTP factors with 6-digit codes, which would only provide 20 bits of entropy. The attack always recovers the target pads (up to their ordering), and terminates in under 10 seconds on a laptop (Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz with 4 physical cores and 16 GB of memory).

Impact. Our first observation means that the recovery algorithm should not be used as a means of *replacing* a compromised factor (e.g., in the event of a password breach), as witnesses for the lost factor can still

be used to compute the share, therefore bypassing the new factor. By our second observation, the recovery process allows an adversary to efficiently recover the share corresponding to low-entropy factors when they are replaced, violating the security claim that individual MFKDF factors cannot be guessed in isolation.

3.3.3.4 Share Format Attack

Recall that in threshold MFKDF, the public MFKDF state B contains a one-time pad encrypted share c_s and factor construction state β_i for each factor F. NS note that the one-time pad can "be replaced by symmetric-key encryption as long as no checksums or integrity mechanisms are included in the scheme" [220, A.4]. The following attack shows that this esoteric requirement is vital for the security of MFKDF, but that even using the one-time pad, threshold MFKDF is vulnerable to a theoretical guessing attack against individual factors, leveraging the byte representation of Shamir shares.

Attack Overview. The attack assumes that the byte representation of the shares produced by Shamir secret sharing is non-uniform. Note that in Shamir's original description [265], shares are elements of a prime field; for an l-byte prime $p < 256^l$, each share is a value in $[0, p)$, and hence cannot be represented as an l-byte string that is uniformly distributed in $[0, 256^l)$. Since the distribution of shares is non-uniform, we can construct a distinguisher algorithm $\mathcal{D}$ that, given as input either a share or a random byte string of the same length, outputs 1 if the input was a share and 0 otherwise. We refer to the success probability of this algorithm as:

$$p = |\Pr[\mathcal{D}(\mathrm{s}) = 1|\mathrm{s \text{ is a share}}] - \Pr[\mathcal{D}(\mathrm{s}) = 1|\mathrm{s \text{ is random}}]|$$

An attacker can use this distinguisher to guess witnesses for each factor independently of the other factors, as follows. The adversary can check a witness guess W_i^* for F by computing:

$$\kappa_F^* \leftarrow \mathsf{Derive}_1(W_i^*, \beta_i)$$
$$\mathsf{pad}^* \leftarrow \mathsf{HKDF.Exp}(\kappa_F^*, \varepsilon, \ell)$$

It can now use the distinguisher to check if guess was correct by evaluating if $\mathcal{D}(c_s \oplus \mathsf{pad}^*) = 1$. This confirms a correct guess with probability p.

Note that if the encryption scheme included any "integrity mechanisms", the distinguisher could be replaced by the decryption algorithm. The attack would then succeed with overwhelming probability even if the shares were uniform.

Impact. This attack undermines the claims of exponential security of MFKDF, allowing an adversary to attack each factor separately if shares have a biased representation. For example, in case the distinguisher fails with probability 2^{-20}, i.e. $p = 1 - 2^{-20}$, HOTP and TOTP factors are trivial to brute-force since they individually provide only ~20 bits of entropy.

Limitations. In the reference implementation, the share representation used is such that the bias is always contained in the first byte of each share. Additionally, because of an implementation bug, the first 16 bytes of shares are actually not encrypted.[18] Therefore, the reference implementation inadvertently avoids being vulnerable to this attack. However, the attack would work if the shares were properly encrypted.

[18] This leaks one third of the 48-bytes share, but that's insufficient to actually recover the secret.

3.4 MITIGATIONS

In this section, we discuss how some of the attacks from Section 3.3 can be mitigated. Unfortunately, this does not apply to Attack 3.3.1; due to the static nature of key derivation[19] and the mismatch between the threat models of authentication and key derivation, this attack *cannot be mitigated.* The result is that dynamic authentication factors such as H/TOTP, OOB codes and HMAC hardware tokens—the most common MFA methods in practice—are unsuitable for use with MFKDF. This represents a significant drawback for MFKDF, as one of its main goals is to

[19] For functionality, the key must be deterministically derivable from the factors.

> realize the full benefits of multi-factor authentication within the key-derivation process

with

> no noticeable change to the user experience, and while supporting all of the same 2FA factors that users have already grown accustomed to. [220, §2]

Section 3.5 discusses this fundamental problem of MFKDF in more detail; in the rest of this section, we focus on the problems that can be fixed.

State Integrity. It is clear at this point that MFKDF should not be used if its state has no integrity protection: the OOB Overwriting (3.3.2.1), Parameter Tampering (3.3.2.2) and Share Dilution (3.3.2.3) attacks all result from this lack of integrity.

A partial solution could be to give up on some of the flexibility and functionality of MFKDF, such as the support for arbitrary threshold policies and user-specified key length, and embed these parameters in the client software instead of keeping them in the state. While fixing the threshold value, key length, and PBKDF parameters prevents Attacks 3.3.2.2 and 3.3.2.3, it is not possible to hard code public keys to protect against Attack 3.3.2.1. Moreover, other attacks that rely on the lack of integrity of the remaining state may still be possible; this mitigation only helps against some of the concrete vulnerabilities that we found, but does not fundamentally solve the issue of state integrity.

Unfortunately, verifying the integrity of the state presents a "chicken-and-egg" dilemma: the purpose of MFKDF is to derive a key, so we cannot assume the existence of a key which can be used to protect the state, but without a key, the integrity of the state cannot be verified. The only option is to use the MFKDF key, derived from the unprotected state,

to itself ensure the integrity of the state. But this circular approach is vulnerable to attacks where the adversary exerts control over the derived key (in the vein of Attack 3.3.2.2).

Therefore, as it stands, MFKDF itself cannot achieve state integrity. The two plausible alternatives to instantiate MFKDF in a secure way are to rely on a trusted third party to protect the state, or to come up with a new, stateless construction.

In their response to our disclosure, NS suggest the first option, saying that the state can be stored on a blockchain or on IPFS. This impacts the practicality of MFKDF, and leaves open the problem of updating the state: either the state is immutable, or the user would need to authenticate in order to modify it. An MFKDF construction without state might be of interest for future work, but requires considerable change to the proposed design.

*MFKDF2 [253] mitigates state integrity issues in two ways. First, it sets some parameters to fixed values, including key length (32 bytes) and PBKDF (*Argon2id*): this limits the adversarial control over the final key output, making it harder for MFKDF2 to output a low-entropy or easy-to-brute-force final key. Second, it adds an integrity check over most values included in the public state, using a key derived from the final key. This corresponds to the circular approach that we mentioned earlier, and, combined with the static PBKDF and key length, seems to frustrate most practical attacks. A security proof of this construction is still outstanding.*

Cryptographic Primitives. The HOTP Compromise (3.3.3.1) and Share Format (3.3.3.4) Attacks result from underspecified requirements for the cryptographic primitives used in MFKDF. NS argued in their response that this underspecification is intentional, to ensure that MFKDF is "future proof" and can be adapted and implemented as required by applications. We believe the specification should be more stringent and clarify the security properties of the primitives on which MFKDF relies. In this vein, NS agree with us that symmetric encryption should at least be IND-CPA secure (hence in particular excluding the usage of one-time pad).

Independently from our other proposed mitigations, we advise to use AEAD encryption where appropriate to protect dynamic parts of the state from changing. A notable exception is the encryption of shares, which instead requires a non-authenticated scheme to avoid factor brute-force attacks through trial-decryption. Avoiding this class of attack requires non-standard assumptions on the encryption scheme: decryptions under incorrect keys should yield plausible-looking shares. This has been studied by Juels and Ristenpart in their work on *honey encryption* (HE) [177]: we postulate that a HE scheme could be used to encrypt the shares while withstanding brute-force attacks on low-entropy keys.

MFKDF2 mitigates some of these attacks by switching from one-time pad to AES-256-ECB. *While this is a questionable choice of cipher (in particular, it is not IND-CPA secure), practical attacks are limited when* ECB *is used to encrypt high-entropy plaintext, which is the case in MFKDF2.*[20]

[20] Nonetheless, ECB usage already resulted in a new vulnerability in MFKDF2, as shown in the report [252]. That particular vulnerability was fixed by switching to AES-CBC.

The integrity tag over the public state contributes to providing a weak form of ciphertext integrity.

HOTP. While the HOTP Bias Attack (3.3.3.2) can be avoided by discontinuing the use of HOTP codes as an MFKDF factor, the fundamental issue with dynamic factors (including HOTP) remains; they should not be used for key derivation.

MFKDF2 suggests the usage of an external trusted service, dubbed "timing oracle", to provide unbiased one-time codes that can be summed (mod 10^d) to the user's OTPs. At the time of writing this thesis, the timing oracle implementation is not fully specified in MFKDF2.

Share Recovery. As described in Section 3.3.3.3, the Share Recovery Attack can be prevented by having Recover generate fresh shares when a factor is lost. If the share threshold t were to be additionally authenticated or constant, this results in a secure factor recovery mechanism (that does not instead expose the user to the Share Dilution Attack 3.3.2.3).

MFKDF2 does generate fresh shares, and the threshold value is included in the integrity tag.

3.5 DISCUSSION

In our analysis of MFKDF, we found multiple issues with both the proposed constructions and the implementation. However, the most damaging finding is arguably the dynamic factor attack, which points to a fundamental flaw in the *concept* of MFKDF, and cannot be mitigated. This flaw stems from a gap between the two threat models of password-based systems in which MFKDF operates: key derivation and authentication.

In the following, we discuss this gap and why it arises. We compare MFKDF to the state of the art in authentication and key derivation, respectively. Our analysis shows that for authentication, MFKDF is always weaker than traditional MFA, and that for key derivation, MFKDF is no better than using PBKDF on a longer password—and the latter is arguably the safer option from a usability point-of-view.

Authentication. In the setting of (multi-factor) authentication, the goal is to prevent *outsiders* from impersonating the prover towards the verifier. For instance, consider the case of a server authenticating a user to an application such as e-banking, unencrypted email or cloud storage. In its role as a verifier, the server is trusted: it aims to prevent an external actor—which may partially compromise user secrets and observe the communication between user and server—from gaining access to the user's resources. For MFKDF, this threat model is relevant when the derived key is used for authentication; NS propose this usage, arguing that the final key implicitly verifies the authentication factors used to derive it, and can hence replace traditional MFA.

Next, we compare the security achieved by using the MFKDF key for authentication, to directly using traditional authentication factors for MFA. Note, however, that the state-of-the-art for authentication are protocols like WebAuthn [160], which leverage public-key cryptography for

completely passwordless authentication. MFKDF does not try to compete with these more modern alternatives; its goal is instead to strengthen the security of password-based systems, making MFA the appropriate baseline.

MFKDF vs. MFA. In traditional MFA, dynamic authentication factors present a significant advantage over static ones: access to a dynamic factor witness is only useful to authenticate *once* (the witness is "one-time"), and sometimes only during a short time interval (cf. TOTP). This is possible thanks to the threat model, which allows the client to put some trust in the server (concretely, in the form of a shared secret such as an HMAC key, or a server-generated OOB code).

This is not the case in MFKDF. Because the final key has to be static, dynamic factors need to be turned into static ones. Consequently, the one-timeness/freshness properties of dynamic factor witnesses are lost; as shown in Attack 3.3.1, corrupting a single witness allows an adversary to recover the source key material for the corresponding factor.

NS do not discuss the consequences of turning dynamic factors into static ones in their paper, and—in response to our disclosure—dismissed this limitation as inherent to the properties of a key derivation function. However, dynamic factors can make a difference in practice: consider a user authenticating to their e-banking account with a password and a TOTP token, on a device infected with a keylogger. The adversary learns the password and the token witness, allowing it to derive the MFKDF key. This enables the attacker to repeatedly authenticate as the user, also in the future. In contrast, if traditional MFA was used, the attacker would be unable to authenticate at all since it does not have access to the TOTP token, and the captured witness has already been used.

Hence, authenticating with traditional MFA is stronger than using MFKDF. Furthermore, the connotation that, for example, TOTP codes are "one-time" (as alluded to in the name) might convey a false sense of security to users, who should not reasonably be expected to understand that their previously dynamic authentication factors have now become static, and that their "one-time passwords" need to be treated with as much care as their normal passwords.

Key Derivation. In the setting of password-based key derivation, the goal is to extract entropy from a password to produce a cryptographic key, while making brute-force and dictionary attacks hard. The resulting key can then be used for cryptographic operations, without requiring the user to memorize or store the key. Notable use-cases include encrypting files for end-to-end encrypted cloud storage (e.g. MEGA [210]), or protecting passwords stored in an online password manager (e.g. Bitwarden [168]): here, the service provider hosting the encrypted data is not trusted. This threat model differs from that of authentication in two ways.

First, if the key is used to protect outsourced data, the server storing that data is considered malicious. As we outlined under *Authentication* above, this deviates from the authentication setting, where the client

may share secret key material with the server for the purpose of authentication.

Second, the attacker "trivially" wins if it compromises the device of the user where the key derivation is taking place, since it then learns the key. In the case of MFKDF, this is true even if the attacker has not compromised any of the factors used for the key derivation. In the authentication setting, in contrast, compromise of the device where authentication is taking place is insufficient to impersonate a user thanks to the dynamic factors; all factor material must be corrupted in order for the attacker to win (see the keylogger example above).

One of the explicit goals of MFKDF is to bring the security guarantees of MFA to key derivation, but this is unattainable, as the two threat models are in conflict: trust in the server cannot be leveraged in the key derivation setting, and MFKDF cannot help against compromise of the user's device (as MFA does) if the server is also malicious.

MFKDF vs. PBKDF. In light of the above considerations, we argue that MFKDF is no stronger than using PBKDF on a longer password (or, equivalently, the combination of multiple static secrets). First of all, Attack 3.3.1 shows that dynamic factors are unsuitable for use with MFKDF; they have to be turned into static source key material to satisfy the requirements for key derivation, and hence lose their advantage over static factors. Since MFKDF internally applies PBKDF to the concatenated source key material of all factors (see lines 5–6 of Algorithm 6), it is clear that this approach is no more secure than applying PBKDF to a longer password of the same entropy as that of the factors combined. More importantly, users are faced with the non-obvious shift in threat model which implies that one-time codes from their dynamic factors need to be treated as secrets lest they permanently degrade the security of their key.

The other claimed advantages of MFKDF, such as threshold key derivation and factor recovery, have to be traded off against their disadvantages. Indeed, our attacks show that the substantial complexity added by MFKDF can easily make it considerably less secure than just PBKDF. Among the reasons for this are the vulnerabilities that arise from having security critical values in unauthenticated state, as well as implementation pitfalls such as the use of biased Shamir shares combined with encryption, a use case for which they were not designed. Such features would require careful formal treatment and vetting of the implementation.

3.6 CONCLUSIONS

MFKDF is promoted as a more secure alternative to PBKDF, bringing the advantages of multi-factor authentication to password-based key derivation. In our analysis, we showed that the security claims made for MFKDF are void: the proposed construction suffers from vulnerabilities that allow an adversary to compromise or bypass the factors used for key derivation, or directly recover the derived key without corrupting all of the factors. Additionally, even with the mitigable vulnerabilities

patched, MFKDF is not more secure than PBKDF with longer passwords of equivalent entropy.

As a concept, MFKDF proposes a way of harvesting the entropy from authentication factors that a user might already have, and use it for key derivation. Indeed, the authors state that MFKDF provides

> a paradigm shift toward direct cryptographic protection of user data using all available authentication factors, with no noticeable change to the user experience [220].

However, when proposing to use authentication factors for key derivation, they fail to discuss the disparate threat models of these objects. The result is that the protection provided by dynamic factors against impersonation attacks is lost when MFA is replaced by authentication through MFKDF with the same factors. Additionally, dynamic factor witnesses such as HOTP codes are no longer one-time when used for MFKDF; they lose their advantage over static factors, and they instead represent a security risk for users who do not understand this change.

Future Work. Constructing an alternative to PBKDF which strengthens the security of derived keys by combining the password with additional inputs is a highly relevant quest. If we move away from the authentication threat model, we can design dynamic factors specifically for key derivation purposes which can achieve this goal, and provide security for the KDF even under leakage of all the inputs and outputs of the factor. One possible direction is to use OPRF evaluation on a separate device as a second factor: this has been explored by Iliescu in her MSc thesis [165]. MFKDF2 [253], a follow-up to MFKDF which includes fixes for many of the attacks we outlined in this work, has received audits but still lacks a formal security analysis. Provable security for MFKDF2 is also a relevant direction for future work.

Chapter 4

Zero Knowledge (About) Encryption: A Comparative Security Analysis of Four Cloud-based Password Managers

This chapter has been previously published as:

> **Scarlata, M.**, Torrisi, G., Backendal, M., Paterson, K. G. "Zero Knowledge (About) Encryption: A Comparative Security Analysis of Three Cloud-based Password Managers." *35th USENIX Security Symposium (USENIX Security 2026). USENIX Association, August 2026.*
> URL: www.usenix.org/conference/usenixsecurity26/presentation/scarlata

For this chapter, I have made several modifications from the original publication, including extended details that were present in the full version [262] or previously unpublished. In particular, I included a more detailed security analysis of 1Password in this chapter, therefore increasing the count of targets from three to four. I also updated the mitigation advice to match our evolving understanding of the issues raised, and our latest discussions with the password manager companies.

Author contributions
This chapter is based on joint work of all co-authors. This research started as GT's MSc thesis, under the supervision of MB. They set out to create a security model for password managers, and to prove Bitwarden secure in this model. During this modelling attempt, they discovered most of the Bitwarden attacks, and GT developed PoC for them. KGP and I joined the project after the initial attacks on Bitwarden were discovered. GT and I extended the analysis to LastPass, Dashlane and 1Password, discovering further attacks, and developing PoCs for all of them. My main contributions are in this analysis, the coordination of the disclosure efforts, the proposal of mitigations, and the systematization and classification of the attacks.

Summary In this chapter, we provide a comparative security analysis of four major cloud-based password managers: Bitwarden, LastPass, Dashlane and 1Password. We present **31** attacks across the four systems, targeting escrow-based key recovery, credential sharing, vault integrity, and legacy encryption. We discuss mitigations and broader lessons for password manager design.

Chapter Overview:

4.1 Introduction 80
4.2 Threat Model 85
4.3 The Password Managers 88
4.4 Attacks 93
4.5 Mitigations 115
4.6 Ethical Considerations 119
4.7 Open Science 121
4.8 Discussion 122
4.9 Conclusions 124

4.1 INTRODUCTION

Chapter Overview:

4.1 Introduction 80
4.2 Threat Model 85
4.3 The Password Managers 88
4.4 Attacks 93
4.5 Mitigations 115
4.6 Ethical Considerations 119
4.7 Open Science 121
4.8 Discussion 122
4.9 Conclusions 124

Despite the rise of alternative authentication methods, users today still have to deal with passwords, often numbering in the hundreds [223]. Password managers help to tame the problem by providing a tool to securely store passwords, reducing the challenge of remembering many passwords to remembering just the one "master password" for the password manager. Cloud-based password managers outsource the storage to a remote server under the control of a service provider. At an abstract level, a user's passwords are collected in a single object which is then encrypted by the user's client under a cryptographic key derived from the user's master password, creating an *encrypted vault*. The client then uploads the encrypted vault to the server. When a user wishes to access a password for a particular service, their client authenticates to the service, retrieves the encrypted vault, and decrypts it locally with a user-provided copy of the master password.

Importantly in solutions of this type, the service provider does not see the vault plaintext and therefore does not immediately learn the user's passwords or other sensitive data. This is akin to the situation with end-to-end encrypted (E2EE) cloud storage, and while the terms *E2EE* or *client-side encryption* are sometimes used by vendors in this space, the most commonly used term is *Zero Knowledge Encryption*. The term *Zero Knowledge* of course has a specific technical meaning in the context of interactive protocols, but here the term is being used with a different meaning, as we shall see.

The cloud-based approach has multiple advantages: users can access their encrypted vaults from multiple devices; vaults can store other sensitive information beyond passwords (e.g. credit card data, personal documents); and the service can be extended to allow sharing of sensitive data within a family, group or organisation. The "access from anywhere" feature creates work for vendors, who have to support access from web browsers as well as stand-alone applications running on different OSes. Many vendors have offerings which allow the cloud storage element to be self-hosted by an organisation instead of by the vendor.

Four prominent providers in this space are Bitwarden, Dashlane, LastPass and 1Password. At the time of writing, Bitwarden claims to have 10 million users, Dashlane 19 million users and 24,000 business customers,

and LastPass 33 million users and 100,000 business customers [57, 90, 190]. A 2024 report [264] based on a survey of 1000 US consumers gives further insight into the popularity and market share of password managers. The built-in password managers of Google and Apple now represent 55% of the market, up from a combined share of only 15% in 2021. Bitwarden and LastPass are the next two largest, according to the study, with 11% and 10% market share, respectively. Dashlane now has only 2% market share, down from 7% in 2021 when it was amongst the market leaders.[1] The same report places 1Password as the fifth most popular password manager (after Google, Apple, LastPass, and Bitwarden) at 5% market share; the company reports 180,000 business customers and "millions" of users, including one million developers [7]. There is a long tail of smaller players in the market.

[1] The apparent inconsistency between number of users and market share arises due to different sources: vendors' self-reported claims vs market study. Dashlane disputes the accuracy of this source due to the small sample size.

Threat Model. We conduct our analysis of cloud-based password managers in a threat model in which the servers storing users' vaults are assumed to be fully malicious, meaning that they can arbitrarily deviate from expected behaviour when interacting with clients. This is a stronger model than considered in prior work. We present three arguments to justify this in Section 4.2, summarised here. First, vendors' explanations of Zero Knowledge Encryption and E2EE terminology imply that security is maintained in this setting. Second, given the large amount of sensitive data that they store, the service providers are likely targets for sophisticated attackers capable of penetrating the servers and then mounting active attacks (indeed, some vendors have been repeatedly breached [187, 130, 278]). Third, in closely related areas such as cloud storage and messaging, security against a malicious server is by now *de rigueur*.

Security Expectations. Given this threat model, what security should users reasonably expect of password managers? Based on prior work and our examination of vendors' security claims, we argue that the appropriate guarantees are:

1. Confidentiality of all data in a user's vault. This includes the *items* – that is, credentials (usernames and passwords), but also credit card information and notes. Most metadata is not kept confidential in the systems we studied, but achieving this is also desirable.

2. Integrity of vault data (and metadata). This should apply at the level of individual items, but also at the level of the data collection as a whole. It should be infeasible for an adversary to modify existing data or inject new data, undetectably delete or modify any data, or change the semantics of any data in a vault, e.g., by altering metadata. In particular, the integrity of any user-specific settings should be maintained.

3. Secure sharing and account recovery. If these features are enabled, then only the set of other users with whom a given user intends to share some or all of their vault data (or grant an account recovery

capability) should be able to access that data (respectively, recover the account).

Several subsidiary security properties are needed to achieve these goals. For example, it should be hard for a malicious server to weaken the cryptography used by clients (e.g., reducing the iteration count used in password hashing).

More advanced goals can also be envisaged, e.g. vault indistinguishability, oblivious client operations, device privacy, etc. Such goals are far from being achieved by today's password managers and thus out of scope for our analysis; however, see [33] for a recent study of possible security properties.

Our Results. In Section 4.4, we give a detailed analysis of Bitwarden, Dashlane, LastPass and 1Password, presenting a cornucopia of practical attacks. In the artefacts that accompany this thesis, described in Section 4.7, we give Proof of Concept (PoC) implementations of all of these attacks, demonstrating their feasibility. The attacks allow us to downgrade security guarantees, violate security expectations, and even fully compromise users' accounts. Table 4.1 lists the attacks and their impacts. Worryingly, the majority of the attacks allow recovery of passwords – the very thing that the password managers are meant to protect.

We group the attacks into four categories: attacks exploiting the key escrow features used for account recovery and SSO login, attacks based on lack of integrity of the vault as a whole, attacks enabled by the sharing features, and, finally, attacks exploiting backwards compatibility features.

These attacks reveal common design anti-patterns and cryptographic misconceptions. *Lack of authentication* of public keys is widespread. When combined with key escrow and sharing features, this results in the adversary being able to fully compromise vaults. Another recurring failure mode is (wrongly) assuming origin-authentication of public key ciphertexts, leading to *key substitution* attacks against Bitwarden in the style of [14, 161]. LastPass stands out for lacking any form of *ciphertext integrity*, using AES-CBC as its main encryption mode. Thanks to legacy code and backwards compatibility exploits, we can downgrade Bitwarden and Dashlane to similarly hazardous states. We also show that integrity is only achieved for single fields in individual items, instead of at the vault level. This enables cut and paste attacks within items and across the vault. Such attacks can often be chained to compromise the confidentiality of the vault as well. These attacks work even when proper authenticated encryption is used. They are possible because of insufficient key separation in vaults with complex structures and/or a lack of cryptographic binding between data and metadata.

Target Selection. We selected the four providers as targets based on their historical and current market share, the availability of unobfuscated client source code (available for Bitwarden and, partially, for Dashlane and LastPass), the richness of the offered feature set, and the diversity of approaches. Our analysis of 1Password is not as thorough as for the other

three, due to the lack of source code. As noted above, Apple and Google now dominate the password manager market. However, the unavailability of source code precludes easy analysis of these systems. Additionally, E2EE is an opt-in feature for Google's password manager, so users do not enjoy security against malicious servers by default. Finally, while numerous other cloud-based password managers could also have been studied, our deep examination of the four selected products – representing over 60 million users and 28% of the market – was already sufficient to surface severe design flaws and common misunderstandings.

Mitigations and Disclosure. We describe potential mitigations for each attack in Section 4.5. We also discuss general strategies to avoid the common pitfalls we discovered. We have shared these mitigations as part of a detailed, vendor-specific disclosure document with Bitwarden, LastPass, Dashlane and 1Password. In each case we have followed an industry-standard coordinated vulnerability disclosure approach, giving vendors at least 90 days to deploy countermeasures. For more details, see our ethics analysis in Section 4.6.

Broader Impact and Contributions. The immediate impact of our work is to help the affected vendors patch their vulnerabilities, thereby providing stronger security for very sensitive data of millions of users. But the impact of our findings also goes beyond the specific products they affect to illustrate some of the frontiers in the space of end-to-end encrypted applications. The importance of the integrity of vaults is a new finding, but parallels can be drawn with similar, recent notions of integrity for cloud storage [37]. Building systems that support key escrow without opening up attack avenues for malicious actors is considered a hard problem, one that has arisen repeatedly in the arena of lawful access and the Crypto Wars [8]. The problem of reliably authenticating public keys that (as we show) plagues password managers also comes up in the messaging space, where it is being tackled by the introduction of Auditable Key Directories, cf. [191]. Section 4.8 considers these and other issues at greater length.

Beyond categorizing the common pitfalls of E2EE cloud-based password managers and drawing parallels to where these arise in other E2EE systems, a main contribution of our work is to firmly establish the need for formal study of these products and their security. The number of severe vulnerabilities we uncover among a diverse set of vendors shows that getting end-to-end encryption right in this setting is not easy. Building on initial notions for password manager security in [128], we propose as future work to formally define the goals of E2EE against a malicious server, akin to what was recently done for cloud storage [37]. Analysing cloud-based password managers in this threat model is a novel aspect of our work, and we have seen from our interactions with vendors that our attacks provide a strong motivation to work towards achieving theoretically well-founded guarantees in this setting, promoting the benefits of formal analysis and provable security in practice.

Related Work. There was substantial prior analysis of password managers shortly after they first appeared on the market. However, none of this work considered the malicious server threat model as we do. Early analysis started more than a decade ago. Li et al. [192] examined five browser-based password managers (LastPass and four others that are no longer extant), uncovering vulnerabilities in four categories: bookmarklet issues, classic web issues, authorization issues, and UI issues. The threat model used in [192] is that of a "web attacker". In the same timeframe, Stock and Johns [274] studied the impact of XSS-based attacks on password managers. In contrast to [192, 274], our vulnerabilities are of markedly different type, stemming more from cryptographic issues, and in some cases are more devastating. Meanwhile, Silver et al. [270] examined the dangers of autofill while Fahl et al. [118] studied the impact of malicious Android apps on password managers. Gasti and Rasmussen [128] provided a theoretical treatment of a range of password manager database formats, focussed on a local storage model and assuming the adversary has either read-only or read-write access to the encrypted database. Such capabilities are available to our malicious server adversary for the user vaults that it stores. Gasti and Rasmussen [128] found only one format offering both integrity and secrecy in their attack model, indicating developers' lack of understanding of the need to provide appropriate cryptographic protection. In contrast to our work, [128] did not explore the practical consequences of their findings. Passive (rather than active) attacks on password manager applications running on iOS and Blackberry were presented by Belenko and Sklyarov in [42]; this work highlighted the use of weak PIN protections or lack of suitably strong password-based key derivation functions. LastPass was one of the targeted products; in 2012 when [42] was published, it used only one iteration of SHA-256 to derive the vault master key from the user password. This has improved since. Zhao et al. [292] also considered the security of LastPass, but focussed mostly on the complexity of brute-force attacks on its password hashing mechanism; this paper also observed that a passive server adversary could observe sensitive data in the RoboForm product because of lack of client-side encryption, a valid attack in their insider attack model which allowed for server-side monitoring and data theft (but not an actively malicious server).

In 2020, Oesch and Ruoti [224] revisited some of this early work to consider the full password manager lifecycle for seven different standalone managers (including the four that we study) and five browser-provided managers. Their consideration of password storage documented vault encryption mechanisms and briefly studied metadata privacy. Their main takeaway is that password managers have improved their security compared to how they performed in earlier studies, though still with significant weaknesses in the areas of unencrypted metadata, insecure defaults, and vulnerability to clickjacking attacks. A similar study was carried out by Carr and Shahandashti in [74], focussed on phishing, clipboard and PIN brute-forcing vulnerabilities. A further five years later, our work challenges the findings of [224, 74] in the malicious server

threat model that we have justified to be appropriate for cloud-based password managers. Fábrega et al. [116] considered injection attacks against ten different password managers, wherein an adversary (only) controls their own application client, which they use to "inject" chosen payloads to a victim's client via, for example, sharing credentials with them. The adversary is also able to observe the protected vaults in some form. The threat model in [116] is different to ours – roughly speaking it considers a malicious client in combination with a passive server (or a network adversary) whereas we consider a malicious server. Finally, Duan et al. [109] provided a categorisation and formal models in the UC framework, and studied offline guessing attacks against 43 password managers.

Parallel to our work, Avoine et al. [33] study password managers in a threat model where the server remains "covert" in any attack it mounts. This preprint classifies cloud-based password managers into three types and sketches attacks by type. The attacks violate vault integrity, as well as several weaker properties not currently targeted by vendors. The attacks' impacts are not fully explored, and there are no PoCs. Some of them rely on weak passwords and/or low PBKDF iteration counts.

A raft of work has studied password managers from human and usability perspectives [78, 178, 209, 117, 75, 232, 162, 241, 225, 208]. In addition, studies on general password usability, password policies, technical measures for improving password security, and password breaches abound.

Close in spirit to our work is a recent line of work analysing the security of E2EE cloud storage providers in the malicious server setting [38, 159, 19, 14, 161]. Many vendors in this space also use the term Zero Knowledge Encryption in a similar way to vendors of password managers. Our findings are broadly analogous to the ones of these papers, though significantly different in the technical details of attacks. We compare our work with this line of work in more detail in Section 4.8.

4.2 THREAT MODEL

As we laid out in our discussion of related work, there has been a significant amount of security analysis of password managers over the last decade (and more). At the same time, the landscape of password managers has changed, with cloud-based solutions becoming significantly more common. This makes it timely to revisit the question: what is the appropriate threat model in this setting?

Before going further, we note that there is a trivial attack against end-to-end encrypted applications that we wish to take out of consideration from the outset. Namely, a user can be supplied with a malicious client which, for example, simply returns the user's master password to the server in the clear. Such an attack would be detectable through code audits and – though possible via an obfuscated functionality – would be very risky for a vendor to conduct. This is especially true for signed software releases (e.g., desktop and mobile clients, browser extensions),

as the existence of a signed malicious client would constitute proof of vendor misconduct.

Setting this trivial attack aside, we argue here that it is appropriate to consider a threat model in which the server is considered fully malicious, meaning that it can deviate arbitrarily from its expected behaviour. Such deviations are hard to detect, especially when targeted at selected users. We argue that this threat model is motivated by the advertised security guarantees by the vendors, the sensitivity of vault data, and the fact that malicious-server security is by now the norm for outsourced data in other, related settings. We expand on each of these below, and end by briefly discussing the role of client interaction in this threat model.

Other E2EE Domains. Security in the face of a malicious server is by now the *de facto* standard in related settings such as messaging systems and cloud storage, where sensitive user data (in the form of private messages or files) is also relayed or stored by service providers. In particular, the malicious server threat model is the accepted interpretation of the security guarantees provided by end-to-end encryption in these settings. (We note that some vendors of E2EE cloud storage also use "Zero Knowledge Encryption" to refer to E2EE. Thus, also in this setting, this term is taken to mean data confidentiality and integrity in the face of a malicious or compromised server.) For example, recent analyses of cloud storage systems [38, 159, 19, 14, 161] have forced vendors in this space to upgrade their approach to provide security against a malicious server, while major messaging systems like Signal, WhatsApp and iMessage have long featured E2EE in the standard sense (Telegram remains a notable and regrettable exception in this area, with E2EE only being optionally available on a per conversation basis rather than as a default, and not available at all for groups). We think it is reasonable to expect these terms (zero knowledge and E2E encryption) to have the same meaning in the context of password managers.

Passwords have High Value. Passwords are among the most sensitive kind of user data, as they often grant access to accounts and services which may in turn reveal more data, or allow an attacker to impersonate the victim. Since password manager servers host high concentrations of passwords from thousands of users, they thus provide an attractive target for hackers. Attacks on the provider server infrastructure can be prevented by carefully designed operational security measures, but it is well within the bounds of reason to assume that these services are targeted by sophisticated nation-state-level adversaries, for example via software supply-chain attacks [76] or spear-phishing. Moreover, some of the service providers have a history of being breached – for example LastPass suffered breaches in 2015 and 2022, and another serious security incident in 2021. In the 2022 breach, the attacker gained access to the development environment, source code, and technical information, used that information to compromise the computer of a senior DevOps engineer, then gained access to an encrypted corporate vault, and finally to the keys of the Amazon S3 "buckets" of the backups to customer files [130,

278]. In a self-hosted deployment involving a less technically advanced customer, it seems highly likely that the server hosting user vaults could not resist such a determined attack. While none of the breaches we are aware of involved reprogramming the server to make it undertake malicious actions, this goes just one step beyond attacks on password manager service providers that have been documented. Active attacks more broadly have been documented in the wild [257, 169].

Vendors' Statements. Additionally, we provide example statements made by the providers which we believe justify this threat model here (taken from their web sites and white papers where they explain their "Zero Knowledge Encryption" claims).

> "Zero knowledge encryption: Bitwarden team members cannot see your passwords. Your data remains end-to-end encrypted with your individual email and master password. [...] Since it's fully encrypted before it ever leaves your device, only you have access to your data. Not even the team at Bitwarden can read your data (even if we wanted to)." [59]

> "Dashlane Password Manager is designed using zero-knowledge architecture, with the data encrypted locally on the user's device. Only the user can access the data by using a password or another form of authentication. Since Dashlane doesn't have access to the user's vault and doesn't store the user's Master Password, malicious actors can't steal the information, even if Dashlane's servers are compromised." [94]

> "With a zero-knowledge approach, you can rest easy knowing that no one else but you, not even your password manager vendor, has the keys to the kingdom. [...] For example, zero-knowledge means that no one has access to your master password for LastPass or the data stored in your LastPass vault, except you (not even LastPass)." [124]

> "Zero-knowledge encryption means that no one but you – not even the company that's storing the data – can access and decrypt your data. This protects your information even if the server where it's held is ever breached." [6]

While none of the vendors quoted here explicitly describe the threat model under which they expect their product to remain secure in terms of attacker capabilities, we argue that – based on statements like the ones above – their customers could reasonably deduce that their vault data remains protected even against a malicious server. Moreover, none of the vendors objected to our adoption of this threat model during the disclosure process.[2]

[2] LastPass have communicated that they don't adopt a generic malicious server threat model, but apply compensating controls to mitigate server risks. 1Password explicitly notes that they don't verify public keys, and confirmed that they view the server as trusted when it comes to providing authentic public keys or authentic public encryption ciphertexts.

Client Interaction. While our threat model concerns a malicious server, some of our attacks rely on interaction with the (honest) client victim. We treat the amount of client interaction needed for an attack to succeed as a resource, and for each attack, we list the expected client interactions required.

The majority of our attacks require simple interactions which users or their clients perform routinely as part of their usage of the product, such as logging in to their account, opening the vault and viewing the items, or performing periodic synchronization of data. We also present attacks that require more complex user actions, such as key rotations, joining an organisation, sharing credentials, or even clicking on a misleading dialog. Although assessing the probability of these actions is challenging, we believe that, within a vast user base, many users will likely perform them. Table 4.1 summarises the interaction requirements for each attack.

4.3 THE PASSWORD MANAGERS

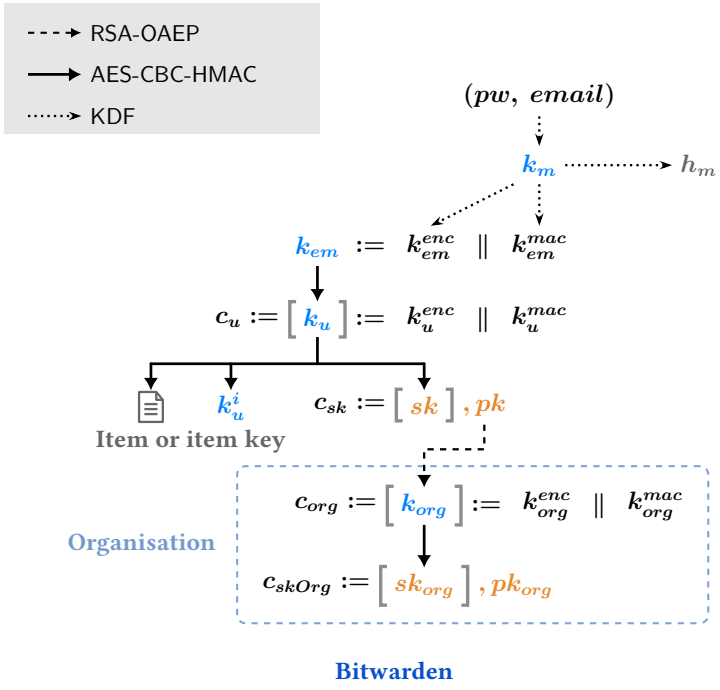


Figure 4.1: Key hierarchy for Bitwarden.

4.3.1 *Bitwarden*

The key hierarchy of a Bitwarden user, illustrated in Figure 4.1, is rooted in the user's master password and email. These are used respectively as key and salt of a KDF, which can either be PBKDF2 or Argon2id, in order to generate a 32-byte-long *master key* (k_m). The master key is used for authentication by sending the result (h_m) of a single iteration of PBKDF2 on the master key with the user's password as salt to the server, which compares it to the value of h_m stored at registration. The

master key is additionally used to derive two 32-byte keys via HKDF-Expand with two different input labels: `"enc"` and `"mac"`. The resulting keys (k_{em}^{enc}, k_{em}^{mac}) are concatenated to form the 64-byte *extended master key* (k_{em}).

After deriving the extended master key, a 64-byte *user key* (k_u) is sampled uniformly at random. Similarly to k_{em}, this key can be interpreted as consisting of an encryption (k_u^{enc}) and a MAC key (k_u^{mac}), each 32-bytes. The role of the user key is crucial: it protects all the data inside the user's vault, either by directly encrypting vault items, or indirectly (if so-called "per-item keys" are enabled, as explained below). Once the user key is generated, it is encrypted under k_{em} and stored on the server to be served at the next login.

Additionally, the client generates an RSA key pair (with a 2048-bit modulus) consisting of a public key pk and a private key sk. This key pair is used in the context of Bitwarden's organisation as described below. Finally, the private key is also encrypted under the user key and sent to the server.

Bitwarden allows users and businesses to self-host the server, in order to have complete control of their data. For users of self-hosted Bitwarden instances, the adversary in our threat model will thus be the self-hosted server.

Key and item encryption. Bitwarden uses AES-CBC-HMAC for authenticated encryption of keys and vault items, using the "enc" and "mac" parts of the key for encryption and MACing, respectively. For backwards compatibility, the HMAC is omitted if the "mac" key part is missing. That is, if the client attempts to encrypt using a key which is only 32 bytes, the client falls back to using AES-CBC.

In vault items, each item data field is encrypted individually. Item fields used to be encrypted directly with the user key k_u, but from version `2024.2.0` new items are encrypted with individually sampled *per-item* keys, which are in turn encrypted by the user key. Old items remain encrypted directly under k_u.

Organisations. Credential sharing is implemented in Bitwarden through organisations. When a user creates an organisation, their client samples a new 64-byte *organisation symmetric key* k_{org}, and encrypts this key under the user's public key, yielding c_{org}^{user}. Additionally, the client also generates an *organisation* RSA *key pair* (pk_{org}, sk_{org}) (used for account recovery purposes), and encrypts the private key using k_{org} to create c_{skOrg} (see Figure 4.1). It then sends c_{org}^{user}, pk_{org}, and the encrypted c_{skOrg} to the server. Items in an organisation are individually encrypted using the organisation symmetric key.

The creator of an organisation can add new members by encrypting the organisation symmetric key for the public key of the invitee. For each user $user'$, the resulting ciphertext $c_{org}^{user'}$ is sent to the server for storage, and served to $user'$ when syncing. The client of $user'$ can then decrypt $c_{org}^{user'}$ using their private key to obtain k_{org}. A client determines that the user is a member of an organisation if it can obtain k_{org}.

Further organisation data, such as the organisation name and policies regarding account recovery, for example, are stored unencrypted on the server and sent to the client whenever a sync operation is performed.

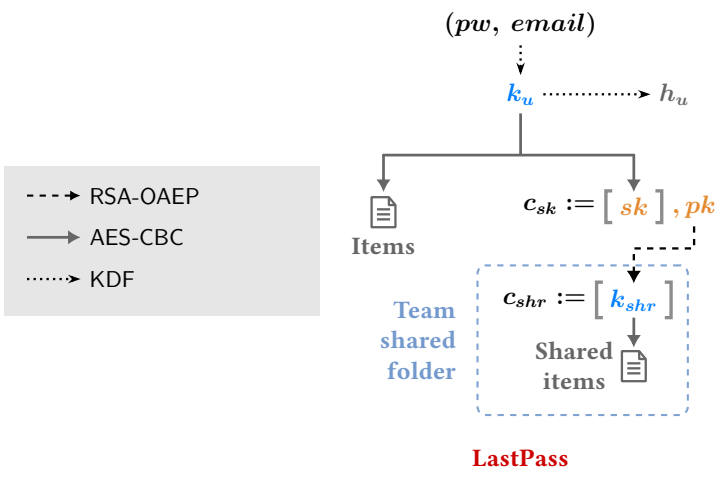


Figure 4.2: Key hierarchy for LastPass.

4.3.2 *LastPass*

As in Bitwarden, the key hierarchy in LastPass (illustrated in Figure 4.2) begins with the user's password, which is used to derive a 32-byte *user key* k_u using PBKDF2 with the user's email address as salt. The default KDF iteration count is 600,000. To authenticate, the client sends h_u, the result of one more iteration of PBKDF2 over k_u with the password as salt, to the server; the server checks if the received h_u matches the value received at registration.

Key and item encryption. The user key is used to directly encrypt the items in the user's vault using AES-CBC (with no integrity protection). Each user also has an RSA key pair (sk, pk); the secret key is encrypted under k_u using AES-CBC and stored at the server.

Sharing. Sharing among LastPass users is implemented through shared folders. When a user first shares an item, their client samples a 32-byte shared folder key k_{shr}. This is subsequently used to encrypt the items in the shared folder using AES-CBC. The folder key is itself encrypted using RSA-OAEP under the public key of every user with whom the folder is shared.

4.3.3 *Dashlane*

Dashlane's vault is maintained in the form of a transactional database. Each transaction can create, modify, or delete an item in the database: the transaction type and timestamp are unencrypted metadata fields, while the transaction content, consisting of compressed XML data with information about items or user settings, is encrypted.

Key and item encryption. Content is encrypted using a key derived from the user's master password. The exact details for these operations

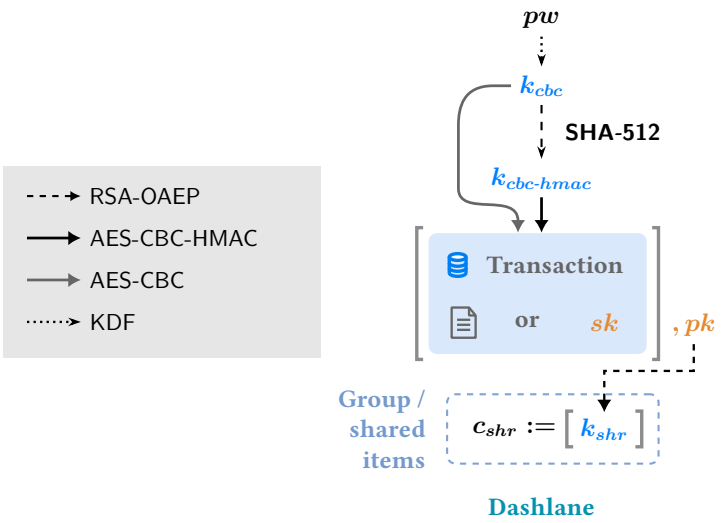


Figure 4.3: Key hierarchy for Dashlane.

are specified in the first bytes of the content by a *payload* string: this defines the password-based key derivation function (either PBKDF2, Argon2d, or noderivation), its parameters (e.g., salt length, hash method, and iterations for PBKDF2), the cipher (by default AES-256), and the cipher mode (by default CBC with HMAC). First, an intermediate 32-byte key k_{cbc} is derived from the password. If the cipher mode is CBC with HMAC, then an additional SHA-512 is used to derive the 64-byte $k_{cbc\text{-}hmac}$. When the user modifies their vault (e.g., by adding an item), a new transaction is created and sent to the server. See the illustration in Figure 4.3.

Sharing. Each user also has a 2048-bit modulus RSA key pair (sk, pk); the secret key is stored encrypted (as a transaction) on the server, together with the public key. The RSA keys are used for sharing: when a user shares an item with another user (either directly or in a group), a symmetric encryption key k_{shr} is sampled and encrypted for the recipient(s) public keys (which the sharer retrieves from the server).

Note. The only fully-available open-source Dashlane client is the CLI client, which does not support encryption nor the credential sharing feature, so the description we provide here is based in part on reverse engineering and may be incomplete. We focus on the parts of the system relevant to our attacks, for which the functionality we describe here is verified by the PoCs.

4.3.4 *1Password*

Similar to the other password managers we study, one root of security in 1Password is the user's master password—but unlike the other three products, 1Password also includes a high-entropy cryptographic key in the key derivation, the so-called *secret key*. This means that users cannot access their vaults using just their master password: they also need access to the secret key. To enable multi-device support, this key needs to be transferred between devices prior to user authorization: this can be done

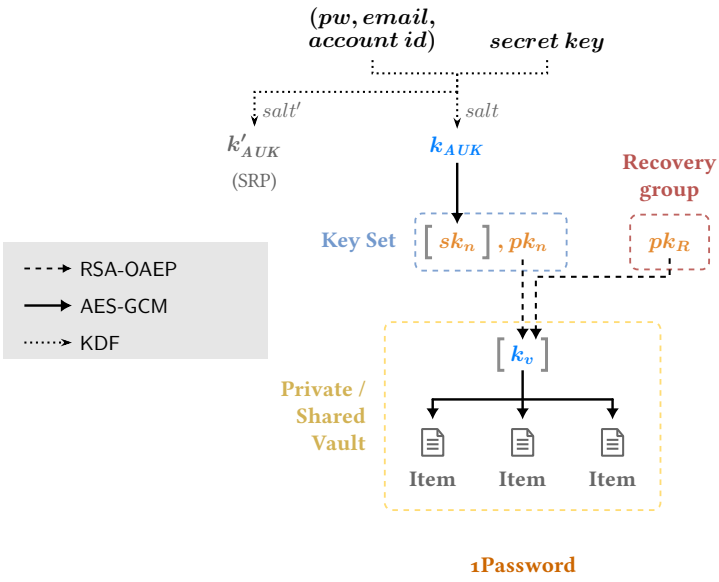


Figure 4.4: Key hierarchy for 1Password. Some levels of key wrapping have been omitted for simplicity. For instance, k_{AUK} encrypts the symmetric key in the key set, which in turn encrypts the secret key—here we show the k_{AUK} as directly encrypting sk_n.

manually by the user, or automatically, using a previously enrolled device to scan a QR code.[3]

1Password therefore starts with a security advantage compared to its competitors (at the cost of some loss of usability)[4]: thanks to the involvement of the secret key, brute-force attacks should be out of reach.

Key and item encryption. The key derivation process is documented in the 1Password whitepaper [4]: at a high level, PBKDF2 (a default of 650,000 iterations in the version we analysed) derives a key from the master password, which is then combined by XOR with a pseudorandom stream derived from the secret key via HKDF-SHA-256. The resulting key is the AUK (Account Unlock Key), denoted k_{AUK}. The k_{AUK} encrypts (AES-GCM) the symmetric key of the user's *keyset* – a bundle containing that symmetric key together with an RSA encryption key pair (and a signing key pair) – and that symmetric key in turn protects the keyset's RSA private key. The keyset's public key is used to encrypt each *vault key* k_v (RSA-OAEP); the vault key encrypts individual items (AES-GCM). Figure 4.4 depicts this hierarchy.

Sharing and Recovery. 1Password users can share vaults: the client fetches the public part of the recipient's keyset and encrypts the vault key under it. 1Password additionally supports recovery via a *recovery group* – a special keyset created at account initialization.[5] Clients encrypt vault keys (and the user's own keyset) under the recovery group's public key.

Client-to-Server Protocol. Like the other password managers we study, 1Password protects all client-to-server communication with TLS. As an additional security layer, 1Password uses a custom secure channel protocol based on the SRP-6a PAKE [164] as a key exchange, and JWT messages with AES-GCM encryption as a record protocol. The client's SRP secret k'_{AUK} is derived from the master password and secret key via the same two-secret KDF as the AUK, but with a different (server-

[3] 1Password actually implements a variety of ways of transferring the secret key. The enrolled device can show a "setup" QR code, which contains the secret key and other account information, for the new device to scan. Alternatively, the new device can show a QR code, that the enrolled device scans; the QR contains a cryptographic key, used to instantiate a secure channel between the two devices ("Mycelium"); the secret key is then transferred using this channel. Another flow, available for SSO users, consists of the enrolled device showing a short enrollment code, which can be typed on the other device; this code is used for a password-authenticated key exchange (CPace), once again instantiating a device-to-device secure channel, used to transfer key material [5].

[4] We found that actually, from our empirical experience, this system was quite easy to use: properly engineered clients remove much of the friction of having to manage cryptographic key material.

[5] For Individual accounts, the group is generated client-side at account setup; for Family/Team it's an account-level group whose private key is delegated to admins/owners.

supplied) salt. This means that the protocol can only be terminated at the 1Password server, which stores $g^{k'_{AUK}}$, the secret verifier, rather than at any earlier TLS termination point, such as load balancers.

4.4 ATTACKS

We now present our attacks on Bitwarden, LastPass, Dashlane and 1Password, grouped into four categories based on cause, as follows:

1. In E2EE password managers, forgetting the master password means that a user irrecoverably loses access to their vault. Thus, many password managers implement some form of account recovery mechanism through **key escrow**, such that an administrator can help users recover their vault contents. In Section 4.4.1, we show attacks against vault confidentiality in Bitwarden, LastPass and 1Password resulting from vulnerabilities in their key escrow designs. We also discuss a weakness in Dashlane's design.

2. None of the password managers we analyse treat the password vault as a monolithic object. Rather, all data items (as well as sensitive user settings) are encrypted as separate objects, and often combined with unencrypted or unauthenticated metadata. In Section 4.4.2, we show how such an *item-level vault encryption* design is often in tension with the **vault integrity** security goal. We find attacks against all of Bitwarden, LastPass, Dashlane and 1Password. Notably, many of these integrity violations also lead to concrete attacks impacting confidentiality.

3. Many password managers implement **sharing** features (targeting for example families and businesses) which allow multiple people to access common shared credentials. In Section 4.4.3, we show attacks against the sharing features of all four of our targets. These impact both vault integrity and confidentiality.

4. All four password managers that we analyse have undergone cryptographic evolution over time. They all maintain some form of **backwards compatibility** to support older client versions—for instance, maintaining support for older encryption schemes, or for different sets of KDF parameters. In Section 4.4.4, we show that such "legacy code" creates cryptographic hazards. Specifically, a malicious server can trigger legacy code paths to be executed in clients, resulting in downgrade attacks in Bitwarden and Dashlane. This holds even if a user intends to use the latest version of the clients. Through such attacks, an adversary can for example downgrade clients to use unauthenticated encryption, opening up avenues for further attacks that violate both vault integrity and confidentiality.

Table 4.1 gives a complete summary of our attacks. Throughout this section, we refer to "the adversary" as an entity that has full control of

Class	Ref	Name	Cause	Impact	Interaction	Mitigations
Key Escrow	BW01	Malicious Auto-Enrolment	Lack of Key Auth, Key Substitution	Full vault compromise †	1 join	PKA, SC
	BW02	Malicious Key Rotation	Key Substitution	Full vault compromise †	1 rotation	SC
	BW03	Malicious KC Conversion	Lack of Key Auth, Key Substitution	Full vault compromise †	1 dialog	PKA, SC
	LP01	Malicious Password Reset	Lack of Key Auth	Full vault compromise †	1 login	PKA, SC
	1P01	Malicious Recovery Group	Lack of Key Auth	Full new vault compromise ‡	1 open	PKA, SC
Vault Integrity	BW04	Unprotected Item Metadata	Lack of Auth Enc	Read and modify metadata	-	AE, AD
	BW05	Item Field Swapping	Lack of Key Sep	Field and item swapping	-	KS
	BW06	Icon URL Item Decryption	Lack of Key Sep	Loss of confidentiality †	1 open	KS
	BW07	Remove KDF Iterations	Lack of Auth Enc	No brute-force protection	1 login	KDF
	LP02	Item Field Swapping	Lack of Auth Enc	Field and item swapping	-	KS, AE
	LP03	Icon URL Item Decryption	Lack of Key Sep	Loss of confidentiality †	1 open	KS, AE
	LP04	Remove KDF Iterations	Lack of Auth Enc	No brute-force protection	1 login	KDF
	LP05	Malleable Vault	Lack of Auth Enc	Loss of vault integrity	-	AE
	LP06	Unprotected Item Metadata	Lack of Auth Enc	Read and modify metadata	-	AD, AE
	DL01	Transaction Replay	Lack of Vault Tag	Loss of vault integrity	-	KS, AD
	1P02	Item Dropping/Duplication	Lack of Vault Tag	Loss of vault integrity	-	AD
	1P03	Vault Injection	Lack of Ctxt Auth	Loss of vault integrity	-	SC
	1P04	Remove KDF Iterations	Lack of Auth Enc	Less brute-force protection	1 login	KDF
Sharing	BW08	Organisation Injection	Lack of Ctxt Auth	Add users to arbitrary orgs	1 sync	SC
	BW09	Organisation Overwrite	Lack of Ctxt Auth	Organisation compromise †	1 join	SC
	LP07	Sharing Key Overwrite	Lack of Key Auth	Shared vault compromise †	1 join	PKA
	DL02	Sharing Key Overwrite	Lack of Key Auth	Shared vault compromise †	1 join	PKA
	1P05	Sharing Key Overwrite	Lack of Key Auth	Shared vault compromise †	1 join	PKA
	1P06	Vault Substitution Attack	Lack Ctxt Auth	Vault compromise ‡	1 open	SC
Backwards Compatibility	BW10	Disable Per-Item Keys	Lack of Auth Enc	Downgrade key hierarchy	-	KS, AD
	BW11	User Key Overwriting	CBC Support	Loss of confidentiality †	2 logins	AE
	BW12	Downgrade to Legacy	CBC Support	Full vault compromise †	2 logins	AE
	DL03	Item Injection	CBC Support	Loss of vault integrity	10^4 syncs	AE
	DL04	Remove KDF Iterations	CBC Support	No brute-force protection	10^4 syncs	AE
	DL05	CBC-Only Downgrade	CBC Support	Loss of confidentiality ‡	10^5 syncs	AE
	DL06	Lucky 64	CBC Support	No brute-force protection	10^4 syncs	AE

Table 4.1: Summary of attacks. The attacks are grouped in four categories, depending on the password manager feature they exploit. The attack ref(erence) also indicates the product: BW for Bitwarden, LP for LastPass, DL for Dashlane and 1P for 1Password. We highlight the main causes for the attack, its high-level impact († denotes recovery of encrypted items in user vaults, ‡ recovery of encrypted items created after the attack took place) and the client interaction required: periodic or user-triggered synchronisation (sync); the user logging in (login), opening the vault (open), joining an organisation (join), sharing a vault (share) or clicking on a misleading dialog (dialog). Finally, we report the mitigations from Section 4.5.

the server. Table 4.2 summarises the analysed software versions and the availability of source code. Unless otherwise specified, all of the attacks affect all the clients mentioned in the table.[6]

Analysis Method. Our approach to analysing password managers follows the "attack" branch of the "Cryptography in the Wild" methodology that is described in detail in [22]. We explain here how we deployed this methodology in the specific context of password managers.

Our process began with reviewing white papers and client code (and server code, when available). We initially focused on Bitwarden, where the availability of source code eased our analysis and enabled us to more easily build our mental model of architecture, protocols, and system features (cf. *object selection* in [22]). From the outset, our goal was to carry out an analysis in the face of a malicious or compromised server: based

[6] Due to the considerable engineering effort required for testing all the clients, we limit our PoC testing to Bitwarden's web client and the browser extensions for LastPass and Dashlane. For most of the attacks, it is evident that they are design issues, independent of specific client implementations.

	Source	Version	Features
Bitwarden			
Web Client	●[58]	`v2024.10.2`	■
CLI Client	●[58]	`v2024.10.2`	◨
Server	●[58]	`v2024.10.2`	-
LastPass			
Web Client	○	Unknown	◨
Browser Ext.	○	`4.140.0`	■
CLI Client	●[2]	`v.1.6.1`	◨
Dashlane			
Web Client	○	Unknown	◨
Browser Ext.	◑[92]	`6.2513.1`	■
CLI Client	●[93]	`v6.2447.2`	◨
Android Client	◑[91]	`018f827f2a01`	■
1Password			
Web Client	○	2285	■
Browser Ext.	○	8.11.4.27	◨

Table 4.2: Summary of the software distributions of the password managers we study, detailing source code availability ranging from fully open source (●), open source releases omitting fundamental cryptographic libraries (◑), to closed source (○); the software version; and whether the client supports the full core feature set of the password manager (■) or just a small subset of operations (◨).

on the functionality of the target systems, we refined this to a concrete *adversarial model*. We developed pseudo-code models for the main cryptographic actions of each feature of interest (*ingestion*). We then analysed these models to find *attacks*. Here we relied on our own experience in finding cryptographic vulnerabilities in systems, along with a list of potential attack vectors gained from the by-now-extensive literature on cryptographic attacks. For password managers, we drew particular inspiration from recent literature on E2EE cloud storage providers in the malicious server setting [38, 159, 19, 14, 161] as well as common attack vectors such as padding oracle attacks [284], key overwriting in various forms, cf. [72], and exploitation of backwards compatibility features, cf. [172]. Finally, we built *PoC* implementations for the attacks, implementing malicious server functionality where needed, and testing against the relevant client. This step is vital for ensuring that attacks developed "on paper" actually work in practice.

Following our initial study of Bitwarden, we iterated the same process to determine whether our findings were applicable to other products. For Dashlane and LastPass, we carried out some reverse engineering of minified client Javascript and of the Android applications. Having already studied Bitwarden helped us in developing our understanding of how these products operate. After finding more attacks on both of them, we extended our scope to 1Password, which required some additional reverse engineering effort. We confirmed that our attacks were applicable to the newly selected targets. Additionally, we manually classified the different attacks and grouped them into four categories, providing a starting point for follow-up work analysing further password manager products. This extends the methodology from [22].

Note that absent formal analyses, the set of attacks we found must be considered incomplete. Indeed, in some cases, we stopped looking for

attacks after accumulating sufficiently many severe enough vulnerabilities to demonstrate that the given target was vulnerable in our threat model. Future work may involve employing formal tools to uncover other vulnerabilities, or conclusively demonstrate the lack thereof.

4.4.1 *Key Escrow Attacks*

These attacks all concern various forms of key escrow functionalities provided by the password managers to enable account recovery in the case of master password loss. Successful attacks on these recovery mechanisms have severe consequences. In most cases, the adversary can recover the full vault of the user, therefore compromising all of the data that it contains. Some recovery mechanisms are less powerful than others, and rely on storing recovery data on the user's devices. In these cases, our attacks are also less impactful.

4.4.1.1 *Bitwarden*

Bitwarden supports two flavours of key escrow: organisation account recovery and "Key Connector".

Organisation Account Recovery. Organisation account recovery allows the admins of organisations to reset the master password of all members who enrol in recovery.[7] Enrolling in recovery can either be a manual step requiring user interaction (so-called *self-enrolment*), or happen automatically when the user joins the organisation (*auto-enrolment*). When a user enrols (by either method) in account recovery, the client fetches the organisation public key pk_{org} from the server, encrypts the user key k_u under pk_{org}, and sends the resulting *account recovery ciphertext* c_{rec} to the server. To reset the master password of an enrolled user, an admin of the organisation can fetch the encrypted k_u from the server, decrypt it using the organisation's private key sk_{org}, choose a new master password (and consequently derive a new master key), and finally re-encrypt k_u under the new master key. We describe two key recovery attacks caused by the account recovery feature, BW01 and BW02, below.

[7] While the recovery feature is restricted to "enterprise" organisations (with a paid subscription), all Bitwarden clients include this feature's code, and our attacks succeed independently of user subscription status.

BW01: Malicious Auto-Enrolment. Recall that when a Bitwarden user joins an organisation, the client fetches the organisation's data, such as name, key material, and organisational policies, from the server. This data is not integrity protected. As a consequence, an adversary controlling the server can compromise users as soon as they accept an invitation to join *any* organisation, as follows.

When a user accepts an invitation, the client asks the server for the account recovery policy and the public-key of the organisation. The adversary replaces the organisation's real data, setting auto-enrolment to true in the policy, and replacing the public key pk_{org} with a malicious pk^{adv}_{org} for which they know the secret key sk^{adv}_{org}. Since account recovery is enabled, the client encrypts the user key k_u under the organisation public key pk^{adv}_{org}, and sends the resulting account recovery ciphertext c_{rec} to the server. The adversary decrypts c_{rec} with sk^{adv}_{org} and recovers k_u.

IMPACT. With possession of the user key, the adversary is able to read and modify all vault data of the targeted user.

REQUIREMENTS. The user joins any organisation. Note that the adversary does **not** have to control the organisation; it can be an honest invitation from a trusted enterprise or family member, for example.

NOTE. The account recovery feature can also be used by an attacker to amplify the reach of a compromise from one user to whole organisations, as follows. An organisation's private key sk_{org} is known to all the members of the organisation. It follows that once the adversary has compromised one user, they learn the private key sk_{org} of all the organisations that user is a member of. If any of these organisations enabled account recovery, the adversary then also compromises the user keys of all members of these organisations through their account recovery ciphertexts. This process can be repeated, infecting all organisations that have key recovery enabled and have overlapping members.

BW02: Malicious Key Rotation. Bitwarden allows its users to change their master password, and optionally rotate their keys. If a user decides to rotate their keys, the client samples a new user key k'_u and encrypts it under the new extended master key k'_{em} (cf. Figure 4.1).

If a user is enrolled in account recovery when they rotate their keys, the client will also have to regenerate their account recovery ciphertext. To do so, it encrypts the new user key k'_u under the organisation public key pk_{org}, and sends the new account recovery ciphertext c'_{rec} to the server.

The key point here is that pk_{org} is not retrieved from the user's vault; rather, the client performs a sync operation with the server to obtain it. Crucially, the organisation data provided by this sync operation is not authenticated in any way (see Attack BW01). This thus provides the adversary with another opportunity to obtain a victim's user key, by supplying a new public key pk^{adv}_{org} for which they know sk^{adv}_{org} and setting the account recovery enrolment to true. The client will then send an account recovery ciphertext c'_{rec} containing the new user key, which the adversary can decrypt to obtain k'_u.

Note that this attack works whether or not the user is a member of an organisation at the time of key rotation: the adversary can either use Attack BW08 or BW09 to, respectively, add the user to a newly created organisation or overwrite an existing organisation's keys.

Concretely, the forged organisation returned to the client will look like the following (where k is a symmetric key for AES-CBC-HMAC):

```
org' = {
  UserId = user.Id, // victim's user ID
  UseResetPassword = true,
  Key = RSA.Enc(pk, k), // k under the victim's public key
  PublicKey = pk^adv_org,
  PrivateKey = AES-CBC-HMAC.Enc(k, sk^adv_org),
  [...]
};
```

IMPACT. With possession of the user key, the adversary is able to read and modify all vault data of the target user.

REQUIREMENTS. The user rotates their encryption keys.

Key Connector. The second flavour of key escrow in Bitwarden is the so-called "Key Connector" feature, offered to enterprise organisations that run self-hosted Bitwarden servers as part of the support for single sign-on (SSO).[8] When an organisation starts using SSO, it delegates the authentication of users to an external identity provider (e.g., Google or Microsoft). The purpose of using SSO is to allow users to access multiple different applications with a single set of credentials. That is, it removes the need for a dedicated account (username and password) for every service. However, in the case of end-to-end encrypted applications like Bitwarden, outsourcing the *authentication* to an external identity provider is not sufficient to remove the need for dedicated Bitwarden credentials. Indeed, recall that the master key k_m, derived from the master password and email address as shown in Figure 4.1, is not just used to authenticate to the server, but also to protect the user key, which in turn is used for client-side encryption of vault items. Thus, in order to allow users of Bitwarden organisations with SSO to entirely remove their Bitwarden password, the user key must also be recoverable without the password. This is the purpose of the Key Connector (KC) application. KC is a self-hosted application which allows Bitwarden organisations using SSO to store users' user keys (k_u in Figure 4.1) such that they do not need a Bitwarden master password or master key. When an organisation starts using KC, its users are redirected to a "remove master password" page. Each user is then prompted to confirm the change; the client subsequently sends the user's master key k_m to the KC service (henceforth referred to as the "connector"), which uses k_m to decrypt k_u and stores it. We refer to this procedure as *conversion*. In attack BW03, we show that a malicious Bitwarden server can trick any client (regardless of whether the user is a member of any organisation) into starting a conversion flow which, if completed, reveals the user's master to the adversary.

[8] Just like for the account recovery feature, Key Connector is restricted to "enterprise" organisations (with a paid Bitwarden subscription), but since all Bitwarden clients contain the code implementing this feature, our attacks affect all users, regardless of subscription status.

BW03: Malicious KC Conversion. The attack works as follows. The adversary tricks the client into believing that the user is part of an organisation which is enabling KC (as described below), and supplies the client with a (hidden) malicious connector URL. The client will then display the "remove master password" page to the user (see Figure 4.5 for an example), the content of which is also partially controlled by the adversary. If the user clicks the "remove master password" button, the client sends the master key k_m to the adversarially controlled URL in plaintext.

We now explain how the adversary can force the client into taking the user to this conversion page. Each time a Bitwarden client performs a sync operation with the server, it checks if the user is logged in via SSO and a member of an organisation which has enabled KC. If so, the client determines that the user needs to perform a conversion. The attack leverages that all of these attributes can be forged by the adversary.

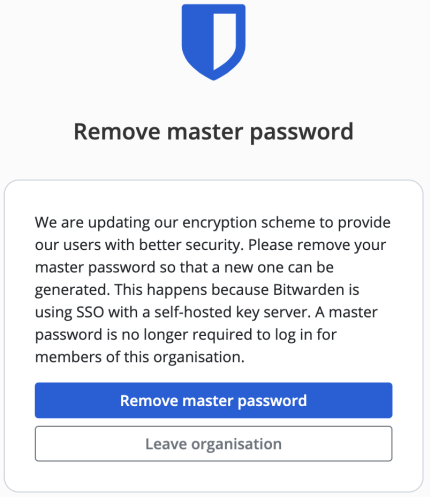


Figure 4.5: Key Connector conversion page with malicious message injected by the adversary.

As we saw in Attack BW02, the organisation data retrieved in sync operations is not integrity protected. This includes the organisation symmetric key, flags determining whether KC is enabled, and the connector URL. This lack of integrity allows the adversary to make the user a member of a newly created fictive organisation by sampling a symmetric key k'_{org} and other organisation data and supplying the client with k'_{org} encrypted under the user's public RSA key (as in Attack BW08). The adversary can also set the connector URL (to which the master key will be sent upon conversion) to an arbitrary URL of its choice.

Furthermore, clients infer whether SSO is enabled by looking at some attributes set by the server at login time. Specifically, the client believes itself to be logged in using SSO if it can find an authentication methods reference with the value "external" inside the JSON WEB Token (JWT) access token which it receives from the server when logging in. It is trivial for the adversary to forge this JWT, thus making *all* Bitwarden users vulnerable to this attack.[9]

[9] With the exception of users who are members of self-hosted Bitwarden organisations and have already performed KC conversion in the past; these users no longer have master keys, and are thus not vulnerable to the attack.

IMPACT. The user's master key is sent to an adversarially controlled connector URL. With possession of the user's master key, the adversary is able to read and modify all vault data of the target user.

REQUIREMENTS. This attack requires an element of social engineering, since the user always needs to confirm the conversion. This can be achieved in several ways. First of all, the conversion page does not contain any warning about what the process entails, and does not show the URL of the Key Connector being used. Furthermore, the adversary can control the text shown in the conversion page by injecting arbitrary data in the "organisation name" field: Figure 4.5 shows an example of how the conversion page could look during an attack.
Additionally, the adversary can disable the *"Leave organisation"* button in Figure 4.5, leaving the user no choice but to click on *"Remove master password"* in order to access their vault, as the client will persist the prompt until the user accepts the conversion.

4.4.1.2 *LastPass*

LastPass also provides two forms of key escrow: account recovery and admin *password reset*. We focus here on the second. The master password reset is a feature for LastPass' enterprise products "business" and "teams".[10] Within a team, privileged user accounts called *super admins* have the ability to reset the master password of users in the team. To this end, each admin account is associated with an RSA key pair (sk_{adm}, pk_{adm}). After a super admin chooses to reset a user's password, the client of the selected user will – upon the next login in the LastPass browser extension – retrieve the list of admins and their public keys from the server, encrypt its user key k_u under each admin's public key pk^i_{adm}, and send the resulting ciphertexts back to the server.[11]

[10] This feature is only available in the paid versions of LastPass [1], but, as with Bitwarden, the code implementing this feature is present in the client independent of their subscription status.

[11] This process is described as a "Key Exchange" by LastPass [3]. This does not correspond to the conventional meaning of the term.

LP01: Malicious Password Reset. Master password reset relies on admin public keys pk^i_{adm} being retrieved from the server. The LastPass client does not authenticate these keys in any way. It follows that an adversary can trivially inject their own pk^{adv}_{adm}, for which they know sk^{adv}_{adm}. They can then recover k_u from the ciphertext sent by the client.

Concretely, the adversary extends the login response with an additional super admin, so that it looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?><response>
   ...
   sauid$N$="$id^{adv}$" // injected super admin
   sakey$N$="hex($pk^{adv}_{adm}$)"
   ...
</response>
```

where N is an arbitrary integer, $\mathsf{hex}(\cdot)$ converts a public key to its hexadecimal representation, and id^{adv} can be an arbitrary integer value, as the client does not validate the "super-admin ID".

In theory, only users in teams where password reset is enabled and who are selected for reset should be affected by this vulnerability. In practice, however, LastPass clients query the server at each login and fetch a list of admin keys. They then send the account recovery ciphertexts independently of enrolment status.

Impact. With possession of the user key, the adversary is able to read and modify all vault data of the target user.

Requirements. The user logs in to LastPass in the browser extension client.[12]

[12] Only the browser extension client supports the "key exchange" procedure used for password reset, thus the attack works only against this client.

4.4.1.3 *Dashlane*

Dashlane offers a limited form of key escrow for users who are part of an organisation with admin-assisted key recovery. Namely, Dashlane clients encrypt some key material, including the vault key k_u, using a recovery key k_r. The resulting ciphertext is never uploaded to the server: it is stored locally on the device, e.g. in the browser.

Dashlane's key escrow only concerns this recovery key: the client encrypts k_r with a server-provided symmetric key, and then encrypts

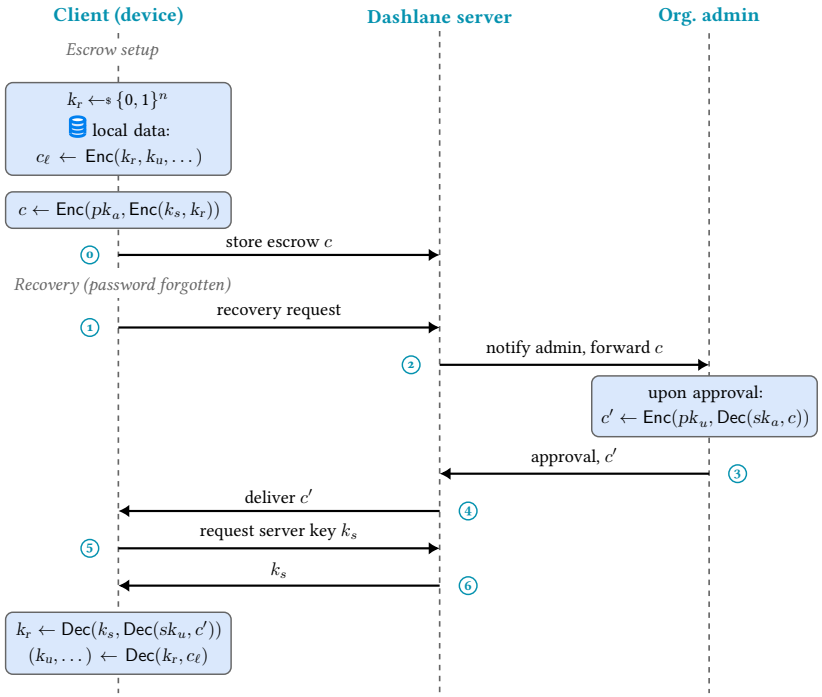


Figure 4.6: Dashlane's admin-assisted key escrow and recovery flow. During setup, the client stores the local data encrypted under the recovery key k_r (which never leaves the device), and uploads the escrowed recovery key c (step 0), wrapped first under a server-provided symmetric key k_s and then under the admin public key pk_a. To recover (1), the server forwards c to the admin (2); the admin peels off the RSA-OAEP layer and re-encrypts to the user (3, 4), after which the client retrieves k_s from the server (5, 6) and unwraps k_r.

the resulting ciphertext under the public keys of the admins in the organisation. If a user forgets their password, but they still have a device with the encrypted local data, they can initiate a recovery request. The admin will be notified of the request, and if they approve it, they will decrypt their layer of encryption on k_r, and send the intermediate ciphertext back to the user encrypted under the user's public key. The client can then decrypt this message, request the symmetric key from the server again, and decrypt the intermediate ciphertext to obtain the recovery key k_r. Finally, k_r can be used to decrypt the local data. This process is illustrated in Figure 4.6.

Note: Recovery Key Compromise. Dashlane's admin-assisted recovery flow suffers from a similar problem to the escrow protocols of Bitwarden and LastPass– namely, an adversary can hijack the admin public keys, since they are not authenticated in any way. Consequently, the adversary recovers k_r. However, in Dashlane's design, recovery requires access to the local data in the client. This data is never uploaded to the server, or transferred to an admin: an adversary cannot use k_r to further compromise users, making this attack vector ineffective.

4.4.1.4 1Password

Recall that, in 1Password, clients encrypt vault keys under the recovery group's public key. This is done *by default and unconditionally—independently of whether account recovery is actually enabled*, and without authenticating that public key.

When a fresh user account is created, the client creates a fresh user vault, and samples a fresh key pair for the recovery group. The vault

key is then immediately encrypted for this recovery group. This means that even a malicious server cannot tamper with the public key of the recovery group used to encrypt this default vault.

1P01: Malicious Recovery Group. At any later point in time, users can create additional vaults. The vault keys corresponding to these vaults are also encrypted under the recovery group's public key—but this time, the recovery group public key pk_R is fetched from the server. This public key is not authenticated. It follows that it is trivial for a malicious server to substitute pk_R with a key pk'_R for which they know sk'_R, and thus recover the encrypted vault key for the new vault.

The possibility of an attack like the one we describe is acknowledged in the 1Password white paper, even though an attack is not described in full:

> ...the potential [MitM] attack applies to any situation in which secrets are encrypted to another's public key. Thus, this applies during the final stages of recovery or when a vault is added to any group as well as when a vault is shared with an individual. This threat is probably most significant with respect to the automatic addition of vaults to the Recovery Group as described in "Restoring a user's access to a vault." [4, Appx. C]

IMPACT. The adversary can read and modify all contents of new vaults.
REQUIREMENTS. The user creates a new vault.

4.4.2 *Vault Integrity*

As we have mentioned, even though the user interacts with the abstraction of a "secure vault", the individual items and the user settings that comprise a vault are managed and (typically) encrypted independently. Hence, vaults are not monolithic cryptographic objects.

If the vault were monolithic, vault integrity would be easy to achieve by using authenticated encryption (AE). However, every minor change would necessitate re-encrypting the entire vault, making its maintenance and synchronisation with the server costly. Thus, a common solution is instead to encrypt each item separately with AE under a single key. But this does not necessarily result in integrity guarantees for the whole vault. For example, items could be swapped around or removed entirely without the client being able to detect it.

The attacks that follow all exploit the lack of *vault integrity* for the vendors we study. Not only can the adversary swap and delete items, they can also change metadata, settings, or the organisations to which a user belongs.

4.4.2.1 *Bitwarden*

In Bitwarden, items and user settings are not uniformly protected. Rather, vault items are encrypted individually, and many settings are not en-

crypted at all. This also holds true for items in organisations and organisation-wide settings.

Each item is an object with various fields, such as item name, password, username, and URLs. All of these fields are individually encrypted using AES-CBC-HMAC. Up until version `2024.2.0` of Bitwarden, the fields of all items were encrypted directly with the user key k_u. As of `v.2024.2.0`, "per-item keys" are used instead: a fresh key k_u^i is sampled and used for each new vault item. The item key is itself encrypted with the user key k_u, and added as a field to the item. Note that old items were not migrated to this new format.

The client retrieves vault items from the server via a sync request; the server will return the item encoded in a JSON object, with the (unencrypted) field names as keys, and the respective ciphertexts as values.

BW04: Unprotected Item Metadata. A surprising number of item metadata fields are neither encrypted nor integrity protected. These include the type of items (e.g. login, card, secure note, etc.), their creation date and whether they require a password reprompt.

An adversary can arbitrarily read and modify these fields. This can leak information about the content of the vault. This attack has no requirements.

BW05: Item Field Swapping. Nothing binds a ciphertext to a particular item field. Hence, an adversary can swap around ciphertexts from different fields within an item.

For all items created before the per-item keys feature was introduced, an adversary can even swap ciphertexts around among different items, as well as create new items by merging fields from other items.

This attack has no requirements, and represents a clear violation of vault integrity. It also functions as a building block for other attacks.

BW06: Icon URL Item Decryption. Items can include a URL field, which is used to autofill the credentials and display an icon on the client. The client decrypts the URL and fetches the icon from the server, including in its request the domain and top-level domain of the URL. For instance, if the URL is "https://host.tld/path", the client request includes "host.tld".

This means that the adversary can learn (part of) the contents of URL fields. Using Attack BW05, an adversary can place the ciphertext of sensitive item fields, such as a username or a password, in the encrypted URL field.[13] After fetching the item, the client will then decrypt the ciphertext, confusing it for a URL. If the plaintext satisfies some conditions (i.e. containing a '.' and no '!'), it will be leaked to the adversary.

A URL checksum feature was deployed in July 2024, making the clients store a hash of the URL in another encrypted item field, therefore providing a rudimentary integrity check and preventing this attack. Note that old items are never updated to add such a checksum: this feature only protects items created after its introduction. Furthermore, URL checksums are only checked if a per-item key is present for the item. As

[13] Note also that in the item JSON representation, the values (e.g., the username or the password) are encrypted, while the field names are in plaintext. The adversary can therefore tell, without any decryption, which ciphertext corresponds to which field (e.g., the password).

we will see, an adversary can prevent per-item keys from being enabled with Attack BW10.

IMPACT. The adversary can recover selected target ciphertexts in the item, such as the username or the password.

REQUIREMENTS. The user opens a vault containing items that do not use per-item keys (i.e., items created before July 2024, or after Attack BW10 is run). The target plaintext must satisfy some additional conditions, as follows:

1. (It does not contain the string "`://`" and contains the character '`.`') or (it starts with the string "`http`" and it contains the character '`.`' and it does not start with either of the following strings "`data:`", "`about:`" and "`file:`").
2. It does not contain the character '`!`'.

Lastly, the sensitive data inserted will be passed to the `getHostname` function of the `tldts` library. As a consequence, some parts of sensitive data might be removed from the final endpoint query.

BW07: Remove KDF Iterations. Client authentication to Bitwarden is performed by checking if a hash of the master key h_m, derived from the master password input by the user at login time, matches a hash stored on Bitwarden's servers. In order to protect the master password from brute-force attacks, computation-intensive Password-Based Key Derivation Functions (PBKDFs) are used to compute this hash. The derivation speed is determined by the parameters of the PBKDF.

For instance, when PBKDF2 is used with the default 600,000 iterations, the hash is obtained as follows:

$$k_m \leftarrow \mathsf{PBKDF2}(pw, salt = email, i = 600000)$$
$$h_m \leftarrow \mathsf{PBKDF2}(k_m, salt = pw, i = 1)\ .$$

The KDF settings are stored unauthenticated and unencrypted on the server. When a target user logs in, the adversary can choose an arbitrary PBKDF configuration, including one with a minimal iteration count. The user will then send a master key hash that is efficient to compute during authentication. In our example, this means that the adversary can set the first iteration count to 1,[14] thereby reducing the number of PBKDF2 iterations from 600,001 to 2 and achieving a 300,000x speed-up in a brute-force attack.

[14] The minimum, values of zero or less result in the client crashing.

IMPACT. The adversary can reduce the cost of a brute-force attack on a target user's master password; successfully recovering the password enables full control of the user's account.

REQUIREMENTS. The user logs in.

4.4.2.2 LastPass

Similarly to Bitwarden, vault items in LastPass are composed of various fields such as item name, username, URL, and password, and all of these fields are individually encrypted under the same user key k_u. Similar attacks therefore apply.

LP02: Item Field Swapping. There is no associated data binding each ciphertext to its field. Hence, an adversary can swap around ciphertexts from different fields within an item, as well as between different items, and create new items by merging fields from other items.

LP03: Icon URL Item Decryption. LastPass login items can include a URL. This is used to autofill the credentials and display an icon on the client. In order to fetch these icons, the client makes a query to LastPass's icon server, including in the request the URL that was stored inside the item. Just like in Bitwarden, an adversary can use the item-field swapping attack (Attack LP02) and place the ciphertext of sensitive item fields, such as the username or the password, in the URL field. The plaintext will be leaked to the adversary via the query to the icon server. Note that, unlike Bitwarden, there are **no conditions** on the plaintext formatting.

IMPACT. The adversary can recover (parts of) selected target ciphertexts in the item, such as the username or the password.

REQUIREMENTS. The client opens the modified vault.

LP04: Remove KDF Iterations. Client authentication to LastPass is performed by checking if a hash of the user key – derived from the master password input by the user at login time – matches a hash stored on LastPass's servers. Just like in Bitwarden, 2 invocations of PBKDFs are used to derive this hash, with a default of $600,000 + 1$ iterations for PBKDF2.

Similarly to Bitwarden, the KDF parameters are unencrypted and unauthenticated, enabling an adversary to choose them arbitrarily. The PBKDF2 iteration count can then be lowered down to 3 (a client-enforced minimum of 2 for the inner invocation, while the outer is fixed to 1). The adversary thus achieves a speed-up of 200,000x in a brute-force attack.

IMPACT. The adversary can reduce the cost of a brute-force attack on a target user's master password; successfully recovering the password enables full control of the user's account.

REQUIREMENTS. The user logs in.

LP05: Malleable Vault. All field items are encrypted separately under a single user key using AES-256-CBC (without integrity protection). An adversary can manipulate the content of the encrypted fields using standard CBC-mode bit-flipping techniques. This attack has no requirements.

LP06: Unprotected Item Metadata. As in Bitwarden, many item metadata fields are unencrypted and not integrity protected: this includes creation timestamp, modification timestamp, and master password reprompt fields. An adversary can change the metadata for an item, including whether or not to reprompt the user for their master password, the item creation and modification timestamp, its folder, and whether the item is in the favourites. They can also enumerate the vault items and their type, such as login credentials, credit card, secure note

etc., and observe the victim's activity by looking at the item creation and modification time. This attack has no requirements.

4.4.2.3 *Dashlane*

Recall that items in Dashlane are treated as transactions and encrypted with a key derived from the master key based on transaction parameters. For all transactions sharing the same parameters (in particular, KDF and cipher mode; the salt is constant for all transactions) the same derived key is used.

DL01: Transaction Replay. Since the transaction key does not depend uniquely on the transaction, the server can duplicate, reorder or drop transactions without the client noticing.

This attack trivially violates vault integrity. It also serves as a building block for other attacks, see Section 4.4.4.2.

4.4.2.4 *1Password*

Vaults in 1Password are encrypted at a per-item granularity: each item is encrypted as a AES-GCM blob, with the metadata ("item overview") encrypted as a separate AES-GCM blob. This reduces the amount of metadata leakage and integrity issues, compared to Bitwarden and LastPass; similarly to Dashlane, all items in a vault are encrypted with the same symmetric vault key k_v, and the only plaintext metadata is the item creation and last modification time. Account metadata such as the KDF parameters are stored unencrypted and unauthenticated.

1P02: Item Dropping/Duplication. Just like for Dashlane, vault items can be duplicated or dropped by the server. This attack trivially violates vault integrity, but does not have a severe practical impact.

1P03: Vault Injection. Due to the lack of ciphertext authentication, it is easy for an adversary to inject an entirely new vault, containing adversary-controlled items, into a 1Password user's account. The attack proceeds exactly as Attack 1P06, which we will see in Section 4.4.3.4, but creating a new vault instead of targeting an existing one.

1P04: Remove KDF Iterations. The KDF parameters that control the computational cost of deriving the AUK (k_{AUK}) from the master password and secret key are provided by the server and trusted without client-side verification. A malicious server can reduce the iteration count from the default 650,000 iterations to a minimal value of 10,000 iterations.

IMPACT. Although this attack weakens the brute-force protection provided by slow hashing, the secret key still contributes enough entropy to make a brute-force attack infeasible.

4.4.3 *Sharing*

These attacks target the *sharing* of items. All of the password managers we study implement some kind of sharing, and they fall victim to attacks due to a lack of key authentication. They all also lack any mechanism for

end users to manually authenticate keys out-of-band (e.g. by comparing key fingerprints) when sharing, leaving their users completely exposed to this class of attacks.

The attacks in this category range in severity: while the vulnerabilities in LastPass and Dashlane only affect the items being shared, the ones in Bitwarden and 1Password have further-reaching consequences.

4.4.3.1 Bitwarden

Recall that in Bitwarden, users can share items with other users in organisations; each organisation has a symmetric key k_{org}, which is stored on the server encrypted under the public RSA key of each member of the organisation.

BW08: Organisation Injection. There is no cryptographic binding between a user's vault (or even the vault abstraction) and the organisations a user is part of. Users learn the list of the organisations they are part of when their client performs a sync to fetch the latest data from the server. Furthermore, copies of the encrypted organisation symmetric key are not authenticated in any way: anyone can encrypt a chosen symmetric key under the public key of any user.

It follows that adding a user to an organisation only requires access to their public key, which is known by the server. Thus, an adversary can create a new organisation and trivially add users to it: they just need to encrypt the organisation symmetric key k_{org} under the target user's public key. The client will silently accept the new organisation at the next sync. As a corollary, the adversary can also add arbitrary users to any organisation whose symmetric key they know.

Concretely, the adversary samples a fresh organisation symmetric key k'_{org} and organisation RSA key pair (pk'_{org}, sk'_{org}), and forges a sync response that injects a new organisation, together with its default *collection* (the grouping Bitwarden uses for an organisation's shared items):

```
syncRes' = {
  newOrg = {
    Key = Enc(pk, k'_org), // org key under victim's pk
    PublicKey = pk'_org,
    PrivateKey = Enc(k'_org, sk'_org), // org priv.  key under the org key
  },
  collections = {
    newColl = { Name = AES-CBC-HMAC.Enc(k'_org, name) } },
  [...]
};
```

Impact. The adversary can add users to arbitrary organisations, thus violating vault integrity by adding organisation items to the user's vault. This can be used in framing attacks, for example, by planting incriminating material. Furthermore, it enables several other attacks (e.g., BW03 and BW09).

REQUIREMENTS. The client performs a sync operation. This is done automatically by the client at startup, when the vault page is loaded, in response to some user actions, and upon server-generated events.

BW09: Organisation Overwrite. Immediately after an organisation is created, the client of the creator performs a sync to fetch the latest data from the server. As part of this, the keys of the newly created organisation are fetched. After the sync is completed, the creator of the organisation is able to decrypt the organisation symmetric key using their private RSA key and use it to decrypt all organisation data.

By applying the same method as in Attack BW08 above, the adversary can overwrite the newly created organisation: they can swap the encrypted symmetric key c_{org} and the organisation data with values they control. Upon sync, the client has no way of distinguishing such a malicious key from the legitimate organisation key.

In contrast to Attack BW08 (which creates a new organisation), this attack is entirely undetectable. Note that it is also possible for an adversary to overwrite an organisation after it is created, but in doing so they would delete any pre-existing organisation items.

IMPACT. The adversary can read and modify all the items in the newly created organisation.

REQUIREMENTS. The user creates an organisation.

4.4.3.2 LastPass

Recall that LastPass implements item sharing by means of shared folders. Items in a shared folder are encrypted with the symmetric shared folder key, which is in turn encrypted under the public key(s) of the recipient(s) of the folder.

LP07: Sharing Key Overwrite. During an item-sharing operation initiated by a user, the server, acting as an adversary, sends an adversary-controlled recipient RSA public key instead of the intended recipient key.

IMPACT. The adversary can read and modify all the items in a shared folder.

REQUIREMENTS. The user shares an item with any recipient.

4.4.3.3 Dashlane

Like LastPass, Dashlane implements item sharing using public key encryption. The problem of unauthenticated public keys described in Attack LP07 also affects Dashlane.

When a client shares an item with a user or in a group, a fresh symmetric sharing key (k_{shr}) is sampled. This key is either used to directly encrypt the shared item (in the case of user-to-user sharing), or to encrypt group keys which in turn protect shared items (for group sharing). Regardless of setting, the sharing client retrieves the RSA public keys of the recipient(s) from the server and, without verifying their authenticity,

encrypts k_{shr} under each recipient's public key, yielding c_{shr}^{user} for each recipient $user$. The client sends each c_{shr}^{user} to the server for storage.

DL02: Sharing Key Overwrite. When the sharing client requests a recipient public key, a malicious server can overwrite it with an RSA key pk^{adv} it controls, causing k_{shr} to be encrypted under the malicious public key. The resulting ciphertext c_{shr}^{adv} is then sent to the adversary, allowing them to decrypt it with sk^{adv} and recover k_{shr}. This in turn gives the adversary access to the shared item(s) protected by k_{shr}.

IMPACT. The adversary can read and modify all shared items.

REQUIREMENTS. The user shares an item with any recipient.

4.4.3.4 1Password

1Password supports sharing vaults. This is implemented by encrypting a vault key under the public key of the user with whom the vault is shared.

1P05: Sharing Key Overwrite. Much like the other products we analyse, 1Password lacks authentication of public keys. This trivially enables sharing attacks similar to BW09, LP07 and DL02, something that the 1Password white paper [4, Appx. C] openly acknowledges:

> At present, there's no robust method for a user to verify the public key they're encrypting data to belongs to their intended recipient. As a consequence, it would be possible for a malicious or compromised 1Password server to provide dishonest public keys to the user and run a successful attack.

We list this known attack here for completeness. Unlike the other vendors we study, 1Password explicitly acknowledges some of the attacks stemming from public key authentication. Nonetheless, these attacks violate security expectations in a malicious server threat model.

1P06: Vault Substitution Attack. 1Password not only lacks authentication of public keys, but also of public-key ciphertexts. This affects not only the security of the credential-sharing feature, but also the confidentiality of the entire vault.

Recall from Figure 4.4 that 1Password uses RSA-OAEP to encrypt the vault key with the user's keyset. While this protects the confidentiality of vault keys, it does not authenticate them, creating an opportunity for a ciphertext substitution attack. In particular, anybody can **encrypt** to the public key of a given keyset, since the public part is not protected.

For instance, a malicious server can execute the following attack at the time of vault creation:

1. The user initiates vault creation, generating a keyset (sk, pk) and a vault key k_v, and saves the encrypted vault key $c = \mathsf{RSA\text{-}OAEP}.\mathsf{Enc}(pk, k_v)$. The user logs out.
2. The server samples a fresh vault key k'_v, and encrypts k'_v under the public RSA key in the keyset: $c' = \mathsf{RSA\text{-}OAEP}.\mathsf{Enc}(pk, k'_v)$.

3. The user logs in again, and is offered c' instead of c by the server. The user decrypts c' with their private key and uses k'_v to encrypt all new vault items. Since the server knows k'_v, it can recover all newly encrypted items as well as inject new items into the vault.

The root cause of this attack is the lack of cryptographic binding between the vault key k_v and the user's key encryption key derived from the password. More specifically, there is no mechanism allowing the client to verify that a vault was genuinely created by the user themselves rather than injected by the server. Unlike Attacks 1P01 and 1P05, this attack is not mentioned in the 1Password white paper [4].

IMPACT. Complete compromise of vault confidentiality and integrity. The adversary can read and decrypt all vault contents encrypted after the attack, including passwords, credit card information, secure notes, and other sensitive data stored in the vault. Similarly, they can inject new items into the vault after the attack.

REQUIREMENTS. The client fetches key material from the server, for example due to the user logging in on a new device. If executed on a non-empty vault, the attack results in the client losing access to all items already in their vault, while leaking any new items added to the vault after the attack took place. If the attack is executed at the time of vault creation, the attack is effectively undetectable by the client, since it cannot distinguish between a ciphertext it created and the ciphertext created by the server during the attack.

4.4.4 Backwards Compatibility

The KDF downgrade attacks for Bitwarden, LastPass and 1Password (BW07, LP04, 1P04, which we classified in this work as vault integrity attacks) all also arguably represent backwards compatibility issues. Password managers select more computationally intensive KDF parameters over time, but want clients of users who have not migrated to the new parameter sets to still be able to access their vault.

This section focuses on some more involved backwards compatibility features of Bitwarden and Dashlane. Both password managers have evolved the cryptographic primitives they support over the years, shifting from AES in CBC mode for encryption and PBKDF2 for key derivation to AE schemes such as AES-CBC-HMAC and memory hard PBKDFs like Argon2d. Both also show changes in the key derivation trees, with Bitwarden migrating to better key separation with the introduction of per-item keys.

The current client versions only use modern cryptography by default, but they still offer "legacy" support. The attacks described below enable the adversary to downgrade, sometimes permanently, the primitives used by the clients, reducing the security of the affected password managers to the security of the weakest supported cryptographic algorithm.

4.4.4.1 Bitwarden

Bitwarden offers several backwards compatibility features, which can all be exploited by a malicious server.

Per-Item Keys. Recall from Section 4.4.2.1 that Bitwarden version `2024.2.0` enables "per-item keys". When creating a new item, per-item keys are only used if the client enables this feature, and if the server version is recent enough to support per-item keys. In order to determine this, the client fetches the server version at login time by sending a request to the `config` endpoint of the server. This result in the attack BW10 below.

BW10: Disable Per-Item Keys. An adversary can perform a downgrade attack by sending a lower server version such that the client does not enable per-item keys. This is trivial, because the server configuration is not authenticated by the client. As a consequence, all new items will be encrypted with the same key (the user key).

IMPACT. This attack disables a critical security feature, downgrading Bitwarden's key hierarchy and exposing unprotected vault items to numerous other attacks. Recall that Bitwarden does not implement a migration path for legacy items, thus indefinitely exposing affected items even after the attack is concluded.

REQUIREMENTS. None.

Legacy encryption. The latest Bitwarden clients use authenticated encryption (AE) for all symmetric encryption operations by default, thereby ensuring both confidentiality and integrity of the encrypted data. The AE scheme used is AES-CBC with HMAC, combined via the encrypt-then-MAC paradigm. In particular, this encryption scheme is used to decrypt the user key received from the server at login time and to encrypt and decrypt the items inside the vault.

However, older versions of Bitwarden used AES-CBC (without HMAC), which does not offer any integrity protection. To accommodate both formats, each ciphertext has an encryption type attribute (type *2* for AE, type *0* for plain AES-CBC). The encryption type is prepended to the ciphertext and is not integrity-protected.

Furthermore, older versions of Bitwarden did not use the current key hierarchy. To support the old hierarchy, a modern client will generate a new RSA key pair for the user (pk', sk') if one was not provided by the server, and use k_m to encrypt sk' if no user key ciphertext c_u was provided by the server. Both of these features can be seen as backwards compatibility measures, allowing a modern client to support old user vaults and to migrate the user vault to the current feature set.

We exploit these features to build the attack gadget BWb1, resulting in the attacks BW11 and BW12 below.

BWb1: Known Plaintext Oracle Gadget. An adversary can leverage the generation of RSA key pairs and the support for encryption using

the master key k_m to obtain AES-CBC encryptions of known plaintexts under an (unknown) k_m.

At login time, the adversary responds to the login request with a null c_u and a null private RSA key ciphertext c_{sk}. The absence of c_{sk} will force the user to generate a new RSA key pair. However, since c_u was also set to null, the client will use k_m to encrypt the new secret key using AES-CBC, yielding c'_{sk}, and will then send pk', c'_{sk} to the server.

Notably, sk' also contains an encoding of pk'. Thus, by observing the encrypted c'_{sk} and pk', the adversary learns a 5-block-long plaintext-ciphertext pair.

BW11: User Key Overwriting. The adversary can forge a ciphertext c_u^{adv}, such that the client will decrypt it to a user key $k_u = k_u^{enc} \parallel k_u^{mac}$ for which the adversary knows k_u^{enc}. Recall that k_u^{enc} and k_u^{mac} are 32 bytes each, spanning two blocks of CBC plaintext.

The adversary constructs c_u^{adv} by using the 5-block plaintext-ciphertext pair from BWb1. Via bit-flipping, the adversary sets the last block of ciphertext to a block P_5 of padding, obtaining a garbled block P_4, and known blocks P_1, P_2, P_3. Finally, the adversary marks this ciphertext as type 0.

Upon receiving c_u^{adv}, the client will accept it as a valid user key encryption: the ciphertext is encrypted under the master key k_m, the type is 0 so CBC decryption is used, and unpadding succeeds. The client then parses $k_u = k_u^{enc} \parallel k_u^{mac}$. Since k_u^{enc} consists of blocks P_1 and P_2, the adversary knows k_u^{enc} fully, while k_u^{mac} is unknown to the adversary because of the garbled block P_4.

Impact. The adversary learns k_u^{enc}, used to encrypt all the vault items. Therefore, the confidentiality of all items encrypted after the attack is mounted is forfeit. Integrity is preserved, as the adversary does not learn k_u^{mac}.

Requirements. The user logs in. Due to the attack, this initial login attempt fails, and the user is redirected to a migration page. Regardless of the action taken by the user after the migration page is displayed, the attack completes successfully during a subsequent login attempt.

BW12: Downgrade to Legacy. The adversary can forge a ciphertext c_u^{adv}, such that the client decrypts it to a user key $k_u = k_u^{enc} \parallel k_u^{mac}$, where k_u^{mac} is empty.

This can be achieved by following the same steps as in Attack BW11, but using only three blocks of ciphertext: in this case, P_3 is set to a block of padding, and P_2 is garbled. As previously discussed, the client will accept this c_u^{adv} as valid. The ciphertext being too short triggers another backwards compatibility feature: the client parses $k_u = k_u^{enc}$ without the "mac" key part, causing all subsequent encryptions to be performed using CBC-only mode. Note that if per-item keys are enabled, item encryption still uses AES-CBC with HMAC. But the adversary can easily downgrade the client to not use per-item keys, see BW10.

Impact. The attack downgrades encryption to CBC-only mode, trivially compromising the integrity of all items encrypted after the attack is

executed. In turn, this enables a padding oracle attack [284], leading to a complete compromise of vault confidentiality. We did not develop a PoC for this last aspect of the attack, but the developers of Bitwarden acknowledged its feasibility. We also note that the adversary knows P_1, and therefore learns half of the encryption key k_u^{enc}.

REQUIREMENTS. The same as BW11.

4.4.4.2 *Dashlane*

Recall from Section 4.3.3 that every transaction has a *payload*, which defines the key derivation function, cipher, and mode to use to decrypt the transaction content. Two payload formats are relevant for our attacks: KWC3, using PBKDF2-SHA-1 and AES-256-CBC; and Flexible, with a format string which is parsed to select the KDF (PBKDF2-SHA-256 or SHA-1, Argon2i, or the default Argon2d), its parameters, the cipher (only AES-256), and the cipher mode (the default CBC-HMAC and GCM being the only documented modes).

Modern Dashlane clients default to Flexible mode; upon detecting a KWC3 payload they automatically migrate the transaction to Flexible. Support for the legacy format varies across clients: we verified that the web extension supports it, while the CLI client does not. All of the following attacks only affect clients that support the legacy format. We produced PoCs against the web extension client.

DLb1: CBC Padding Oracle. Unexpectedly, CBC-only encryption is still accepted as an encryption mode in the Flexible payload format of Dashlane and does not trigger a migration. Upon receiving a malformed transaction content ciphertext, the client will emit distinguishable error messages, depending on whether the padding removal fails or if the padding removal succeeds but parsing of the transaction content fails at the application level. This means that an adversary can mount a padding oracle attack [284] using transactions with the modern payload format and CBC-only encryption.

For the attack to work, two conditions must be satisfied. First, due to the error handling code in the client, the transaction type must be Setting. Second, the transaction must use the Flexible payload format with CBC-only set as the cipher mode. Note that Dashlane clients use a separate key k_{cbc} for CBC-only transactions: this means that the padding oracle attack cannot be used to decrypt any modern CBC-HMAC transactions, since they use key $k_{cbc\text{-}hmac}$ instead. Nonetheless, it can be used as a building block for other attacks.

REQUIREMENTS. This attack requires 256 oracle queries per byte of plaintext recovered. The client fetches new transactions (sync) every 5 minutes, meaning that adversary can decrypt one full block in 14 days. Note that this attack can be significantly sped up by more frequent syncs: this can be achieved for instance in the presence of multiple clients.

DLb2: Encryption Oracle. While modern Dashlane clients do not themselves produce transactions vulnerable to the padding oracle attack described above, the adversary can now use the encryption oracle tech-

nique of Rizzo and Duong [110, §3] to create a ciphertext that decrypts to a plaintext of their choosing under the key k_{cbc}.

REQUIREMENTS. This gadget requires running Attack DLb1 once per block of plaintext encrypted.

DL03: Item Injection. Using DLb2, an adversary can encrypt arbitrary plaintext under the key k_{cbc}.

An adversary can then forge arbitrary transactions with Flexible payload, and CBC-only cipher mode. Since the type is a plaintext field of the transaction and does not influence key derivation, the adversary can pick an arbitrary transaction type. The client will accept this transaction as authentic, and correctly derive key k_{cbc} to decrypt it.

IMPACT. The adversary can forge arbitrary transactions. This means that the adversary can now plant arbitrary credentials and notes into the vault, thus violating vault integrity, and change arbitrary settings, enabling further attacks.

REQUIREMENTS. Execution of DLb2. Recall that each block of transaction content needs 14 days to be encrypted.

DL04: Remove KDF Iterations. User settings in Dashlane control many client behaviours. Among these, they define what KDF to use for all new transaction payloads, and the respective parameters.

Using Attack DL01, the adversary can remove any existing transaction of type Setting. This moves the user settings to a known clean state. Then the adversary uses Attack DL03 to forge a new transaction of type Setting. When the user creates a new item, the client will derive a key from the master password with the adversary-chosen KDF parameters.

For instance, the adversary can change the number of PBKDF2 iterations from the default of 200,000 iterations to 1, achieving a 200,000x speed-up in a brute-force attack.

IMPACT. The adversary reduces the cost of a brute-force attack on a target user's master password; success enables full control of the target user account.

REQUIREMENTS. Execution of DL01 and DL03.

DL05: CBC-Only Downgrade. The setting transaction also controls which cipher and mode of operation the client uses for future transactions. Similarly to Attack DL04, the adversary can forge settings instructing the client to use CBC-only encryption. When the user creates new items, the client will then always use CBC-only mode. The resulting transaction will be vulnerable to the padding oracle attack in DLb1.

IMPACT. The adversary can decrypt all the transactions produced after the attack is completed.

REQUIREMENTS. Execution of Attacks DL01 and DL03, plus one execution of DLb1 for each decrypted transaction.

Attack timing. In our PoCs, we successfully forge a minimal transaction of type Setting which is only 9 blocks long, enabling us to test Attack DL04 and DL05. We artificially sped up the execution for testing,

but we estimate this would take more than 126 days at the normal attack speed. This is a long time, but it is reasonable for a targeted attack.

DL06: Lucky 64. Another Flexible payload setting allows for the client to completely skip key derivation (noderivation), and use the master password directly, if and only if the master password is exactly 64 bytes long. We presume that the intended use of the noderivation mode is for the setting where the user authenticates via SSO, and the master password contains a 64-byte cryptographic key.

In the rare instance of a user actually having picked a master password that is 64 bytes long, the adversary can use Attack DL04 to pick noderivation as a PBKDF and AES-CBC-HMAC as a cipher mode of operation. Then the master password will be split in two: the first 32 bytes will be used as the AES-256 key, and the next 32 bytes will be used as the HMAC key.

IMPACT. The adversary can reduce the cost of a brute-force attack on a target user's master password, via separate brute-force attacks on the two 32-byte halves of the user's master password. For instance, a 64-byte password composed of 10 words has an estimated entropy of 120 bits, but would require on average 2^{60} guesses to recover after the attack, compared to 2^{119} otherwise: a speed-up of a factor 2^{59}.

REQUIREMENTS. Execution of Attack DL04, plus the user selects a (memorable, low entropy) master password of exactly 64 bytes.

4.5 MITIGATIONS

Our attacks can all be mitigated using a combination of authenticated encryption (AE), key separation (KS), plaintext authentication (AD), public key authentication (PKA), and ciphertext authentication (SC). We detail how in Section 4.5.1; Table 4.1 additionally maps attacks to mitigations.

In general, these techniques can be used to design the core of a secure E2EE password manager: in Section 4.5.2 we briefly sketch such a high-level design.

4.5.1 Immediate Remediations

In our ongoing discussions with the vendors, the introduction of breaking changes emerges as the major obstacle to deploying our countermeasures. As we argue in Section 4.8, breaking changes are necessary, and even inevitable. To this aim, we propose the use of specialized password manager clients, with no functionality besides implementing a forced migration to the new vault format. This would prevent any user from losing access to their data, while preserving security for the entire user base. Below, we detail the changes we suggest.

Only use AE (AE). Disabling legacy support for encryption schemes that provide no integrity immediately remediates most of our Backwards Compatibility attacks on Bitwarden (BWb1, BW11, BW12) and Dashlane (DLb1, DLb2, DL03, DL05, DL06), since they exploit the support of AES-

CBC. Encrypting item metadata with AE would remediate BW04 (Unprotected Item Metadata). LastPass would need to introduce AE from scratch. This would remediate many of its Vault Encryption issues (LP05, LP06), and, together with the KS mitigation (see below), also attacks LP02 (Item Field Swapping) and LP03 (Icon URL Item Decryption).

Additionally using authenticated data (AD) to protect metadata fields in the items and user settings in the vault would mitigate the Vault Integrity issues in Bitwarden (BW05, BW04, BW06), LastPass (LP02, LP03, LP06), and Dashlane (DL01).

Proper key separation (KS). Most vault integrity issues can be mitigated by using proper key separation: each key should be used only to encrypt a single field of a single vault item, with all keys being created from the master key using a suitable key derivation function and per-item and per-field context strings. Recall that, in contrast to encrypting the whole vault as a monolith, this approach allows efficient synchronisation of changes: each item can be encrypted and synchronised independently, while still protecting the overall integrity.

In Bitwarden, item keys provide a good starting point. Extending this approach to the field level would both make URL checksums redundant, and remediate BW05 (Item Field Swapping) and BW06 (Icon URL Item Decryption). Item keys should also be made mandatory, mitigating BW10 (Disable Per-Item Keys). Together with the AE mitigation, KS would also prevent issues like BWb1 (Known Plaintext Oracle Gadget) from arising. LastPass currently lacks any form of key separation within the vault. Introducing it along with an AE scheme would mitigate LP02 and LP03. In Dashlane, each transaction could be encrypted with a different key, derived according to a unique transaction identifier, to partially resolve DL01.

Authenticate all plaintext (AD). All settings and metadata that cannot be encrypted should at least be authenticated (i.e., integrity protected). This can be achieved by using Authenticated Encryption with Associated Data (AEAD), as an extension of AE. This would mitigate the metadata malleability issues (BW04, LP06).

Of course, authenticated encryption of individual items does not grant integrity to the full vault. As we saw, attacks like DL01 and 1P02 allow an attacker to duplicate or drop items at will within a vault. In order to resolve these attacks, the full vault state should be authenticated: a MAC would ensure that the vault structure cannot be changed without possession of the right key. This would only offer protection up to rollback attacks: the adversary can always present an old version of the vault as the current one, unless the vault key changes because of updates to the KDF parameters or to the master password. Nevertheless, vault integrity up to rollbacks would be a significant improvement over the current integrity guarantees. As far as we are aware, such rollback attacks cannot be prevented without either assuming that clients maintain some state, or adding some additional trust assumptions (e.g. relying on a decentralized ledger).

Similarly, including the last known server version in the authenticated vault state mitigates BW10 (Disable Per-Item Keys). An attacker can still replay an old version of the vault (with the corresponding server version), but all newly created items are protected.

Protect KDF Parameters (KDF). Authenticating security-critical user settings like PBKDF parameters (such as the iteration count, which enables attacks BW07, LP04, 1P04) is highly nontrivial. Our suggestion is to err on the side of security, break backwards compatibility for very old parameter sets, and only allow secure parameter choices in the clients. A special standalone version of the client application, with no functionality other than migrating to more modern parameter sets, can be provided for users that missed the parameter upgrades.

As an alternative mitigation, we may try to authenticate the KDF parameters themselves. The client would, for instance, use the server-provided KDF parameters to derive the authentication key, use it to verify the integrity of the parameters themselves, and – in case of a mismatch – abort before any further communication with the server. On the one hand, this thwarts our immediate attacks: an adversary tampering with the KDF parameters would need to predict the authentication key for the client to accept the modified parameters. On the other hand, as pointed out in Section 3.4, the security in this setting is circular and relies on non-standard properties of MACs. Furthermore, while this solves KDF downgrades in isolation, it introduces a complication for user authentication: the server should not reveal the authenticated KDF parameters to unauthenticated users, yet user authentication itself requires knowing the KDF parameters. This circular dependency is not impossible to resolve. For instance, unlike vault data (which requires client-side re-encryption), the server could unilaterally upgrade the authentication material, allowing clients to only support the latest parameter set for authentication while still accepting older sets for vault decryption. However, such schemes are non-standard and would require further analysis.

Public key authentication (PKA). Introducing proper authentication of public keys is non-trivial: a stop-gap mitigation would be storing known public keys in the vault after an (optional) out-of-band verification. This would allow the users to detect public key changes, preventing active attacks. Even when no out-of-band verification is performed, this would still prevent some attack scenarios: the attacker can then only substitute public keys on their first use.

An alternative is to introduce a PKI where a Certificate Authority (CA) signs all public keys. This CA should not be under the control of the same entity that controls the password manager's server, so that it can truly act as a trusted third party. This is particularly relevant in a business setting, given that many companies that use password managers may already operate their own CA infrastructure.

These measures would remediate most of the Sharing issues (LP07, DL02, 1P05). They would also allow the user to authenticate the public keys of admins and organisations used for Key Escrow. Additionally,

assuming that a correctly signed list of admins is published, this would remediate several Key Escrow issues (BW01, BW03, LP01).

Ciphertext authentication (SC). In Bitwarden, the only cryptographic check of organisation membership is through the decryption of organisation data, encrypted by the user under their own public key. Similarly, in 1Password, vault keys are encrypted by the user under their own public key. However, as we have noted, anyone can produce such ciphertexts. This results in BW02 (Malicious Key Rotation), BW08 (Organisation Injection), BW03 (Malicious KC Conversion), and 1P06 (Vault Substitution Attack).

While it is true that authenticating public keys of other users is non-trivial, it is easy for a user to verify whether a public key ciphertext has been produced by themselves: these attacks can be prevented by switching to signcryption [293]. By signing plaintexts before encryption, clients can verify that the user themself is the sender. An alternative would be to forgo public-key encryption entirely in favour of symmetric (authenticated) encryption. This is possible since the user is both the sender and the recipient of the affected data. An added benefit of this approach is efficiency gains from removing unnecessary public-key cryptography. However, it might require a larger re-design compared to simply adding signatures.[15]

A combination of signcryption and PKA is needed when receiving public key ciphertext encrypted by other users, such as organisation invites in Bitwarden, or the list of admins for a team in LastPass. This would be needed to remediate BW01 (Malicious Auto-Enrolment), BW09 (Organisation Overwrite) and LP01 (Malicious Password Reset).

[15] 1Password argues that the signcryption solution would be hard for them to implement, because "access to vaults must remain valid across key rotation, recovery workflows, and membership transitions", and that they are investing in a solution that also authenticates other user's public keys. We argue that non-shared vaults, which are the default for some fraction of the user-base, would still benefit from signcryption, and a careful implementation would carry over through rotation and recovery flows.

4.5.2 *Future Designs and Limitations*

The mitigations we propose can serve as a roadmap for a provably secure password manager design, with minimal performance overhead compared to existing designs, as follows. Starting from a PBKDF-derived key computed from the user's master password, standardized and efficient key derivation functions are used to derive separate symmetric keys for all vault items, thus providing proper key separation. With these keys, vault data is encrypted using a modern AEAD scheme. Using symmetric cryptography with built-in authentication both provides ciphertext integrity and has low performance cost compared to using bespoke mechanisms for achieving vault integrity. All user data is both encrypted and authenticated, with only unavoidable metadata, such as PBKDF parameters, left as authenticated-only additional data. Wherever public-key cryptography is necessary (e.g., for sharing), measures such as certificates are taken to authenticate public keys, and signatures are used to authenticate ciphertexts.

Caveat. We have presented mitigations for all the attacks we found, but there may be other cryptographic attacks yet to be discovered for which our mitigations would be ineffective. Formally defining the security goal of a password manager in the face of a malicious server is a necessary

first step toward provable security. The same applies to our high-level sketch of a secure design: only a security analysis backed by formal models and proofs can properly inform a more detailed construction; such analysis is beyond the scope of this chapter.

4.6 ETHICAL CONSIDERATIONS

Our work concerns the security analysis of four products claimed to have tens of millions of users in a threat model that we have argued as being worthy of serious consideration. Our analysis has shown these products to be badly broken in our chosen model. This situation certainly calls for a detailed statement concerning possible harms arising from our research, how these are mitigated in our approach, and the extent to which they are balanced by other outcomes. We formulate this in terms of a stakeholder-based analysis. We take a consequentialist viewpoint to conduct our analysis.

The main stakeholders related to our work, as we see them, are: us, the researchers; the vendors of the affected products (Bitwarden, Dashlane, LastPass, 1Password); vendors of related products; customers using the affected products (both individuals and organisations); society at large.

We eschew further discussion of ourselves as stakeholders, since we cannot readily set aside our own feelings on the matter to provide a rational analysis of how we might be affected by the publication (or otherwise) of our work.

Concerning the vendors of the affected products: they may suffer reputation and thence commercial damage as a result of our work being published. For example, it has been reported that LastPass has lost many customers due to repeated breaches [264, 130]. This potential damage is mitigated in our approach by providing the affected vendors with a 90-day disclosure window during which we did not talk in public about the research. During this window, we supported the vendors, initially with very detailed descriptions of the security issues we have found, then repeatedly offering support to address the issues (video-conferences, review of patches). We believed that 90 days should be sufficient time to address the identified issues, but we always remained open to extending this time period upon request, and actually did so for one vendor. We consider this approach to be relatively standard for the field of software vulnerability disclosure, except perhaps for the level of our engagement in the triage process. For example, it is endorsed by ENISA, an EU body [113].

We disclosed to Bitwarden on 27.01.2025, to LastPass on 04.06.2025, to Dashlane on 29.08.2025 and to 1Password on 21.11.2025. We had a video-conference and numerous email exchanges with Bitwarden. At the time of writing this thesis, they are well advanced in deploying mitigations for our attacks: BW01, BW03, BW11, BW12 were addressed, the minimum KDF iteration count for BW07 is now 5000, and their roadmap includes completely removing CBC-only encryption, enforcing per-item keys and

changing the vault format for integrity. On 22.12.25 they shared with us a draft for a signed organisation membership scheme, which would resolve BW08 and BW09. In the case of LastPass, our initial contact via the official email channel on 04.06.2025 did not receive a response. One of us reached out to the CTO over social media, after which our original email was found in the organisation's spam filter on 26.06.2025. We mutually agreed with LastPass to reset the 90-day clock. They provided us with regular updates and we also met for a video-conference to clarify our findings and discuss their mitigation plans. They accepted our request to work with them directly rather than with their preferred bug bounty program because the latter would impose publication restrictions that were not acceptable to us; on 7.10.25 they awarded us two bug bounty payments. At the time of writing this thesis, LP03 has been addressed. Our initial email to Dashlane was also lost, but our second attempt at contacting them on 05.09.2025 was successful. Since then, we worked with their engineers to replicate our attacks, and had several discussions over email as well as one video-conferencing call. At the time of writing this thesis, DL03, DL04, DL05 and DL06 have been mitigated by disallowing CBC-only mode in Flexible payloads; no remediation is planned for DL01 and DL02. 1Password did not request an embargo period, and commented that our attacks 1P04, 1P06 arise from already known architectural limitations. They remain unfixed. Our other attacks on 1Password were discovered after this first disclosure period, but since they are similar in spirit to our original attacks, we elected not to start a second disclosure phase. All of the vendors have publicly acknowledged our analysis and published responses [34, 240, 249, 99].

Upon agreement with all involved vendors, we shared a preprint of our research paper with them in a joint thread, allowing them to also discuss the findings and take-aways among themselves. We also coordinated with them on holding back on public statements concerning the vulnerabilities until the mutually agreed date of 16.02.2026, significantly later than the end of any of the 90-day disclosure periods.

Concerning vendors of related products: there is a clear risk that the issues we have found in the four vendors we have studied may extend to other vendors, either *mutatis mutandis* or with more substantial yet simple changes. In mitigation to the possible harms, we are aware that there is an informal industry forum where our work was already discussed as a result of the disclosure to Bitwarden. While this formally broke any agreement concerning coordinated disclosure (and arguably relieved us of the burden to maintain confidentiality), we also found it understandable and saw it as having a positive effect in alerting other vendors to the possibility of problems in their products. Vendors who have a mature security mindset could already start to analyse their own systems. We publicised our work widely after the end of the disclosure periods, including by sharing it (by request from Dashlane) with a group of CTOs of other password managers on the market, preparing conference presentations for USENIX Security and the Real World Cryptography Symposium [243], publishing a website (`webro.ke/passwordmanagers`).

We turn now to customers of this collection of vendors. Here, it is useful to differentiate between two deployment models for the password managers. Among the products we study, LastPass, Dashlane and 1Password only offer cloud-based services on their own, centralized servers. Bitwarden, on the other hand, is offered both as a cloud-based solution and as a self-hosted, on-premises application. In the first case, although we consider the malicious threat model to be realistic, we also consider that the majority of users would not be targeted by attacks of the type we have presented because of the stringent conditions needed to mount them. The dense concentration of sensitive data, however, makes the centralized servers a high-value target for attackers. Higher risk individuals and organisations could also be at substantial risk of harm. In the case of a self-hosted Bitwarden instance, the applicability of the attacks varies greatly depending on who hosts the application, and how much trust is placed in the host. These deployments are decentralized: the threat modelling should consider the trust in the administrators, the active maintenance of the hosting software stack, and the physical security of the servers themselves. A compromise in any of these areas makes our attacks realistic to carry out, and hard to detect at scale. Unfortunately, we cannot exclude the possibility that our attacks were already known to advanced threat actors – after all, we have learned from the Snowden revelations that national security agencies are routinely tasked with penetrating systems like the ones we analyse and are willing to conduct active attacks on targets. The best mitigation for these parties is to trust that vendors will rapidly and effectively patch their systems, and here we have made real effort to engage with the affected vendors to assist them in this process.

Finally, we see society at large as benefitting from our work, in that our work should help to raise the security bar for password managers, pushing their vendors to either improve security or make clearer statements about what security their systems actually provide, so that customers can judge (perhaps with the help of expert guides) whether the products meet their requirements or not. In the longer term, we see our work as contributing to reducing the possible harms from using password managers by making them more secure.

Given the overall balance of benefits versus harms, and the extensive mitigations to limit those harms, we took the decision to proceed with the research and to publish it.

4.7 OPEN SCIENCE

Our artefacts consist of proof-of-concept code for all the attacks in this chapter, with the exceptions noted. While some of the attacks are in the process of being mitigated, others remain exploitable due to the need for longer-term mitigations (e.g. public key authentication). Due to the sensitivity of the PoCs, we came to the decision that we should also not make these PoCs generally available before all of the agreed embargo periods were complete. However, we made all the artefacts available to

our reviewers in the USENIX Security '26 artefact evaluation committee. At the time of writing this thesis, all the 90-day disclosure periods have ended, and we are past the common public disclosure date on 16.02.2026. PoCs for mitigated attacks are being made available on an ongoing basis, at the following location: https://doi.org/10.5281/zenodo.17977565.

4.8 DISCUSSION

Password managers deal with critically sensitive data – yet, we have exposed severe attacks against four important vendors in this space. This section explores the manifold reasons for those attacks, discusses broader impacts of our findings, and provides directions for future work.

A burdensome legacy. Password managers were early entrants in the world of end-to-end encrypted applications: they predate all modern end-to-end encrypted messaging applications, with the initial release of LastPass dating back to 2008. Encryption of vaults has always been a focus of these products, but cryptographic best practices have changed a lot in the last years. Dashlane and Bitwarden are more recent, with first releases in 2012 and 2016 respectively, but they were entering a market with quite low cryptographic standards. The early releases of both show a taste for early-2000s cryptography, with lack of integrity protection for encryption and unprincipled designs being prevalent.

Motivated by backwards compatibility, residues of this legacy cryptography have remained across codebase changes and redesigns. In our communication with vendors, they expressed deep concern about leaving users with impossible-to-decrypt vaults because of cryptographic changes. This motivated the extreme lengths to which some of them go in order to support old formats. In the meantime, the security community had learned, through the long saga of attacks on SSL and TLS [24, 23, 52, 9, 51, 127, 283, 53, 54], to put aside backwards compatibility and orient itself towards *cryptographic alacrity*: introducing sharp, but clean, breaking changes to keep the ghosts of past cryptographic attacks at bay.

Deceptively simple. Unlike secure channel protocols and end-to-end encrypted messaging, password managers have arguably escaped deep academic scrutiny before now.[16] Perhaps this is because the problem they address looks, on the surface, like a simple matter of key derivation and then encryption, without any interesting problems to solve. After a closer look, we have seen that password managers are far from simple: they have evolved to include complicated protocols for key synchronisation, recovery and rotation, sharing of encrypted elements, and migration between different cryptographic primitives. The complexity they have reached is comparable to that of modern E2EE cloud storage systems.

Even though password managers have evolved, with cryptographic best practices trickling in over time (e.g. AEAD and memory-hard PBKDFs being eventually integrated in Dashlane and Bitwarden), there was never a gold standard to match (like TLS or Signal), or even precise security notions to aim for. This is similar to the situation for E2EE cloud storage,

[16] They are frequently subjected to code audits and security reviews, but not to *cryptographic* reviews.

where a recent rash of research [38, 159, 19, 14, 161] has exposed the frailties of deployed systems, and formal foundations closely linked to practice are only now emerging. Indeed, the two applications are close enough that the recent foundational work of [37] could provide a good starting point for password managers too.

Integrity in motion. To draw another parallel with cloud storage, password managers also face the challenge of efficiently synchronizing encrypted data. The naïve approach of storing all user files in a single AE-protected blob is clearly unviable for large amounts of data. Modern password managers, supporting large numbers of items, consequently structure their vaults to enable individual item synchronisation.

As we suggest in Section 4.5, the apparent tension between integrity and incremental updates can be solved by clean key separation. Preventing deletion remains an open problem; Merkle tree techniques can prevent selective deletion attacks by authenticating the vault structure, but complete roll-back of vaults cannot be ruled out without outsourcing versioning information to a trusted third party or local cryptographic state. The latter approach is problematic for password managers, since users expect access to their vault upon presenting only their master password, even on a stateless client.

Share with care. Many of the vulnerabilities we identify rely, in one form or another, on substituting public keys with adversary-controlled ones. Such attacks have been known since the dawn of public key cryptography [102, §III].

When it comes to verifying public keys of other users, administrators or organisations, password managers run into a very well-known problem: there is no way around the need to provide some form of verification of public keys. In short, PKI (of some form) cannot be avoided.

In our mitigations, we propose relying on PKIs or on Trust-On-First-Use (TOFU) systems. Secure messaging applications are currently exploring the use of Verified Key Directories (VKDs) to overcome this problem. While VKDs would not prevent a malicious server from associating the wrong public key with a user, they make this kind of tampering evident, so that users are alerted when their providers misbehave.

Authenticating public keys is only half the battle. It seems to be a common misunderstanding both here and for E2EE cloud storage [14, 161] that public key encryption (PKE) somehow produces authenticated ciphertexts. In fact, *no* PKE scheme can offer data origin authentication (because, by definition, any party can encrypt). This is why we suggest the usage of primitives like signcryption [293] to correctly bind ciphertexts to their senders.

Securing a "backdoor". Password managers attempt to implement mechanisms for recovery of the vault content in the case the user forgets their master password. There is a clear tension here between security and usability: not having any recovery mechanism provides for the best possible security, but might be untenable for large populations of non-technical users. Admin-assisted recovery can be implemented safely, but

success hinges on the key authentication problem discussed above being solved first. Again, we can look at the secure messaging space for state-of-the-art solutions to this problem: messengers like Signal and WhatsApp [84, 95] are converging to the usage of hardware security modules (HSMs) in the cloud for protecting account recovery and backups. Similarly, Apple's Advanced Data Protection uses HSMs to allow users to recover their key material from just their PIN [29].

Implementing part of the Key Escrow functionality on an HSM can offer at least some form of guarantee that the server is behaving honestly. Hybrid solutions could include using the HSMs to distribute the keys of the admin, without having to rely on other forms of PKI or out-of-band verification. This is an interesting future direction in the context of formal security guarantees for password managers.

4.9 CONCLUSIONS

We have argued that the malicious server threat model is the correct one for cloud-based password managers: security in this setting is both claimed in vendor advertisements and represents the gold standard for modern end-to-end encrypted systems. We then showed that four popular password managers with more than 60 million users in total are badly broken in this threat model. The vulnerabilities that we describe are numerous but mostly not deep in a technical sense. Yet they were apparently not found before, despite more than a decade of academic research on password managers and the existence of multiple audits of the four products we studied. This motivates further work, both in theory and in practice.

Beyond showcasing the relevance of the malicious server threat model in practice, our attacks show that these systems would benefit from theoretically well-founded designs. Vendors need to update their products to use modern cryptographic primitives and best practices – but to ensure solid foundations, novel definitions to capture security in this setting are also needed. Furthermore, the common issues with features such as key escrow, item-level vault encryption, password sharing, and backward compatibility represent interesting questions for the academic community on how to achieve security while still providing these functionalities.

Working together with vendors to fix the vulnerabilities, our research has contributed to improved security in practice for millions of affected users. It has also increased the vendor-driven interest in formal guarantees of security. We hope that this work serves to help raise the relevant industry sector's knowledge about encryption further above zero. At the same time, we hope our work spurs the academic community to further research in this area – as well as to consider what more it can do to convert its research artefacts into meaningful impacts on industrial practice.

Chapter 5

Conclusions – How to Encrypt

During his time in the Applied Cryptography group, the author witnessed a spectacular amount of cryptographic breakage. Some of that is included in this thesis: we broke the messaging application Threema in Chapter 2, a multi-factor key derivation function proposal in Chapter 3, and a significant subset of the most popular password managers in Chapter 4. We also broke content-defined chunking [279] and email encryption gateways [275]. In this final chapter, we will distill some *constructive* lessons from those breaks, concluding this anthology of attacks.

We structure our lessons following an abstraction ladder, going from the most abstract ones about the general process of defining security, to the concrete takeaways from attacks closer to the software implementations. Throughout this chapter, we will reference a generic cryptographic *system* as the object of our study. This system sees *users* interact through *clients* over the *network*, either with each other, mediated by a *server*, or with the server itself. The server is controlled by a *provider*, which may or may not also develop the client software.[1] Further details of the system are purposely elided, so that our discussion can be adapted to many different actual systems.

[1] We explicitly set the topic of client distribution and the verification of client code out of the scope of this chapter, since it is complex enough to warrant its own doctoral dissertation.

The first section of this chapter is dedicated to threat modelling. There we discuss the high-level exercise of defining the right adversaries our system must defend against, and the resulting security guarantees for the system. We then turn to cryptographic protocol design in Section 5.2, discussing some principles for designing a system that achieves a given set of security guarantees. In Section 5.3, we climb further down the abstraction ladder, and discuss guidance for concrete implementations.

We will link our discussion back to the case studies in the thesis, and to other notable examples in the space of cryptographic systems, using highlight boxes. These come in green, yellow and red, depending on whether we want to point out a good feature of a system in the wild, to flag elements that deserve cautious examination, or to warn against a particularly bad mistake. An example of the latter follows.

Example Cryptosystem: A Dangerous Feature

This Dangerous Feature is very dangerous. Example Cryptosystem, for example, suffered a bad attack because of it. It is best to avoid it.



5.1 THREAT MODELLING

Security does not exist in a vacuum. For it to even be meaningful to say whether a system is secure or not, we first have to define *security goals* and a *threat model*.

The security goals represent the ideal notion of security that we want to achieve. For instance, we may desire confidentiality of a message. This is the main goal of encryption itself: only the intended recipient of the encrypted message should be able to read it. More formally, we may ask that the encrypted message be indistinguishable from the encryption of a message of the same length, or that no information about the message is leaked by its encrypted form. Other common security goals for encryption are integrity and authenticity, but many more exist.[2]

The threat model identifies what the possible adversarial entities are, and assigns to them *capabilities*. The historical Caesar's cipher[3] is perfectly secure if your adversary can't read or count, plaintext radio communication is secure against an adversary without access to radio equipment, and quantum key establishment is unbreakable if you assume away the insecurity of repeaters. While all trivially untenable by modern standards, some of these threat models have been considered appropriate in their given circumstances and times.Caesar's cipher was sufficient to encrypt your diary in ancient Rome; lack of radio equipment is what protected the confidentiality of all non-military wireless communication up to the adoption of the GSM family of standards in the 90s [149] and, perhaps surprisingly, of large swaths of satellite communication even at the time this thesis is written [291].

Armed with security goals and a precise threat model, we can turn to designing a system that achieves a specific set of *security guarantees*: that is, security goals modulated by a threat model. As a concrete example, an end-to-end encrypted messaging application aims to achieve confidentiality, integrity and authenticity, in the face of an adversary that controls all but the two very ends (that is, the two clients) of the communication channel.

[2] For extensive guidance on the common security goals set by cryptographers, we refer the reader to cryptography textbooks [179, 64].

[3] A substitution cipher, said to be used by the Roman general, where each letter in the message is shifted by n positions in the alphabet to produce a ciphertext. With the historical $n = 3$, a plaintext "ferum victorem cepit" becomes "ihuxp ylfwruhp fhslw".

5.1.1 *The Network Adversary*

The weakest adversary we consider only sees, and possibly tampers with, the traffic on the wire. It goes without saying that our cryptographic systems should be secure in this setting.

Making a protocol secure against a network adversary is one of the problems modern cryptography solves well: we just need a *secure channel*, which protects the confidentiality and the integrity of all communication in the channel, and authenticates either (or both) ends. We have extensive literature on secure channels, and many efficient instantiations—chiefly, TLS 1.3. Using a secure channel like TLS allows us to stop worrying about network adversaries, and focus on the server adversaries presented in the next sections.

Server authentication. Secure channels need, at the very least, a way to authenticate the server. Capturing how this authentication works is also an important element of threat modelling in the network adversary setting: we have to trust the mechanism for public key distribution. TLS relies on the already established Internet PKI for authentication; other options include using a custom PKI, or bundling key material to authenticate the server with the client application.

Password Managers: Content Delivery Networks



Many of the password managers we analyse in Chapter 4 use Content Delivery Networks (CDNs) in front of their servers. This is common because of load balancing and traffic filtering, but it also changes the security posture of the systems: CDNs terminate TLS connections, so they can see the network traffic in the clear. We therefore need to consider them at least as trusted as the end server of the service provider.
At the time this thesis is written, Bitwarden uses Fastly, Dashlane uses Cloudflare and LastPass uses Akamai.

Password Managers: Lack of Certificate Pinning



Even though we can trust the cryptography in TLS 1.3, the authentication of the server ultimately rests on the Internet PKI, which includes hundreds of different Certificate Authorities. Pinning the server's public key (or a dedicated CA) removes this reliance and protects against misissued or compromised certificates. None of the password manager Android applications pinned certificates: this is probably due to the reliance on CDNs discussed above. An alternative motivation lies in the fact that all password managers also implement web clients, accessible through HTTPS. Those clients must implicitly rely on the Internet PKI and its revocation and transparency mechanisms.

Client authentication. While in some settings it is sufficient to only authenticate the server to ward off a network adversary, for most end-to-end encrypted systems we want to additionally authenticate the client to the server. If that is not the case, a network adversary could impersonate any user to the server.

1Password: Password-Authenticated Key Exchange



When the client has a password or a symmetric key, then the secure channel can be instantiated with a Password-Authenticated Key Exchange (PAKE), as in 1Password. This achieves a strong form of mutual authentication: not only does the server authenticate the client, but the client can also rest assured that they have established a channel with the very same server that they registered with. In comparison, when using public key authentication for the server in the secure channel, the client only knows that they established a

channel with a server that possesses the corresponding private key. The use of a PAKE also precludes the secure channel from being terminated early, e.g., at a CDN edge.

The External Actor. A stronger variant of the network adversary threat model is that of an external actor. In this setting, the adversary additionally gets access to the system as an external user. This can generally be treated as equivalent to the network adversary threat model when the system is publicly available on the network. Then, the adversary can trivially register a new user account and interact with the system as a user. We implicitly consider this in our case studies of Threema and MFKDF.

We note, though, that these two models may diverge in the setting of closed systems, such as privately hosted Threema or self-hosted Bitwarden: here, users may need approval to register. For instance, the C2S Key Compromise attack of Section 2.4.1.1 translates to a private Threema instance, but the Vouch Box Forgery of Section 2.4.1.2 does not, since it requires the adversary to have access to the E2E protocol to receive messages, as well as the ability to register user accounts with arbitrary keys.

5.1.2 *The Honest (but Curious) Server*

In this threat model, we allow the adversary access to everything the server sees—essentially stripping away the confidentiality of the secure channel between server and client. We retain the integrity: we don't allow the adversary the ability to tamper with any message. Here, the server faithfully runs its code, but is curious about—or is compelled to hand over—the data it stores.

Here, we hit somewhat contentious ground over whether a system is truly end-to-end encrypted or not. It is trivial to realise a lot of functionality assuming honest servers: there is no need for authenticating public keys of other users, since we can trust the server to honestly deliver them. We can also offload integrity guarantees to the server, since it won't tamper with stored data. As we saw for password managers in Chapter 4, much of the older literature stops at this threat model. Throughout this thesis, we contend that an honest-but-curious server is not the right threat model for end-to-end security guarantees: as we argue at length in Section 5.1.3, there are many reasons not to blindly trust the goodwill of the providers.

There are properties, on the other hand, where trust in the server is not only well-placed, but also sometimes essential. Even an end-to-end encrypted service should shield against the dissemination of encrypted user data: this means authenticating the users before letting them access the data. This layer of authentication relies entirely on the server.

MFKDF: Read Access to Public State



The public MFKDF state is instrumental in all of our attacks. The state doesn't just leak metadata: its encrypted components also inadvertently leak key material, either on their own (like in the HOTP Bias Attack of Section 3.3.3.2), or when combined with other information (like in the HOTP Compromise Attack of Section 3.3.3.1). It also exposes users to offline brute-force attacks against their password and factors. Arguably, the way MFKDF is designed does not allow it to protect the state: MFKDF users need to access the state *before* authenticating, since it contains all the material to derive keys.
Naturally, this state would be available to a "curious" server, but MFKDF's design makes it available to any adversary on the network. This is a clear case of how *not* targeting a weaker threat model for some security properties hurts the overall security of a system.

Password Managers: Authentication and Vault Access



The password managers we study avoid the MFKDF pitfall mentioned just above, by authenticating the users before letting them access their vaults.
Yet, they all fall victim to KDF downgrade attacks (BW07, LP04, DL04, 1P04): clients blindly trust the PBKDF parameters sent by the servers, which are needed to correctly derive a key from the password before authentication can begin.
The subtle issue here is that we want to trust the servers for authentication, but not for the end-to-end security of the vault, and the PBKDF parameters impact both. The easy fix—having the client only support the most modern PBKDF parameters—is then in conflict with backwards compatibility: new clients would be unable to decrypt old vaults.
This tension can be partially resolved by separating the KDF parameters used for authentication and the ones used for deriving vault keys, as we propose in our KDF Mitigation in Section 4.5.

Honest servers can also contribute to the security of the system by implementing rate-limiting and hardening users' passwords via oblivious PRFs—defending, respectively, against online and offline password brute-force attacks. Sometimes, these features are implemented within a hardware security module, which can be thought of as an honest component of a possibly malicious server (see, for instance, the WhatsApp system for protecting backups [95] and the key recovery system for Signal [84]).

Dropping traffic is a trivial attack that any network adversary can carry out: availability is therefore commonly considered only in the face of an honest server, outside of cryptographic systems explicitly designed with network fault tolerance in mind.

Snapshot Adversary A slight weakening of the honest-but-curious threat model is the *snapshot* adversary. That is, an adversary that can see all the data on the server, but rather than continuously (e.g., in the form of live traffic), they can only observe discrete snapshots of the data.



Signal: Snapshot Anonymity

Signal implements several anonymity features (most notably Sealed Sender [280]) which are ineffective against a fully malicious server, or even an honest-but-curious one: statistical analysis, correlating IP addresses with sent messages, would defate most of them [175, 207]. Even though these security guarantees only hold in a snapshot setting, we consider this appropriate. The aim here is to reduce the amount of metadata that can possibly accumulate on the server, forcing a potential adversary to be more invasive than a snapshot one.

5.1.3 *Malicious Server*

In this threat model, the adversary fully controls the server, and is free to arbitrarily deviate from expected behaviour. For true end-to-end encryption, we want security goals such as confidentiality and integrity to hold against such an adversary.

Digital infrastructure can and does fall into adversarial hands, meaning that trust in the service providers to follow protocol is largely insufficient. Notable examples include infiltrating the infrastructure through supply chain compromise, like the record-setting cryptocurrency heist of 1.5 billion USD on Bybit [286]. Sometimes, malicious actors gain access through outdated software, like the 2019 hack of Matrix [276]. Another vector is law enforcement action: both the EncroChat and Sky ECC encrypted communication networks were breached by taking over some of the server infrastructure used by the services [21, 145]. Some service providers, such as Microsoft, are known to help security agencies bypass encryption, as in the documented cases of Skype [148] and BitLocker keys [70].



Threema: End-to-End Forward Secrecy

End-to-end properties such as forward secrecy should hold in the presence of a malicious server. This was not the case in Threema. This is surprising: Threema achieves confidentiality against a malicious server, but forward secrecy only against an honest-but-curious one. Even setting aside the merits of forward secrecy, surprises like this are undesirable in themselves. Cryptography should be boring.

Assuming a fully malicious server means that the security guarantees the system achieves rest entirely on the client(s). From the point of view of a malicious server, the clients are then a set of oracles: every functionality they expose that can be subverted becomes a component for an

attack. A complete break of the underlying primitives is rare with modern cryptography: many modern attacks rely on unexpected interactions of functionality and sub-protocols. There is no rulebook: halting protocols mid-execution, forwarding protocol messages into other protocols, or exploiting client telemetry messages are all fair game.

Threema and Password Managers: Client as Oracles



Some of the most interesting oracles we discovered in our attacks include the Vouch Box Forgery and the Kompromat cross-protocol attacks in Threema (see Sections 2.4.1.2 and 2.4.2.3, respectively), and the compression side channel (Section 2.4.3.2). The cross-protocol attacks relied on the lack of separation between keys and messages of different protocols, while the compression side channel was only practical because the client re-attempted failed backups. The Kompromat attack should also teach us that the server should not have full control of the plaintext to be encrypted using a client secret. Many password manager attacks rely on the server triggering synchronisation behaviour, and in Dashlane's case, we built an entire padding oracle attack chain out of telemetry messages (see DLb2 and DLb1).

'Stunt' cryptography aside, a malicious server also has more basic tools available to them: observing all plaintext and ciphertext, and tampering with data in transit or stored on the server.

MFKDF: Write Access to Public State



MFKDF does authenticate write access to the state. Naturally, this does not help when the adversary is the server itself. This enables several attacks, like Parameter Tampering (Section 3.3.2.2) and Share Dilution (Section 3.3.2.3). The threat model was not explicit in the original work [220], which is probably part of the reason these attacks were missed.

Dashlane: Safer Key Escrow



Dashlane is the only password manager we study that is not susceptible to attacks exploiting the key escrow functionality. As we note in Section 4.4.1.3, this is because the data needed for key recovery is never actually uploaded to the server. The adversary—even one who controls the server—cannot access it.

5.1.4 Compelled and Physical Access

In this threat model, the adversary is given direct access to the client. It is clear that many security properties won't hold here: the adversary is in control of one of the very ends that end-to-end encryption is supposed to protect. Compelled and physical access are highly relevant in settings like border crossings, where border agents routinely ask for access to

unlocked devices, or intimate partner violence, where an abusive partner is the threat.

Even in such a strong threat model, there are still security guarantees that we can hope to achieve. First, we may want to limit the impact of compromise that the adversary can realistically carry ou given access to the device. This may entail programming the application interface in a way that makes it hard to export secrets; encrypting backups with keys chosen at setup time rather than at the time the backup is made; and notifying the user of security-critical events, like access to the server from unexpected devices.



Threema and Aegis: Exporting Secrets

Threema's "ID Export" allowed a user to directly save their long-term key. This feature, included in the name of interoperability, is clearly a liability against an adversary with access to an unlocked device. In contrast, Signal also allows migration to a new phone, but when the migration is completed, the old device is locked out of the account.
Aegis [41], a popular T/HOTP application for Android, suffers from a similar pitfall: the user interface can show the T/HOTP secrets without requiring further authentication, meaning that it is trivial for an adversary with access to the unlocked app to clone the second factors stored there. In sharp contrast to showing the current one-time code, showing the secret means that the second factor is permanently compromised.

Second, even when assuming a full compromise of every secret on the device, we may limit the fallout of the compromise into the future and the past. In messaging applications, forward secrecy, combined with message deletion policies, can limit the compromise of past messages; and protocols that heal through post-compromise security limit how long the adversary can retain access to messages in the future. Features supported by an honest server can also help in this setting: multi-factor authentication with one-time codes, for instance, can limit the access an adversary has to protected data by locking them out when a session expires, even if all key material is compromised. Finally, designing robust key rotation procedures also helps recover from a compromise.



MFKDF: No One-Timeness

Our main finding on MFKDF is the complete lack of one-timeness, even when one-time factors are used for key derivation (see Section 3.3.1). This is obvious *a posteriori*: MFKDF always derives the same key, so it cannot hope to recover from compromise without key rotation. But it exposes a sharp incompatibility between the expected threat model of one-time codes (which usually maintain security after compromise) and the one in which MFKDF operates. This incompatibility is surprising—and, as we noted before, surprises are bad in cryptography.

At-Compromise Security Much of this discussion of the compelled access threat model is purely speculative. A recent work [17] has highlighted the importance of realistic security goals and threat models, backed by ethnographic studies. One such model is at-compromise security, capturing the setting of abductions, where the adversary has prolonged physical control of both the user and their client. In this extreme, but realistic, threat model, the same work shows that it is still possible to achieve At-Compromise Alert Blindness, allowing the user to send an alert as they are taken without their abductors noticing.

5.2 CRYPTOGRAPHIC DESIGN

With a precise threat model and a set of security goals in mind, one can start designing the cryptographic system itself. This involves designing concrete *protocols* that the clients (and servers) in our system will execute. Comprehensive guidance on how to build protocols from first principles is out of the scope of this thesis. As the existence of an entire field of requirement engineering attests, even just collecting the right requirements for a system is complicated. We instead focus on a few principles distilled from our attacks.

Choice of primitives is, today, relatively easy: there are many standard, provably secure and widely available building blocks. An opinionated list of cryptographic primitives can be found in Appendix A. The challenge is usually rather the availability of these primitives in our target programming language and execution environment: here design needs to proceed non-linearly, keeping implementation constraints in mind.

The one constant across all of the case studies in this thesis is that using secure primitives does not yield secure protocols. Threema, MFKDF, and all the password managers we study mostly[4] use secure primitives, and yet fall victim to attacks galore. We list below some of the recurrent reasons behind these attacks.

[4] The exception being the use of non-AEAD encryption schemes in MFKDF and LastPass, both of which predictably led to attacks.

Key Reuse. A constant problem across many protocols is the lack of key separation. A (symmetric) key should only be used for a single purpose.

Many building blocks take a symmetric key as an input, and, if we have a key available, it is clearly tempting to reuse it in multiple ways: sharing a key for encryption schemes and MACs, or KDFs, or even just for the encryption of different unrelated objects.

Any key reuse is bound to either result in attacks, or make provable security very hard,[5] or both.

[5] Telegram's MTProto is infamously unprincipled, which resulted in an unpleasantly complex security proof [20].

Password Managers: Cut-And-Paste Attacks



All of the password managers we study contained some form of key reuse. In Bitwarden and LastPass, the same key was used to encrypt different fields of an item: fields could then be swapped

around by a malicious server without the client noticing, resulting in the password recovery attacks BW06 and LP03.
In Dashlane and 1Password, the same key is used to encrypt different items, meaning that a malicious server can duplicate items (attacks DL01 and 1P02).



MFKDF: Full Compromise

The worst-case scenario for key reuse is showcased in the HOTP Compromise attack for MFKDF (Section 3.3.3.1). MFKDF committed two sins: using a one-time pad as the encryption scheme for the factor state, and reusing the key K as both the MFKDF output and the key for encrypting the factor state with the one-time pad. The encrypted factor state is public: this meant that if the plaintext state (in this case, the HOTP secret) is leaked to the adversary, the adversary can recover K. This defies the entire point of MFKDF: the leakage of a single factor bypasses all the others.

The solution is simple: use a KDF to derive multiple, independent keys, and keep the derived keys cleanly scoped to a single purpose. This is in most cases cheap enough to warrant a change in the design, even (or especially) before a concrete attack exploiting a particular instance of key reuse has emerged.



Password Managers: Key Hierarchy Hygiene

Bitwarden uses the same key both for the legacy encryption path for RSA keys, and for the encryption of modern user keys. While, arguably, the legacy path is the bigger problem here, key separation would have prevented attack BW12.
LastPass uses the vault key both for encrypting the items in the vault and to derive the authentication h_u. This, to our knowledge, does not result in a direct attack, but it is bad design.
Dashlane has a similar stain in its key derivation diagram: the legacy CBC key is used both for encrypting legacy transactions with AES-CBC, and for deriving the vault key (via SHA-512, which notably is *not* a KDF). This also did not result in attacks that we could find.

Using the wrong building block. Many protocols reach for security properties that a particular building block actually does not grant. This can be due to subtleties in how the security notion of a primitive is defined, because of security degradation of the primitive due to attacks, or because of a misunderstanding of the primitive itself.

A classical example of a subtle property is the lack of key commitment of AEAD schemes. Authenticated encryption guarantees that a ciphertext c' which was not produced by encryption with key k will fail to decrypt under key k. It does not guarantee much in the case a ciphertext is decrypted under a key $k' \neq k$. If we wish decryption to provably fail in that setting, we need a key-committing AEAD scheme.

MFKDF, Tor: The Limits of Off-the-Shelf AEAD



Some settings impose unusual constraints that rule out the authenticated encryption schemes discussed above. Share encryption in MFKDF is one such case, as we show in Attack 3.3.3.4. If the Shamir shares are encrypted with a low-entropy key, authenticated encryption provides an oracle for guessing that key.
An example from outside this thesis is the Onion encryption in the Tor protocol. Tor nests n layers of encryption: using standard AEAD would incur a cost of n authentication tags. Tor requires instead a ciphertext size independent of n.
Both settings require a careful, fresh cryptographic design, with provable security helping to make the goals formal, and checking that the design actually achieves them. MFKDF2 still lacks such a building block; in the case of Tor, this gap was recently filled by the Counter Galois Onion construction [96, 97].

Primitives that degraded over time include the hash algorithms MD5 and SHA-1; it is easy to find pre-images for MD5 hashes, and collisions for SHA-1 inputs. Those primitives are usually already avoided in modern cryptographic protocols [143].

A very surprising example of a misunderstanding of a primitive is represented by the many instances of public key encryption being used as if it provided sender authentication—which it does not.

1Password, Bitwarden: Trusting Public-Key Ciphertexts



The Vault Substitution attack on 1Password (1P06) and Organisation Injection and Overwrite attacks on Bitwarden (BW08, BW09) both result from the password manager clients receiving public key ciphertext from the server, and not authenticating it correctly.
In Bitwarden, when a user joins an organization, the Organisation key is encrypted to that user's public key. In 1Password, users encrypt their vault key under their own public key. In both cases, the resulting ciphertext is then stored on the server. A malicious server can simply replace the ciphertext with an encryption of a key they control: after that, the client will use an adversary-controlled key to, respectively, encrypt shared items in the organization, or encrypt items in the user's vault.
This would not happen if the public key encryption additionally provided sender authentication, but this is not a property that the RSA-OAEP scheme used by both products provides. This can be realized either via explicit authentication (e.g. signcryption), or implicit authentication via HPKE authenticated encapsulation mode.
A similar attack also occurred in the cloud storage system Nextcloud [15].

Another recurring issue is assuming that Diffie-Hellman outputs are unique per (public key, secret key) input. This is also generally not the case, as argued at length by Perrin [236].



Threema: Gateway Low Order Points

The Threema Gateway allowed users to register public keys without verifying that they held the corresponding private key. We used this to register the public key S that the server uses in the C2S protocol, as part of our cross-protocol Vouch Box Forgery attack of Section 2.4.1.2.
Notably, Threema uses the Curve25519 group. Due to clamping, the private DH key a is always a multiple of 8. This means that for all points P such that $8P = 0$, $(P \cdot S)^a = S^a$: claiming the public key $T = P \cdot S$ is equivalent to claiming S, for the purpose of Diffie-Hellman. There are 8 such points P, since Curve25519's order has a cofactor of 8, including the point at infinity. We could therefore register up to 7 keys that are equivalent to the server's, but not the server's. Even after initial mitigation from Threema stopped us from registering the server's public key, this property of Curve25519 allowed us to keep registering keys that worked just as well for forging vouch boxes.

Backwards Compatibility The security community learned this lesson the hard way over the long saga of SSL and TLS: a protocol is only as secure as the weakest primitive it supports. Support for legacy cryptographic paths almost always leads to attack. The answer is cryptographic *alacrity*: making sharp but clean breaking changes.



Password Managers, MFKDF: Weak PBKDF Support

KDF downgrade attacks are widespread: we observe them on both password managers (BW07, LP04, DL04, 1P04) and MFKDF (Section 3.3.2.2). If the client supports multiple PBKDFs, and the server controls the choice of PBKDF and its parameters, a malicious server can always pick the weakest PBKDF supported. Clients should aggressively drop support for weak PBKDFs, at least when it comes to authenticating to the server. MFKDF2, for instance, now only allows Argon2id.



Password Managers: Legacy Encryption Support

LastPass still protects vaults with unauthenticated AES-CBC, and Bitwarden and Dashlane carry enough legacy cryptography that we can downgrade them into a similar state. Our attacks (BW12, DLb1) show that using good encryption primitives is not enough: one has to actively stop using bad ones.

Careless Protocol Composition Even protocols that are secure in isolation can break when composed, because security is not, in general, preserved under composition.

Password Managers, Threema: Dangerous Interactions

Threema is a cautionary tale of *accidental* compositions: two protocols sharing key material interact to produce an attack, while each is arguably (more) secure in isolation. Vouch Box Forgery (Section 2.4.1.2) shows a C2S and E2E interaction, while the Kompromat attack (Section 2.4.2.3) relies on an E2E interaction with the registration and contact discovery protocols.
The Icon URL attacks on password managers (BW06 and LP03) can also be seen as cross-protocol interactions: here the vault encryption protocol interacts badly with the icon fetching protocol.
All of these attacks can be prevented with key separation.

Reinventing the Wheel A final bit of cryptographic advice is: do not build a protocol at all, if an existing protocol can serve your purpose. "Don't roll your own crypto", goes the saying, and using an existing protocol minimizes the amount of crypto being rolled. Both TLS 1.3 and the Noise framework [235] provide provably secure and easy-to-use secure channel protocols. The messaging primitives of Signal and MLS can be used to implement a variety of end-to-end encrypted systems, since they provide encrypted event streams and (group) continuous key agreement.

5.3 IMPLEMENTATION

Implementations realize cryptographic designs. A cryptographic design and its realization are often deeply intertwined: not only are there multiple ways to implement a design, but the implementation constraints often shape the design itself. As we hinted at in the previous section, designs depend on the availability of primitives in the target execution environment; they also have to adhere to memory, computation and latency constraints.

Implementations then contribute to attacks in at least two ways. On the one hand, the implementation can *constrain* the design: the primitives and resources at hand force the design into a shape that is hard to secure. On the other, an implementation can realize an otherwise sound design imperfectly, leaving *gaps*.

Implementation Constraints. There is little to be done about constraints that relate to resources: the security limitations that result from them should simply be documented clearly. The more interesting setting is that of the primitives and APIs, and how they limit and shape the designs that can be built on them.

A convenient but restrictive interface can nudge implementers towards insecure constructions. It is folklore that the Web Crypto API can be difficult to work with,[6] but that is often all that is available in browsers. When it is possible to choose, it is then important to pick a library that provides the right building blocks for designing secure protocols.

[6] Dardanis had some first-hand experience of Web Crypto's limitations in an implementation project [89] co-supervised by the author.



Threema: A Restrictive `Box()`

Many of our attacks on Threema can be directly traced back to Threema's usage of the NaCl library. In particular, to its `crypto_box()` asymmetric encryption primitive (`box()` henceforth). `box()` offers a very simple interface: it takes as input the sender's private key, the receiver's public key, a nonce and a message. It produces a ciphertext as an output.

This clean and apparently foolproof API does not expose enough detail to easily build a secure protocol. It lacks both a *label* input for key separation, and an *additional data* input for parts of the message to be authenticated but not encrypted.

Some of the label restrictions can be worked around. A protocol can enforce separation at the message or nonce level, introducing distinct message formats or split nonce spaces per intended use; alternatively, it can use separate public keys. Threema did not take these extra steps: this resulted in cross-protocol attacks (Sections 2.4.1.2 and 2.4.2.3), since messages from different subprotocols could be confused for one another.

`box()` also lies behind the awkward "nonce database" component: end-to-end messages used the same DH keys for both directions of communication, which could result in reflection attacks. Threema attempted to mitigate this by keeping track of all already-seen nonces; a brittle design which failed if the database was reset, e.g., after reinstalling the app (Section 2.4.2.2).

The lack of an additional-data input led Threema to store metadata in a separate "metadata box", which was not bound to the ciphertext and could therefore be stripped away, resulting in a reordering attack (Section 2.4.2.1).

A clean design would involve a KEM component to create a shared key, a KDF to derive distinct keys from the shared key, and an AEAD scheme. `libsodium` [98], a NaCl fork, provides interfaces for all of these primitives, but arguably the better approach is to use the modern Hybrid Public Key Encryption (HPKE) standard [40], which provides this combination of KEM, KDF and AEAD with a single-shot API not too different from `box()`. An interface that makes the safe thing awkward will be routed around, usually insecurely, by the non-cryptographer who is its most important user.

Implementation Gaps. Implementations are the most concrete and faithful representation of a cryptographic system, but they are, at the same time, the hardest to reason about. Our case studies of Threema and password managers all start from concrete instantiations rather than a clean design. Analysing them therefore begins with *modelling*: extracting a high-level design from the code,[7] abstracting away details until the description fits on a page or a whiteboard. This process is inherently lossy. Conversely, implementers do not just go from a design to its implementation: they have to expand the design to fill in all the details, a process that is inherently creative. The design an author intends, the

[7] This can be human-readable code, or reverse-engineered machine code.

code they write, and the model an analyst recovers are thus three distinct objects, and attacks live in the gaps between them.

At the low level, gaps arise because the code may simply get something wrong, or because the design may be too imprecise to implement unambiguously, so that the implementation resolves the underspecified parts in an insecure way.

MFKDF: Underspecified by Design



Several MFKDF attacks exploit underspecification (Section 3.3.3). The specification treats one-time codes and Shamir shares as uniform bitstrings, but their concrete encodings are not uniform. HOTP codes are decimal, and therefore biased, which yields the HOTP Bias attack (Section 3.3.3.2); the shares produced by the reference secret-sharing library are biased too (Section 3.3.3.4). Because the design leaves these encodings unspecified, whether an attack actually applies ends up being decided by the implementation: the biased-share attack, for one, is only avoided by an unrelated implementation quirk that happens to leave part of each share unencrypted. Precise designs do not leave their security to such accidents.

At the high level, gaps emerge when the machinery around the cryptographic core—the user interface, or session and device management—fails to map cleanly onto the protocol. This can leave the application broken even when every individual protocol is implemented faithfully.

Wire: Bad Integrations



Tsouloupas' analysis of the Wire messenger [281] uncovered several attacks, including a trivial attack against the confidentiality of messages. Wire uses the Double Ratchet protocol, but, over the course of a rewrite, the application stopped checking whether the keys of the recipient were actually authenticated. A malicious server could simply inject its own public key and intercept Double Ratchet sessions. Similar vulnerabilities also appeared in the security analysis of Jitsi, a videoconferencing software [201], and of Bridgefy [18]; both used the Signal library.

5.4 EPILOGUE

In this thesis, we explored how cryptography fails, and what we can learn from it.

There are many more ways for a system to fail, than for it to be secure. This is why cryptographic systems should be constructed in a principled way, with provable security in mind from the outset. Reactive security—the process of iteratively patching attacks as they are found—may never converge, as the SSL/TLS saga attests.

That said, cases like TLS 1.3, which embodies the "proactive" security approach, remain the exception, not the rule. Fully designing a protocol with formal support requires expert input, and can be extremely slow.

TLS 1.3 itself was in the works for more than six years, before a final specification was published—with a non-negligible fraction of the entire cryptographic community working at it. Threema, Bitwarden, LastPass, Dashlane, 1Password, all evolved over time. They are moving targets for security analysis, with simpler original designs that grew organically as new functionality was added. Nor could they draw on a comparable amount of cryptographic expertise.

Developers now have access to solid libraries and APIs. As we show, this, by itself, is insufficient: protocols are not a simple matter of mixing together secure primitives. Not only can "good" primitives result in "bad" protocols, but even good protocols can be integrated in a way that leads to broken applications. Measuring the concrete end-to-end guarantees requires a holistic security approach, analysing cryptographic systems at the protocol level, in its implementation, and with different threat modelling angles.

We argue that an attack-driven methodology is important to realize this holistic security. Attacks cut through the layers of abstractions, help to surface the limitations of relaxed threat models, and provide an incentive to shift towards higher assurance alternatives supported by a security analysis. They're also *fun*.

We expose recurring cryptographic failures with our case studies, and our contribution is dual. We outline important attack strategies that make up a cryptanalyst's *toolbox*. These attack strategies, at the same time, are *lessons* that inform a principled cryptographic design.

Appendix A

An Opinionated Selection of Primitives

As we argue in Section 5.2, choosing cryptographic primitives is, today, relatively easy: there are many standard, provably secure and widely available building blocks. The concrete selection is ultimately limited by what our chosen libraries can afford us, and by the constraints of the target execution environment. This appendix collects the primitives the author would reach for, when available.

Hashes and MACs. Most of the designs we study correctly use the SHA-2 and BLAKE2 families of hash functions, and use HMAC with SHA-2 hashes or keyed BLAKE2 as a MAC. Modern alternatives include the SHA-3 function family, including the SHAKE eXtendable Output Functions (XOFs), which can produce arbitrarily long outputs. BLAKE3 also offers XOF functionality, and can be instantiated directly as a MAC.

KDFs. HKDF [185] is the standard tool for key derivation: one applies HKDF-Extract once to the input keying material, and then HKDF-Expand with a distinct label for each derived key. A useful generalization of the concept of KDF is the idea of key combiners, which take as input multiple keys, and can output multiple keys. This has recently been formally studied [36], and a number of competing constructions exist. HMAC can be used as a 2-key combiner, as long as the keys are fixed-size [35].

PBKDFs. Password-based key derivation functions are designed to be expensive to compute, in order to slow down brute-force attacks. PBKDF2 with a large number of iterations remains common, but the most modern alternatives are Argon2id and scrypt, which are *memory hard*: they require a large amount of memory, in addition to being computationally expensive.

Symmetric Encryption. The standard advice is to always use Authenticated Encryption with Associated Data (AEAD) schemes. This practically translates to AES-GCM or ChaCha20-Poly1305, or AES-GCM-SIV if we want to harden against nonce reuse.

Other notable schemes include XChaCha20-Poly1305 [30], which extends plain ChaCha20-Poly1305 to 24-byte nonces. A larger nonce space means that we can encrypt more data with the same key: the probability of a nonce collision when using random nonces decreases. AES-GCM-DNDK [151] similarly extends AES-GCM with 24-byte nonces,

and additionally supports key commitment. This means that a ciphertext encrypted with key k will fail to decrypt under a different key k'—a strengthening of the normal AEAD security goal. AES-GCM-DNDK is reportedly widely used within Meta [150]; XAES-256-GCM [282] is a similar proposal without the commitment property.

If we do not need low-level access to the encryption primitive, Google's Tink library [146] provides a series of good, high-level wrappers.

Schemes that are not yet in widespread use, but that have potential, include FLOE [115] for streaming encryption with random access, and AES-OCH [211] for commitment to both key and nonce, as well as the early "flexible AEAD" proposal [45].

Public Key Encryption. The go-to standard for modern public key encryption is HPKE [40]. The Hybrid Public Key Encryption specification supports different combinations of Key Encapsulation Mechanisms (KEMs), KDFs and AEADs. Supported KEMs include classical Diffie–Hellman over elliptic curves, post-quantum ones like ML-KEM, and hybrid KEMs combining both [86]. HPKE also optionally provides sender authentication, based both on public keys and on pre-shared secrets.

RSA-OAEP is still a common and sound choice for key transport, but should be combined with HPKE-like KDFs and AEADs.

Signatures. Deterministic ECDSA and DSA [238] or EdDSA [176] are good low-level primitives. For post-quantum security, the NIST-standardized ML-DSA (FIPS 204) and SLH-DSA (FIPS 205), or hybrid combinations with a classical scheme, are the natural choices.

Key Agreement. HPKE's DH-KEM [40] and ML-KEM [86] constructions.

Bibliography

1. LastPass. *LastPass Business*. https://www.lastpass.com/products/team-password-manager. Accessed: 2025-03-25.
2. LastPass. *LastPass CLI Client Source Code*. https://github.com/lastpass/lastpass-cli. Accessed: 2025-03-25.
3. LastPass. *LastPass Whitepaper*. https://support.lastpass.com/s/document-item?language=en_US&bundleId=lastpass&topicId=LastPass/lastpass_technical_whitepaper.html&_LANG=enus. Accessed: 2025-03-25.
4. 1Password. *1Password Whitepaper*. https://1passwordstatic.com/files/security/1password-white-paper.pdf. Accessed: 2025-03-25.
5. 1Password. *Linking Other Devices*. 1Password Security Design White Paper, https://agilebits.github.io/security-design/passkeySSO.html#linking-other-devices. Accessed: 2026-06-05.
6. 1Password. *Zero-Knowledge Encryption*. https://1password.com/features/zero-knowledge-encryption. Accessed: 2026-06-05.
7. *1Password Surpasses $400M ARR and Expands Executive Team to Advance the Next Era in Identity Security*. https://1password.com/press/2025/nov/1password-strengthens-leadership-amid-growth-milestone. Accessed: 2026-01-09.
8. Harold Abelson et al. 'Keys under doormats'. In: *Commun. ACM* 58.10 (2015), pp. 24–26. DOI: 10.1145/2814825.
9. David Adrian et al. 'Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice'. In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li and Christopher Kruegel. ACM Press, Oct. 2015, pp. 5–17. DOI: 10.1145/2810103.2813707.
10. Jan Ahrens. *Threema protocol analysis*. Mar. 2014. URL: https://blog.jan-ahrens.eu/files/threema-protocol-analysis.pdf.
11. Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx and Sophie Schmieg. 'How to Abuse and Fix Authenticated Encryption Without Key Commitment'. In: *IACR Cryptol. ePrint Arch.* (2020), p. 1456. URL: https://eprint.iacr.org/2020/1456.
12. Martin R Albrecht, Jorge Blasco, Rikke Bjerg Jensen and Lenka Mareková. 'Mesh messaging in large-scale protests: Breaking Bridgefy'. In: *Cryptographers' Track at the RSA Conference*. Springer. 2021, pp. 375–398.
13. Martin R Albrecht, Raphael Eikenberg and Kenneth G Paterson. 'Breaking Bridgefy, again: Adopting libsignal is not enough'. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/albrecht.

14. Martin R. Albrecht, Matilda Backendal, Daniele Coppola and Kenneth G. Paterson. 'Share with Care: Breaking E2EE in Nextcloud'. In: *2024 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, July 2024, pp. 828–840. DOI: 10.1109/EuroSP60621.2024.00051.

15. Martin R. Albrecht, Matilda Backendal, Daniele Coppola and Kenneth G. Paterson. *Share with Care: Breaking E2EE in Nextcloud*. Cryptology ePrint Archive, Report 2024/546. 2024. URL: https://eprint.iacr.org/2024/546.

16. Martin R. Albrecht, Sofía Celi, Benjamin Dowling and Daniel Jones. *Practically-exploitable Cryptographic Vulnerabilities in Matrix*. https://nebuchadnezzar-megolm.github.io. Sept. 2022. URL: https://nebuchadnezzar-megolm.github.io/static/paper.pdf.

17. Martin R. Albrecht, Simone Colombo, Benjamin Dowling and Rikke Bjerg Jensen. 'Automatic-Compromise Security - The Case for Alert Blindness'. In: *EUROCRYPT 2026, Part II*. Ed. by Joan Daemen and Emmanuel Thomé. Vol. 16542. LNCS. Springer, Cham, May 2026, pp. 191–221. DOI: 10.1007/978-3-032-25317-0_7.

18. Martin R. Albrecht, Raphael Eikenberg and Kenneth G. Paterson. 'Breaking Bridgefy, again: Adopting libsignal is not enough'. In: *USENIX Security 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, Aug. 2022, pp. 269–286. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/albrecht.

19. Martin R. Albrecht, Miro Haller, Lenka Mareková and Kenneth G. Paterson. 'Caveat Implementor! Key Recovery Attacks on MEGA'. In: *EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. LNCS. Springer, Cham, Apr. 2023, pp. 190–218. DOI: 10.1007/978-3-031-30589-4_7.

20. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson and Igors Stepanovs. 'Four Attacks and a Proof for Telegram'. In: *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2022, pp. 87–106. DOI: 10.1109/SP46214.2022.9833666.

21. Martin R. Albrecht, Sunoo Park, Michael A. Specter and Douglas Stebila. *A Real-World Law-Enforcement Hack: The Case of Encrochat*. Cryptology ePrint Archive, Report 2026/1319. 2026. URL: https://eprint.iacr.org/2026/1319.

22. Martin R. Albrecht and Kenneth G. Paterson. 'Analyzing Cryptography in the Wild: A Retrospective'. In: *IEEE Secur. Priv.* 22.6 (2024), pp. 12–18. DOI: 10.1109/MSEC.2024.3441764.

23. Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering and Jacob C. N. Schuldt. 'On the Security of RC4 in TLS'. In: *USENIX Security 2013*. Ed. by Samuel T. King. USENIX Association, Aug. 2013, pp. 305–320. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan.

24. Nadhem J. AlFardan and Kenneth G. Paterson. 'Lucky Thirteen: Breaking the TLS and DTLS Record Protocols'. In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

25. FIDO Alliance. *FIDO Authentication: A Passwordless Vision*. https://fidoalliance.org/fido2/. Accessed on 2024-01-25. Jan. 2019.

26. Leonardo C. Almeida, Ewerton R. Andrade, Paulo S. L. M. Barreto and Marcos A. Simplício Jr. 'Lyra: password-based key derivation with tunable memory and processing costs'. In: *Journal of Cryptographic Engineering* 4.2 (June 2014), pp. 75–89. DOI: 10.1007/s13389-013-0063-5.

27. ambiso. *NaCL – should I keep track of expired nonces*. Cryptography Stack Exchange, https://crypto.stackexchange.com/questions/106375. May 2023. URL: https://crypto.stackexchange.com/questions/106375/nacl-should-i-keep-track-of-expired-nonces.

28. AppBrain. *Threema - Android app on AppBrain*. en. Oct. 2022. URL: https://www.appbrain.com/app/threema/ch.threema.app (visited on 17/05/2022).

29. Apple Support. *Advanced Data Protection for iCloud*. https://support.apple.com/guide/security/advanced-data-protection-for-icloud-sec973254c5f/web. Accessed: 2025-08-25. Sept. 2024.

30. Scott Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*. Internet-Draft draft-irtf-cfrg-xchacha-03, IRTF CFRG. Jan. 2020. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha-03.

31. 2FAS Auth. *How to use/sync more devices with 2FAS?* https://2fas.com/support/2fas-mobile-app/how-to-use-sync-more-devices-with-2fas/. Accessed on 2024-01-12. Jan. 2024.

32. Nimrod Aviram et al. 'DROWN: Breaking TLS Using SSLv2'. In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, Aug. 2016, pp. 689–706. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram.

33. Gildas Avoine, Xavier Carpent and Diane Leblanc-Albarel. *In the Vault, But Not Safe: Exploring the Threat of Covert Password Manager Providers*. Cryptology ePrint Archive, Report 2025/1278. 2025. URL: https://eprint.iacr.org/2025/1278.

34. Kasey Babcock. *Security Through Transparency: ETH Zurich Audits Bitwarden Cryptography*. https://bitwarden.com/blog/security-through-transparency-eth-zurich-audits-bitwarden-cryptography/. Accessed: 2026-07-08. 2026.

35. Matilda Backendal, Mihir Bellare, Felix Günther and Matteo Scarlata. 'When Messages Are Keys: Is HMAC a Dual-PRF?' In: *CRYPTO 2023, Part III*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. LNCS. Springer, Cham, Aug. 2023, pp. 661–693. DOI: 10.1007/978-3-031-38548-3_22.

36. Matilda Backendal, Sebastian Clermont, Marc Fischlin and Felix Günther. 'Key Derivation Functions Without a Grain of Salt'. In: *EUROCRYPT 2025, Part VIII*. Ed. by Serge Fehr and Pierre-Alain Fouque. Vol. 15608. LNCS. Springer, Cham, May 2025, pp. 393–426. DOI: 10.1007/978-3-031-91101-9_14.

37. Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller and Kenneth G. Paterson. 'A Formal Treatment of End-to-End Encrypted Cloud Storage'. In: *CRYPTO 2024, Part II*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14921. LNCS. Springer, Cham, Aug. 2024, pp. 40–74. DOI: 10.1007/978-3-031-68379-4_2.

38. Matilda Backendal, Miro Haller and Kenneth G. Paterson. 'MEGA: Malleable Encryption Goes Awry'. In: *2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2023, pp. 146–163. DOI: 10.1109/SP46215.2023.10179290.

39. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol*. RFC 9420 (Proposed Standard). July 2023. DOI: 10.17487/RFC9420. URL: https://www.rfc-editor.org/rfc/rfc9420.

40. Richard L. Barnes, Karthikeyan Bhargavan, Benjamin Lipp and Christopher A. Wood. *Hybrid Public Key Encryption*. RFC 9180 (Informational). Feb. 2022. DOI: 10.17487/RFC9180. URL: https://www.rfc-editor.org/rfc/rfc9180.

41. Beem Development. *Aegis Authenticator*. Software application. Accessed on 2024-01-12. URL: https://getaegis.app/.

42. A. Belenko and D Sklyarov. *"Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really?* Tech. rep. Elcomsoft Co. Ltd., 2012.

43. Mihir Bellare. 'New Proofs for NMAC and HMAC: Security without Collision-Resistance'. In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Springer, Berlin, Heidelberg, Aug. 2006, pp. 602–619. DOI: 10.1007/11818175_36.

44. Mihir Bellare, Ran Canetti and Hugo Krawczyk. 'Keying Hash Functions for Message Authentication'. In: *CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. LNCS. Springer, Berlin, Heidelberg, Aug. 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1.

45. Mihir Bellare, Shay Gueron, Viet Tung Hoang, Julia Len, Sanketh Menda and Thomas Ristenpart. *Building the Next Generation of AEAD*. Talk at Real World Crypto Symposium 2024, Toronto, Canada. Mar. 2024. URL: https://rwc.iacr.org/2024/program.php.

46. J. Berger. *openMittsu*. Sept. 2016. URL: https://github.com/blizzard4591/openMittsu (visited on 09/09/2022).

47. Daniel J Bernstein. *Cryptography in NaCl*. 2009. URL: https://cr.yp.to/highspeed/naclcrypto-20090310.pdf.

48. Daniel J. Bernstein. *Cache-timing attacks on AES*. Manuscript. 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

49. Daniel J. Bernstein. *CurveCP: Usable security for the Internet*. 2011. URL: https://curvecp.org/.

50. Daniel J. Bernstein. *High-speed high-security cryptography: encrypting and authenticating the whole Internet*. Invited lecture at the 27th Chaos Communication Congress (27C3), Berlin, Germany. Dec. 2010. URL: https://cr.yp.to/talks.html#2010.12.28.

51. Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub and Jean Karim Zinzindohoue. 'A Messy State of the Union: Taming the Composite State Machines of TLS'. In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 535–552. DOI: 10.1109/SP.2015.39.

52. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti and Pierre-Yves Strub. 'Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS'. In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 98–113. DOI: 10.1109/SP.2014.14.

53. Karthikeyan Bhargavan and Gaëtan Leurent. 'On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN'. In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers and Shai Halevi. ACM Press, Oct. 2016, pp. 456–467. DOI: 10.1145/2976749.2978423.

54. Karthikeyan Bhargavan and Gaëtan Leurent. 'Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH'. In: *NDSS 2016*. The Internet Society, Feb. 2016. DOI: 10.14722/ndss.2016.23418.

55. Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves and Douglas Stebila. 'Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange'. In: *PQCrypto 2019*. Ed. by Jintai Ding and Rainer Steinwandt. Springer, Cham, 2019, pp. 206–226. DOI: 10.1007/978-3-030-25510-7_12.

56. Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. 'Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications'. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31.

57. Bitwarden. *Bitwarden: About Us*. https://bitwarden.com/about/. Accessed: 2025-08-26.

58. Bitwarden. *Bitwarden Clients Source Code*. https://github.com/bitwarden/clients. Accessed: 2025-03-25.

59. Bitwarden. *How End-to-End Encryption Paves the Way for Zero Knowledge - White Paper*. https://bitwarden.com/pdf/resources-zero-knowledge-encryption-white-paper.pdf. Accessed: 2025-03-25. 2025.

60. John Black, Phillip Rogaway and Thomas Shrimpton. 'Encryption-Scheme Security in the Presence of Key-Dependent Messages'. In: *SAC 2002*. Ed. by Kaisa Nyberg and Howard M. Heys. Vol. 2595. LNCS. Springer, Berlin, Heidelberg, Aug. 2003, pp. 62–75. DOI: 10.1007/3-540-36492-7_6.

61. Daniel Bleichenbacher. 'Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1'. In: *CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Berlin, Heidelberg, Aug. 1998, pp. 1–12. DOI: 10.1007/BFb0055716.

62. Hanno Böck, Juraj Somorovsky and Craig Young. 'Return Of Bleichenbacher's Oracle Threat (ROBOT)'. In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, Aug. 2018, pp. 817–849. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

63. Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky and Philipp Jovanovic. *Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS*. Cryptology ePrint Archive, Report 2016/475. 2016. URL: https://eprint.iacr.org/2016/475.

64. Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Draft, version 0.6. 2023. URL: https://toc.cryptobook.us/.

65. Nikita Borisov, Ian Goldberg and Eric Brewer. 'Off-the-record communication, or, why not to use PGP'. In: *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*. WPES '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 77–84. DOI: 10.1145/1029179.1029200.

66. Nikita Borisov, Ian Goldberg and Eric A. Brewer. 'Off-the-record communication, or, why not to use PGP'. In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*. Ed. by Vijay Atluri, Paul F. Syverson and Sabrina De Capitani di Vimercati. ACM, 2004, pp. 77–84. DOI: 10.1145/1029179.1029200.

67. Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. Dec. 2020. DOI: 10.17487/RFC8949. URL: https://www.rfc-editor.org/info/rfc8949.

68. Xavier Boyen, Stanislaw Jarecki, Phillip Nazarian, Jiayu Xu and Tianyu Zheng. 'Two-Factor Authentication Can Harden Servers Against Offline Password Search'. In: *Advances in Cryptology – EUROCRYPT 2026*. Ed. by Joan Daemen and Emmanuel Thomé. Cham: Springer Nature Switzerland, 2026, pp. 128–158. ISBN: 978-3-032-25317-0.

69. Joachim Breitner and Nadia Heninger. 'Biased Nonce Sense: Lattice Attacks Against Weak ECDSA Signatures in Cryptocurrencies'. In: *FC 2019*. Ed. by Ian Goldberg and Tyler Moore. Vol. 11598. LNCS. Springer, Cham, Feb. 2019, pp. 3–20. DOI: 10.1007/978-3-030-32101-7_1.

70. Thomas Brewster. *Microsoft Gave FBI Keys to Unlock BitLocker Encrypted Data*. Forbes. Jan. 2026. URL: https://www.forbes.com/sites/thomasbrewster/2026/01/22/microsoft-gave-fbi-keys-to-unlock-bitlocker-encrypted-data/.

71. Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk and Sebastian Schinzel. 'ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication'. In: *USENIX Security 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 4293–4310. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/brinkmann.

72. Lara Bruseghini, Daniel Huigens and Kenneth G. Paterson. 'Victory by KO: Attacking OpenPGP Using Key Overwriting'. In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers and Elaine Shi. ACM Press, Nov. 2022, pp. 411–423. DOI: 10.1145/3548606.3559363.

73. Tim Cappalli and Matthew Miller. *Passkey Developer Resources*. en-US. https://passkeys.dev/. Accessed on 2023-09-19.

74. Michael Carr and Siamak F. Shahandashti. 'Revisiting Security Vulnerabilities in Commercial Password Managers'. In: *ICT Systems Security and Privacy Protection - 35th IFIP TC 11 International Conference, SEC 2020, Maribor, Slovenia, September 21-23, 2020, Proceedings*. Ed. by Marko Hölbl, Kai Rannenberg and Tatjana Welzer. Vol. 580. IFIP Advances in Information and Communication Technology. Springer, 2020, pp. 265–279. DOI: 10.1007/978-3-030-58201-2_18.

75. Sunil Chaudhary, Tiina Schafeitel-Tähtinen, Marko Helenius and Eleni Berki. 'Usability, security and trust in password managers: A quest for user-centric properties and features'. In: *Comput. Sci. Rev.* 33 (2019), pp. 69–90. DOI: 10.1016/J.COSREV.2019.03.002.

76. Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla and Hovav Shacham. 'A Systematic Analysis of the Juniper Dual EC Incident'. In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers and Shai Halevi. ACM Press, Oct. 2016, pp. 468–479. DOI: 10.1145/2976749.2978395.

77. Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella and Daniel Genkin. 'GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers'. In: *USENIX Security 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru.

78. Sonia Chiasson, P. C. van Oorschot and Robert Biddle. 'A usability study and critique of two password managers'. In: *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, 2006.

79. Dave Childers. *State of the Auth*. https://duo.com/assets/ebooks/state-of-the-auth-2021.pdf. Accessed on 2023-10-19. Sept. 2021.

80. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt and Douglas Stebila. 'A Formal Security Analysis of the Signal Messaging Protocol'. In: *2017 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, Apr. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27.

81. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt and Douglas Stebila. 'A Formal Security Analysis of the Signal Messaging Protocol'. In: *J. Cryptol.* 33.4 (2020), pp. 1914–1983. DOI: 10.1007/s00145-020-09360-1.

82. WinZip Computing. *AES Encryption Information: Encryption Specification AE-1 and AE-2*. en. 2009. URL: https://www.winzip.com/win/en/aes_info.html (visited on 04/07/2022).

83. Graeme Connell. *Technology Deep Dive: Building a Faster ORAM Layer for Enclaves*. Aug. 2022. URL: https://signal.org/blog/building-faster-oram (visited on 11/09/2022).

84. Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman and Raluca Ada Popa. 'Secret Key Recovery in a Global-Scale End-to-End Encryption System'. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 703–719. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/connell.

85. Deirdre Connolly, David Adrian and Thomas Ptacek. *Threema with Kenny Paterson, Matteo Scarlata, & Kien Tuong Truong*. Security, Cryptography, Whatever (podcast). Jan. 2023. URL: https://securitycryptographywhatever.com/2023/01/27/threema/.

86. Deirdre Connolly and Duncan Wong. *Post-Quantum and Post-Quantum/Traditional Hybrid Algorithms for HPKE*. Internet-Draft draft-ietf-hpke-pq-05, IETF. 2025. URL: https://datatracker.ietf.org/doc/html/draft-ietf-hpke-pq-05.

87. Cas Cremers, Jaiden Fairoze, Benjamin Kiesl and Aurora Naska. 'Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice'. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1481–1495. DOI: 10.1145/3372297.3423354.

88. Cybersecurity and Infrastructure Security Agency. *Cybersecurity Best Practices*. https://www.cisa.gov/topics/cybersecurity-best-practices. Accessed on 2023-10-19. Oct. 2023.

89. Nicola Dardanis. 'Bridging the Gap: Design and Implementation of Secure Shared Folders'. Supervised by Kenneth G. Paterson, Matilda Backendal, and Matteo Scarlata. Master's thesis. ETH Zurich, Oct. 2024. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/masters-thesis_nicola-dardanis.pdf.

90. Dashlane. *2023 Report - Password Health Scores Around the World*. https://www.dashlane.com/up-loads/2023/10/2023-Global-Password-Health-Score-Report-1.pdf. Accessed: 2025-08-26.

91. Dashlane. *Dashlane Android Client Source Code*. https://github.com/Dashlane/android-apps. Accessed: 2025-03-25.

92. Dashlane. *Dashlane Browser Extension Client Source Code*. https://github.com/Dashlane/dashlane-web-extension. Accessed: 2025-03-25.

93. Dashlane. *Dashlane CLI Client Source Code*. https://github.com/Dashlane/dashlane-cli. Accessed: 2025-03-25.

94. Dashlane. *Dashlane's Security Principles & Architecture*. https://www.dashlane.com/download/whitepaper-en.pdf. Accessed: 2025-03-25.

95. Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth and Tibor Jager. 'Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol'. In: *CRYPTO 2023, Part IV*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14084. LNCS. Springer, Cham, Aug. 2023, pp. 330–361. DOI: 10.1007/978-3-031-38551-3_11.

96. Jean Paul Degabriele, Alessandro Melloni, Jean-Pierre Münch and Martijn Stam. 'Counter Galois Onion (sfCGO): Fast Non-malleable Onion Encryption for Tor'. In: *EUROCRYPT 2026, Part VI*. Ed. by Joan Daemen and Emmanuel Thomé. Vol. 16546. LNCS. Springer, Cham, May 2026, pp. 304–334. DOI: 10.1007/978-3-032-25333-0_11.

97. Jean Paul Degabriele, Alessandro Melloni and Martijn Stam. 'Secure Onion Encryption and the Case of Counter Galois Onion'. In: *EUROCRYPT 2026, Part VI*. Ed. by Joan Daemen and Emmanuel Thomé. Vol. 16546. LNCS. Springer, Cham, May 2026, pp. 335–363. DOI: 10.1007/978-3-032-25333-0_12.

98. Frank Denis. *The Sodium Cryptography Library (libsodium)*. Software library documentation. URL: https://libsodium.gitbook.io/doc.

99. Jacob DePriest and Andrew Hall. *Zero knowledge vs. a malicious server: A look at ETH Zurich's research*. https://1password.com/blog/eth-zurich-zero-knowledge-malicious-server-review. Accessed: 2026-07-08. 2026.

100. Peter Deutsch. *DEFLATE Compressed Data Format Specification Version 1.3*. RFC 1951. May 1996. DOI: 10.17487/RFC1951. URL: https://www.rfc-editor.org/rfc/rfc1951.

101. Peter Deutsch. *GZIP File Format Specification Version 4.3*. RFC 1952. May 1996. DOI: 10.17487/RFC1952. URL: https://www.rfc-editor.org/rfc/rfc1952.

102. Whitfield Diffie and Martin E. Hellman. 'New Directions in Cryptography'. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

103. Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart and Joanne Woodage. 'Fast Message Franking: From Invisible Salamanders to Encryptment'. en. In: *Advances in Cryptology – CRYPTO 2018*. Ed. by Hovav Shacham and Alexandra Boldyreva. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 155–186. ISBN: 9783319968841. DOI: 10.1007/978-3-319-96884-1_6.

104. Jason A. Donenfeld. 'WireGuard: Next Generation Kernel Network Tunnel'. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

105. Benjamin Dowling, Felix Günther and Alexandre Poirrier. 'Continuous Authentication in Secure Messaging'. In: *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen and Weizhi Meng. Vol. 13555. Lecture Notes in Computer Science. Springer, 2022, pp. 361–381. DOI: 10.1007/978-3-031-17146-8_18.

106. Benjamin Dowling, Torben Brandt Hansen and Kenneth G. Paterson. 'Many a Mickle Makes a Muckle: A Framework for Provably Quantum-Secure Hybrid Key Exchange'. In: *PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Cham, 2020, pp. 483–502. DOI: 10.1007/978-3-030-44223-1_26.

107. Benjamin Dowling and Kenneth G. Paterson. 'A Cryptographic Analysis of the WireGuard Protocol'. In: *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. Lecture Notes in Computer Science. Springer, 2018, pp. 3–21. DOI: 10.1007/978-3-319-93387-0_1.

108. Benjamin Dowling, Paul Rösler and Jörg Schwenk. 'Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework'. In: *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part I*. Ed. by Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden and Vassilis Zikas. Vol. 12110. Lecture Notes in Computer Science. Springer, 2020, pp. 341–373. DOI: 10.1007/978-3-030-45374-9_12.

109. Yihe Duan, Ding Wang and Yanduo Fu. 'Security Analysis of Master-Password-Protected Password Management Protocols'. In: *2025 IEEE Symposium on Security and Privacy*. Ed. by Marina Blanton, William Enck and Cristina Nita-Rotaru. IEEE Computer Society Press, May 2025, pp. 701–719. DOI: 10.1109/SP61157.2025.00050.

110. Thai Duong and Juliano Rizzo. 'Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET'. In: *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2011, pp. 481–489. DOI: 10.1109/SP.2011.42.

111. Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer and Vern Paxson. 'The Matter of Heartbleed'. In: *Proceedings of the 2014 Internet Measurement Conference (IMC)*. CVE-2014-0160. ACM, 2014, pp. 475–488. DOI: 10.1145/2663716.2663755.

112. Kit Eaton. *These Apps Promise to Encrypt Your Smartphone Communications*. en-US. Mar. 2016. URL: https://www.nytimes.com/2016/03/24/technology/personaltech/encryption-by-app-adds-security-to-smartphones.html (visited on 17/05/2022).

113. ENISA. *Coordinated Vulnerability Disclosure Policies in the EU*. Tech. rep. European Union Agency for Cybersecurity (ENISA), Apr. 2022. URL: https://www.enisa.europa.eu/publications/coordinated-vulnerability-disclosure-policies-in-the-eu.

114. European Telecommunications Standards Institute (ETSI). *Quantum-Safe Hybrid Key Exchanges*. Technical Specification TS 103 744 v1.1.1. ETSI, Dec. 2020. URL: https://www.etsi.org/deliver/etsi_ts/103700_103799/103744/01.01.01_60/ts_103744v010101p.pdf.

115. Andrés Fábrega, Julia Len, Thomas Ristenpart and Gregory Rubin. 'Random-Access AEAD for Fast Lightweight Online Encryption'. In: *EUROCRYPT 2026, Part VI*. Ed. by Joan Daemen and Emmanuel Thomé. Vol. 16546. LNCS. Springer, Cham, May 2026, pp. 273–303. DOI: 10.1007/978-3-032-25333-0_10.

116. Andrés Fábrega, Armin Namavari, Rachit Agarwal, Ben Nassi and Thomas Ristenpart. 'Exploiting Leakage in Password Managers via Injection Attacks'. In: *USENIX Security 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/fabrega.

117. Michael Fagan, Yusuf Albayram, Mohammad Maifi Hasan Khan and Ross Buck. 'An investigation into users' considerations towards using password managers'. In: *Hum. centric Comput. Inf. Sci.* 7 (2017), p. 12. DOI: 10.1186/S13673-017-0093-6.

118. Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders and Matthew Smith. 'Hey, You, Get Off of My Clipboard - On How Usability Trumps Security in Android Password Managers'. In: *FC 2013*. Ed. by Ahmad-Reza Sadeghi. Vol. 7859. LNCS. Springer, Berlin, Heidelberg, Apr. 2013, pp. 144–161. DOI: 10.1007/978-3-642-39884-1_12.

119. Corin Faife. *Swiss Army drops WhatsApp for homegrown messaging service, citing privacy concerns*. en. Jan. 2022. URL: https://www.theverge.com/2022/1/7/22871881/swiss-army-whatsapp-messaging-threema-privacy-concerns-us-jurisdiction (visited on 16/05/2022).

120. Marc Fischlin and Felix Günther. 'Multi-Stage Key Exchange and the Case of Google's QUIC Protocol'. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by Gail-Joon Ahn, Moti Yung and Ninghui Li. ACM, 2014, pp. 1193–1204. DOI: 10.1145/2660267.2660308.

121. Lorenzo Franceschi-Bicchierai. *Snowden's Favorite Chat App Is Coming to Your Computer*. VICE Motherboard. Dec. 2015. URL: https://www.vice.com/en/article/signal-snowdens-favorite-chat-app-is-coming-to-your-computer/.

122. Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart and Nicola Dell. '"A Stalker's Paradise": How Intimate Partner Abusers Exploit Technology'. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*. Ed. by Regan L. Mandryk, Mark Hancock, Mark Perry and Anna L. Cox. ACM, 2018, p. 667. DOI: 10.1145/3173574.3174241.

123. Michael J. Freedman, Yuval Ishai, Benny Pinkas and Omer Reingold. 'Keyword Search and Oblivious Pseudorandom Functions'. In: *TCC 2005*. Ed. by Joe Kilian. Vol. 3378. LNCS. Springer, Berlin, Heidelberg, Feb. 2005, pp. 303–324. DOI: 10.1007/978-3-540-30576-7_17.

124. Rose de Fremery. *How Zero Knowledge Keeps Passwords Safe*. LastPass Blog. 2023. URL: https://blog.lastpass.com/posts/how-zero-knowledge-keeps-passwords-safe.

125. Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk and Thorsten Holz. 'How Secure is TextSecure?' In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 457–472. DOI: 10.1109/EuroSP.2016.41.

126. Georg Fuchsbauer, Eike Kiltz and Julian Loss. 'The Algebraic Group Model and its Applications'. In: *CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. LNCS. Springer, Cham, Aug. 2018, pp. 33–62. DOI: 10.1007/978-3-319-96881-0_2.

127. Christina Garman, Kenneth G. Paterson and Thyla van der Merwe. 'Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS'. In: *USENIX Security 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 113–128. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/garman.

128. Paolo Gasti and Kasper Bonne Rasmussen. 'On the Security of Password Manager Database Formats'. In: *ESORICS 2012*. Ed. by Sara Foresti, Moti Yung and Fabio Martinelli. Vol. 7459. LNCS. Springer, Berlin, Heidelberg, Sept. 2012, pp. 770–787. DOI: 10.1007/978-3-642-33167-1_44.

129. Gabriel K. Gegenhuber, Moritz Grefner, Maximilian Günther, Matthäus Wininger, David Schmidt and Aljosha Judmayer. 'Send and Pretend: Exploiting Transcript Consistency Issues in End-to-End Encrypted Group Chats'. In: *CoRR* abs/2607.27510 (2026). DOI: 10.48550/ARXIV.2607.27510. arXiv: 2607.27510. URL: https://doi.org/10.48550/arXiv.2607.27510.

130. Jessica Gentles, Mason Fields, Garrett Goodman and Suman Bhunia. *Breaking the Vault: A Case Study of the 2022 LastPass Data Breach*. https://arxiv.org/abs/2502.04287. Accessed: 2025-03-25.

131. Paul Gerhart, Paul Rösler and Dominique Schröder. *Security Analysis of Ibex*. July 2023. URL: https://threema.com/assets/6-resources/audits/security_analysis_ibex_2023.pdf.

132. Federico Giacon, Felix Heuer and Bertram Poettering. 'KEM Combiners'. In: *PKC 2018, Part I*. Ed. by Michel Abdalla and Ricardo Dahab. Vol. 10769. LNCS. Springer, Cham, Mar. 2018, pp. 190–218. DOI: 10.1007/978-3-319-76578-5_7.

133. Yoel Gluck, Neal Harris and Angelo Prado. *BREACH: Reviving the CRIME Attack*. Presented at Black Hat USA 2013. 2013. URL: https://www.breachattack.com/resources/BREACH%5C%20-%5C%20SSL,%5C%20gone%5C%20in%5C%2030%5C%20seconds.pdf.

134. Threema GmbH. *About – Threema*. 2022. URL: https://threema.ch/en/about.

135. Threema GmbH. *Cryptography Whitepaper*. en. Nov. 2021. URL: https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf (visited on 19/05/2022).

136. Threema GmbH. *Cryptography Whitepaper*. en. Apr. 2026. URL: https://threema.com/assets/8-press/2026-04-20-threema-cryptography-whitepaper.pdf.

137. Threema GmbH. *GitHub - threema-ch/threema-android: Threema App for Android*. en. Dec. 2020. URL: https://github.com/threema-ch/threema-android (visited on 09/09/2022).

138. Threema GmbH. *Threema Changelog*. en. 2022. URL: https://threema.ch/en/versionhistory (visited on 11/07/2022).

139. Threema GmbH. *Threema Provides Maximum Security and Comprehensive Privacy Protection*. en. 2022. URL: https://threema.ch/en/security (visited on 06/07/2022).

140. Threema GmbH. *Threema.OnPrem*. en. 2022. URL: https://threema.ch/en/onprem (visited on 06/07/2022).

141. Threema GmbH. *Version 5.0*. Commit 316dfd22. Nov. 2022. URL: https://github.com/threema-ch/threema-android/commit/316dfd22 (visited on 09/09/2022).

142. Threema GmbH. *Why Threema Instead of WhatsApp?* en. May 2021. URL: https://threema.ch/en/work/blog/posts/why-threema-instead-of-whatsapp (visited on 17/05/2022).

143. Sharon Goldberg, Miro Haller, Nadia Heninger, Mike Milano, Dan Shumow, Marc Stevens and Adam Suhl. 'RADIUS/UDP Considered Harmful'. In: *USENIX Security 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/goldberg.

144. Shafi Goldwasser and Silvio Micali. 'Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information'. In: *14th ACM STOC*. ACM Press, May 1982, pp. 365–377. DOI: 10.1145/800070.802212.

145. Bill Goodwin. *Spanish Court Acquits Suspects Denied Access to Raw Sky ECC Intercepts in Landmark Decision*. Computer Weekly. On the 2021 law-enforcement interception of the Sky ECC encrypted phone service; archived at https://archive.is/nem1q. 2025. URL: https://www.computerweekly.com/news/366637705/Spanish-court-acquits-suspects-denied-access-to-raw-Sky-ECC-intercepts-in-landmark-decision.

146. Google. *Tink: A Multi-Language, Cross-Platform Cryptographic Library*. Software library. 2024. URL: https://github.com/tink-crypto/tink.

147. Google. *Transfer your Google Authenticator codes*. https://support.google.com/accounts/answer/1066447. Accessed on 2024-01-12. Apr. 2023.

148. Glenn Greenwald, Ewen MacAskill, Laura Poitras, Spencer Ackerman and Dominic Rushe. *Microsoft Handed the NSA Access to Encrypted Messages*. The Guardian. July 2013. URL: https://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data.

149. GSMA. *Our History*. GSMA. Accessed 17 July 2026. URL: https://www.gsma.com/about-us/who-we-are/our-history/.

150. Shay Gueron. *I Want to Encrypt 2^{64} Bytes with AES-GCM Using a Single Key*. Talk at Real World Crypto Symposium 2024, Toronto, Canada. Mar. 2024. URL: https://rwc.iacr.org/2024/program.php.

151. Shay Gueron and Thomas Ristenpart. *DNDK: Combining Nonce and Key Derivation for Fast and Scalable AEAD*. Cryptology ePrint Archive, Report 2025/785. 2025. URL: https://eprint.iacr.org/2025/785.

152. Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko and Thomas Schneider. 'All the Numbers are US: Large-scale Abuse of Contact Discovery in Mobile Messengers'. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/all-the-numbers-are-us-large-scale-abuse-of-contact-discovery-in-mobile-messengers/.

153. Helena Handschuh and Bart Preneel. 'Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms'. In: *CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, Berlin, Heidelberg, Aug. 2008, pp. 144–161. DOI: 10.1007/978-3-540-85174-5_9.

154. Christopher Harrell. *YubiKeys, passkeys and the future of modern authentication*. https://www.yubico.com/blog/passkeys-and-the-future-of-modern-authentication/. Accessed on 2024-01-25. Mar. 2022.

155. Sven Hebrok, Simon Nachtigall, Marcel Maehren, Nurullah Erinola, Robert Merget, Juraj Somorovsky and Jörg Schwenk. 'We Really Need to Talk About Session Tickets: A Large-Scale Analysis of Cryptographic Dangers with TLS Session Tickets'. In: *USENIX Security 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, Aug. 2023, pp. 4877–4894. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/hebrok.

156. Sven Hebrok, Tim Leonhard Storm, Felix Matthias Cramer, Maximilian Radoy and Juraj Somorovsky. 'STEK Sharing is Not Caring: Bypassing TLS Authentication in Web Servers using Session Tickets'. In: *USENIX Security 2025*. Ed. by Lujo Bauer and Giancarlo Pellegrino. USENIX Association, Aug. 2025, pp. 8017–8034. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/hebrok.

157. M. Heiderich, M. Wege and C. Kean. *Pentest- And Audit-Report Threema Mobile Apps*. Oct. 2020. URL: https://threema.ch/press-files/2_documentation/security_audit_report_threema_2020.pdf.

158. Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman. 'Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices'. In: *USENIX Security 2012*. Ed. by Tadayoshi Kohno. USENIX Association, Aug. 2012, pp. 205–220. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger.

159. Nadia Heninger and Keegan Ryan. 'The Hidden Number Problem with Small Unknown Multipliers: Cryptanalyzing MEGA in Six Queries and Other Applications'. In: *PKC 2023, Part I*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. Vol. 13940. LNCS. Springer, Cham, May 2023, pp. 147–176. DOI: 10.1007/978-3-031-31368-4_6.

160. J. Hodges, J.C. Jones, M.B. Jones, A. Kumar and E. Lundberg. *Web Authentication: An API for accessing Public Key Credentials*. https://www.w3.org/TR/webauthn/. Accessed on 2024-01-25. Apr. 2021.

161. Jonas Hofmann and Kien Tuong Truong. 'End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem'. In: *ACM CCS 2024*. Ed. by Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda and David Lie. ACM Press, Oct. 2024, pp. 3988–4001. DOI: 10.1145/3658644.3690309.

162. Nicolas Huaman, Sabrina Amft, Marten Oltrogge, Yasemin Acar and Sascha Fahl. 'They Would do Better if They Worked Together: The Case of Interaction Problems Between Password Managers and Websites'. In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 1367–1381. DOI: 10.1109/SP40001.2021.00094.

163. David A. Huffman. 'A Method for the Construction of Minimum-Redundancy Codes'. In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

164. IEEE. *IEEE Standard Specification for Password-Based Public-Key Cryptographic Techniques*. Tech. rep. IEEE Std 1363.2-2008. DLAPKAS-SRP6. IEEE, Jan. 2009. DOI: 10.1109/IEEESTD.2009.4773330.

165. Valentina-Florentina Iliescu. 'Multi-Device Password Hardening'. MA thesis. ETH Zurich, Sept. 2024. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/masters-thesis_valentina-iliescu.pdf.

166. Russell Impagliazzo. 'A Personal View of Average-Case Complexity'. In: *Proceedings of the Tenth Annual Structure in Complexity Theory Conference, Minneapolis, Minnesota, USA, June 19-22, 1995*. IEEE Computer Society, 1995, pp. 134–147. DOI: 10.1109/SCT.1995.514853.

167. Apple Inc. *Apple Platform Security*. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf. Accessed on 2023-12-08. May 2021.

168. Bitwarden Inc. *Bitwarden Security*. https://bitwarden.com/help/bitwarden-security-white-paper. Accessed on 2024-01-11. Feb. 2023.

169. Microsoft Threat Intelligence. *Frozen in transit: Secret Blizzard's AiTM campaign against diplomats*. https://www.microsoft.com/en-us/security/blog/2025/07/31/frozen-in-transit-secret-blizzards-aitm-campaign-against-diplomats/. 2025.

170. Fabian Ising, Damian Poddebniak and Sebastian Schinzel. *Security Audit Report Threema 2019*. Mar. 2019. URL: https://threema.ch/press-files/2_documentation/security_audit_report_threema_2019.pdf.

171. Tibor Jager, Florian Kohlar, Sven Schäge and Jörg Schwenk. 'On the Security of TLS-DHE in the Standard Model'. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 273–293. DOI: 10.1007/978-3-642-32009-5_17.

172. Tibor Jager, Kenneth G. Paterson and Juraj Somorovsky. 'One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography'. In: *NDSS 2013*. The Internet Society, Feb. 2013.

173. Tibor Jager, Jörg Schwenk and Juraj Somorovsky. 'Practical Invalid Curve Attacks on TLS-ECDH'. In: *ESORICS 2015, Part I*. Ed. by Günther Pernul, Peter Y. A. Ryan and Edgar R. Weippl. Vol. 9326. LNCS. Springer, Cham, Sept. 2015, pp. 407–425. DOI: 10.1007/978-3-319-24174-6_21.

174. Stanislaw Jarecki, Hugo Krawczyk and Jiayu Xu. 'OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks'. In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Cham, Apr. 2018, pp. 456–486. DOI: 10.1007/978-3-319-78372-7_15.

175. Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr and Paul F. Syverson. 'Users get routed: traffic correlation on Tor by realistic adversaries'. In: *ACM CCS 2013*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor and Moti Yung. ACM Press, Nov. 2013, pp. 337–348. DOI: 10.1145/2508859.2516651.

176. Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032 (Informational). Jan. 2017. DOI: 10.17487/RFC8032. URL: https://www.rfc-editor.org/rfc/rfc8032.

177. Ari Juels and Thomas Ristenpart. 'Honey Encryption: Security Beyond the Brute-Force Bound'. In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Berlin, Heidelberg, May 2014, pp. 293–310. DOI: 10.1007/978-3-642-55220-5_17.

178. Ambarish Karole, Nitesh Saxena and Nicolas Christin. 'A Comparative Usability Evaluation of Traditional Password Managers'. In: *Information Security and Cryptology - ICISC 2010 - 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers*. Ed. by Kyung Hyune Rhee and DaeHun Nyang. Vol. 6829. Lecture Notes in Computer Science. Springer, 2010, pp. 233–251. DOI: 10.1007/978-3-642-24209-0_16.

179. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Third. Chapman and Hall, CRC Press, 2014. ISBN: 978-0815354369.

180. John Kelsey. 'Compression and Information Leakage of Plaintext'. In: *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*. Ed. by Joan Daemen and Vincent Rijmen. Vol. 2365. Lecture Notes in Computer Science. Springer, 2002, pp. 263–276. DOI: 10.1007/3-540-45661-9_21.

181. John Kelsey, Niels Ferguson, Bruce Schneier and Mike Stay. *Cryptanalytic Progress: Lessons for AES*. Public comment on the AES finalists, NIST. May 2000. URL: https://csrc.nist.rip/encryption/aes/round2/comments/20000501-jkelsey-1.pdf.

182. H. Kikuchi, M. Tada and S. Nakanishi. 'Secure instant messaging protocol preserving confidentiality against administrator'. In: *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004*. Vol. 2. Mar. 2004, 27–30 Vol.2. DOI: 10.1109/AINA.2004.1283749.

183. Vlastimil Klima and Tomas Rosa. *Attack on Private Signature Keys of the OpenPGP Format, PGP(TM) Programs and Other Applications Compatible with OpenPGP*. Cryptology ePrint Archive, Report 2002/076. 2002. URL: https://eprint.iacr.org/2002/076.

184. Nadim Kobeissi, Karthikeyan Bhargavan and Bruno Blanchet. 'Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach'. In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38.

185. Hugo Krawczyk. 'Cryptographic Extraction and Key Derivation: The HKDF Scheme'. In: *CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. LNCS. Springer, Berlin, Heidelberg, Aug. 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34.

186. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

187. Brian Krebs. 'Password Manager LastPass Warns of Breach'. In: *Krebs on Security* (June 2015). Accessed: 2025-08-25.

188. Brian A. LaMacchia, Kristin E. Lauter and Anton Mityagin. 'Stronger Security of Authenticated Key Exchange'. In: *Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings*. Ed. by Willy Susilo, Joseph K. Liu and Yi Mu. Vol. 4784. Lecture Notes in Computer Science. Springer, 2007, pp. 1–16. DOI: 10.1007/978-3-540-75670-5_1.

189. Adam Langley. *Apple's SSL/TLS Bug*. ImperialViolet blog; CVE-2014-1266. Feb. 2014. URL: https://www.imperialviolet.org/2014/02/22/applebug.html.

190. LastPass. *Knowledge workers have a false sense of password security*. https://www.lastpass.com/-/media/7e2b8b715dc44d23ae21c5cf94dc4876.pdf. LastPass Business Report; Accessed: August 25, 2025. 2025.

191. Sean Lawlor and Kevin Lewi. *Deploying key transparency at WhatsApp*. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/. 2023.

192. Zhiwei Li, Warren He, Devdatta Akhawe and Dawn Song. 'The Emperor's New Password Manager: Security Analysis of Web-based Password Managers'. In: *USENIX Security 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, Aug. 2014, pp. 465–479. DOI: 10.21236/ada614474. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei.

193. Srikanth Lingala. *Zip4j*. en. URL: https://github.com/srikanth-lingala/zip4j.

194. Tianren Liu, Angelos Pelecanos, Stefano Tessaro and Vinod Vaikuntanathan. 'Layout Graphs, Random Walks and the t-Wise Independence of SPN Block Ciphers'. In: *CRYPTO 2023, Part III*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. LNCS. Springer, Cham, Aug. 2023, pp. 694–726. DOI: 10.1007/978-3-031-38548-3_23.

195. Tianren Liu, Stefano Tessaro and Vinod Vaikuntanathan. 'The t-wise Independence of Substitution-Permutation Networks'. In: *CRYPTO 2021, Part IV*. Ed. by Tal Malkin and Chris Peikert. Vol. 12828. LNCS. Virtual Event: Springer, Cham, Aug. 2021, pp. 454–483. DOI: 10.1007/978-3-030-84259-8_16.

196. Signal Messenger LLC. *libsignal*. URL: https://github.com/signalapp/libsignal (visited on 08/07/2022).

197. David M'Raihi, Frank Hoornaert, David Naccache, Mihir Bellare and Ohad Ranen. *HOTP: An HMAC-Based One-Time Password Algorithm*. RFC 4226. Dec. 2005. DOI: 10.17487/RFC4226. URL: https://www.rfc-editor.org/info/rfc4226.

198. David M'Raihi, Johan Rydell, Mingliang Pei and Salah Machani. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. May 2011. DOI: 10.17487/RFC6238. URL: https://www.rfc-editor.org/info/rfc6238.

199. Internet Archive Wayback Machine. *KeePass Plugins and Extensions*. https://web.archive.org/web/20150106032133/https://keepass.info/plugins.html. Accessed on 2024-01-11. Jan. 2015.

200. Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky and Jörg Schwenk. 'TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries'. In: *USENIX Security 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, Aug. 2022, pp. 215–232. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/maehren.

201. Robertas Maleckas, Kenneth G. Paterson and Martin R. Albrecht. *Practically-exploitable Vulnerabilities in the Jitsi Video Conferencing System*. Cryptology ePrint Archive, Report 2023/1118. 2023. URL: https://eprint.iacr.org/2023/1118.

202. Itsik Mantin. 'Predicting and Distinguishing Attacks on RC4 Keystream Generator'. In: *EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. LNCS. Springer, Berlin, Heidelberg, May 2005, pp. 491–506. DOI: 10.1007/11426639_29.

203. Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. https://signal.org/blog/private-contact-discovery. Sept. 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 01/08/2022).

204. Moxie Marlinspike. *The New TextSecure: Privacy Beyond SMS*. Open Whisper Systems Blog, https://whispersystems.org/blog/the-new-textsecure/. Accessed via Wayback Machine. Feb. 2014. URL: https://web.archive.org/web/20170226002239/https://whispersystems.org/blog/the-new-textsecure/.

205. Moxie Marlinspike. *WhatsApp's Signal Protocol integration is now complete*. https://signal.org/blog/whatsapp-complete. 4th Apr. 2016. URL: https://signal.org/blog/whatsapp-complete/ (visited on 28/07/2022).

206. Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. Revision 1. Apr. 2016. URL: https://signal.org/docs/specifications/x3dh/.

207. Ian Martiny, Gabriel Kaptchuk, Adam J. Aviv, Daniel S. Roche and Eric Wustrow. 'Improving Signal's Sealed Sender'. In: *NDSS 2021*. The Internet Society, Feb. 2021. DOI: 10.14722/ndss.2021.24180.

208. Peter Mayer, Collins W. Munyendo, Michelle L. Mazurek and Adam J. Aviv. 'Why Users (Don't) Use Password Managers at a Large Educational Institution'. In: *USENIX Security 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, Aug. 2022, pp. 1849–1866. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/mayer.

209. Daniel McCarney, David Barrera, Jeremy Clark, Sonia Chiasson and Paul C. van Oorschot. 'Tapas: design, implementation, and usability evaluation of a password manager'. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 89–98. ISBN: 9781450313124. DOI: 10.1145/2420950.2420964.

210. Mega Ltd. *MEGA Security White Paper – 3rd Ed*. https://mega.nz/SecurityWhitepaper.pdf. June 2022.

211. Sanketh Menda, Mihir Bellare, Viet Tung Hoang, Julia Len and Thomas Ristenpart. 'The OCH Authenticated Encryption Scheme'. In: *ACM CCS 2025*. Ed. by Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie and Véronique Cortier. ACM Press, Oct. 2025, pp. 2234–2248. DOI: 10.1145/3719027.3765224.

212. Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann and Jörg Schwenk. *Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)*. Cryptology ePrint Archive, Report 2020/1151. 2020. URL: https://eprint.iacr.org/2020/1151.

213. Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel and Erik Tews. 'Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks'. In: *USENIX Security 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, Aug. 2014, pp. 733–748. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer.

214. Microsoft. *BitLocker overview*. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/. Accessed on 2023-10-19. Oct. 2023.

215. MITRE. *CVE-2023-22899*. Jan. 2023. URL: https://www.cve.org/CVERecord?id=CVE-2023-22899 (visited on 10/01/2023).

216. Bodo Möller, Thai Duong and Krzysztof Kotowicz. *This POODLE Bites: Exploiting the SSL 3.0 Fallback*. Google Security Advisory; CVE-2014-3566. Sept. 2014. URL: https://openssl-library.org/files/ssl-poodle.pdf.

217. Vivek Nair. *MFKDF GitHub Repository*. en-US. https://github.com/multifactor/MFKDF. Accessed on 2024-01-12. Aug. 2023.

218. Vivek Nair and Dawn Song. 'Decentralizing Custodial Wallets with MFKDF'. In: *2023 IEEE International Conference on Blockchain and Cryptocurrency*. 2023. DOI: 10.1109/ICBC56567.2023.10174998.

219. Vivek Nair and Dawn Song. 'Multi-Factor Credential Hashing for Asymmetric Brute-Force Attack Resistance'. In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023, pp. 56–72. DOI: 10.1109/EuroSP57164.2023.00013.

220. Vivek Nair and Dawn Song. 'Multi-Factor Key Derivation Function (MFKDF) for Fast, Flexible, Secure, & Practical Key Management'. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/nair-mfkdf.

221. Vivek Nair and Dawn Song. *Multi-Factor Key Derivation Function (MFKDF) for Fast, Flexible, Secure, & Practical Key Management (Full Version)*. https://arxiv.org/abs/2208.05586v3. 2023. arXiv: 2208.05586 [cs.CR].

222. Moni Naor and Omer Reingold. 'Number-theoretic Constructions of Efficient Pseudo-random Functions'. In: *38th FOCS*. IEEE Computer Society Press, Oct. 1997, pp. 458–467. DOI: 10.1109/SFCS.1997.646134.

223. NordPass. *People have around 170 passwords on average, study shows*. https://www.globenewswire.com/news-release/2024/05/21/2885556/0/en/People-have-around-170-passwords-on-average-study-shows.html. Press Release, Commissioned by NordPass, Conducted by Cint. Accessed: 2025-08-25. May 2024. (Visited on 23/05/2024).

224. Sean Oesch and Scott Ruoti. 'That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers'. In: *USENIX Security 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 2165–2182. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/oesch.

225. Sean Oesch, Scott Ruoti, James Simmons and Anuj Gautam. '"It Basically Started Using Me:" An Observational Study of Password Manager Usage'. In: *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022*. Ed. by Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, David A. Shamma, Steven Mark Drucker, Julie R. Williamson and Koji Yatani. ACM, 2022, 33:1–33:23. DOI: 10.1145/3491102.3517534.

226. Paul C. van Oorschot and Michael J. Wiener. 'Parallel Collision Search with Cryptanalytic Applications'. In: *J. Cryptol.* 12.1 (1999), pp. 1–28. DOI: 10.1007/PL00003816.

227. Kenneth G. Paterson and Thyla van der Merwe. 'Reactive and Proactive Standardisation of TLS'. In: *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*. Ed. by Lidong Chen, David A. McGrew and Chris J. Mitchell. Vol. 10074. Lecture Notes in Computer Science. Springer, 2016, pp. 160–186. DOI: 10.1007/978-3-319-49100-4_7.

228. Kenneth G. Paterson, Matteo Scarlata and Kien T. Truong. 'Three Lessons From Threema: Analysis of a Secure Messenger'. In: *USENIX Security 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, Aug. 2023, pp. 1289–1306. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/paterson.

229. Kenneth G. Paterson, Matteo Scarlata and Kien Tuong Truong. 'Three Lessons From Threema: Analysis of a Secure Messenger'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1289–1306. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/paterson.

230. Kenneth G. Paterson, Matteo Scarlata and Kien Tuong Truong. *Three Lessons From Threema: Analysis of a Secure Messenger (Full Version)*. 2022. URL: https://breakingthe3ma.app/files/Threema-PST22.pdf.

231. Kenneth G. Paterson and Arnold K. L. Yau. *Cryptography in Theory and Practice: The Case of Encryption in IPsec*. Cryptology ePrint Archive, Report 2005/416. 2005. URL: https://eprint.iacr.org/2005/416.

232. Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin and Lorrie Faith Cranor. 'Why people (don't) use password managers effectively'. In: *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 319–338. ISBN: 978-1-939133-05-2. URL: https://www.usenix.org/conference/soups2019/presentation/pearman.

233. Colin Percival. *Cache Missing for Fun and Profit*. Presented at BSDCan 2005. 2005. URL: https://www.daemonology.net/papers/htt.pdf.

234. Colin Percival. *Stronger key derivation via sequential memory-hard functions*. https://www.tarsnap.com/scrypt/scrypt.pdf. 2009.

235. Trevor Perrin. *The Noise Protocol Framework*. Version 34. July 2018. URL: http://www.noiseprotocol.org/noise.html (visited on 08/07/2022).

236. Trevor Perrin. *X25519 and Zero Outputs*. Message on the CFRG "curves" mailing list. Archived at https://web.archive.org/web/20260514024659/https://moderncrypto.org/mail-archive/curves/2017/000896.html. May 2017. URL: https://moderncrypto.org/mail-archive/curves/2017/000896.html.

237. Meta Platforms. *WhatsApp Encryption Overview*. Version 6. 15th Nov. 2021. URL: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf (visited on 22/09/2022).

238. Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979 (Informational). Aug. 2013. DOI: 10.17487/RFC6979. URL: https://www.rfc-editor.org/rfc/rfc6979.

239. Mario Di Raimondo, Rosario Gennaro and Hugo Krawczyk. 'Secure off-the-record messaging'. In: *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005*. Ed. by Vijay Atluri, Sabrina De Capitani di Vimercati and Roger Dingledine. ACM, 2005, pp. 81–89. DOI: 10.1145/1102199.1102216.

240. Jason Rasmussen. *Details on Hardening in Response to ETH Zurich Reported Security Issues*. https://blog.lastpass.com/posts/details-on-hardening-in-response-to-eth-zurich-reported-security-issues. Accessed: 2026-07-08. 2026.

241. Hirak Ray, Flynn Wolf, Ravi Kuber and Adam J. Aviv. 'Why Older Adults (Don't) Use Password Managers'. In: *USENIX Security 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 73–90. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ray.

242. Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida and Herbert Bos. 'Flip Feng Shui: Hammering a Needle in the Software Stack'. In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, Aug. 2016, pp. 1–18. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

243. Real World Cryptography Symposium. *RWC 2026 Accepted Talks*. https://rwc.iacr.org/2026/acceptedtalks.php. Accessed: 2026-07-08. 2026.

244. Oded Regev. 'On lattices, learning with errors, random linear codes, and cryptography'. In: *37th ACM STOC*. Ed. by Harold N. Gabow and Ronald Fagin. ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603.

245. Dominik Reichl. *KeePass Password Safe*. https://keepass.info/index.html. Accessed on 2024-01-11. Oct. 2023.

246. Dominik Reichl. *Keepass Plugins: OtpKeyProv*. https://keepass.info/plugins.html. Accessed on 2024-01-11. June 2023.

247. Eric Rescorla, Marsh Ray, Steve Dispensa and Nikos Oskov. *Transport Layer Security (TLS) Renegotiation Indication Extension*. RFC 5746 (Proposed Standard); addresses CVE-2009-3555. Feb. 2010. DOI: 10.17487/RFC5746. URL: https://www.rfc-editor.org/rfc/rfc5746.

248. Johannes Ritter. *Was Russland stört, überzeugt Olaf Scholz*. de. Jan. 2022. URL: https://m.faz.net/aktuell/wirtschaft/unternehmen/threema-was-russland-stoert-ueberzeugt-olaf-scholz-18248712.amp.html (visited on 17/08/2022).

249. Frederic Rivain. *Testing Zero Knowledge Against a Malicious Server*. https://www.dashlane.com/blog/zero-knowledge-malicious-server. Accessed: 2026-07-08. 2026.

250. Juliano Rizzo and Thai Duong. *Browser Exploit Against SSL/TLS*. CVE-2011-3389. 2011. URL: http://packetstormsecurity.com/files/105499/Browser-Exploit-Against-SSL-TLS.html.

251. Juliano Rizzo and Thai Duong. *The CRIME Attack*. 2012. URL: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU.

252. Colin Roberts and Vivek Nair. *Grant Application – MFKDF2*. Zcash Community Forum. 2025. URL: https://forum.zcashcommunity.com/t/grant-application-mfkdf2/51614.

253. Colin Roberts, Vivek Nair and Dawn Song. *Wrangling Entropy: Next-Generation Multi-Factor Key Derivation, Credential Hashing, and Credential Generation Functions*. Cryptology ePrint Archive. Sept. 2025. URL: https://arxiv.org/abs/2509.05893.

254. Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong and Yuval Yarom. 'The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations'. In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 435–452. DOI: 10.1109/SP.2019.00062.

255. Paul Rösler, Christian Mainka and Jörg Schwenk. 'More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema'. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036.

256. Ben Rush. *KeeChallenge: A plugin for KeePass2 to add Yubikey challenge-response capability*. https://keepass.info/index.html. Accessed on 2024-01-11. May 2016.

257. Glenn Greenwald Ryan Gallagher. *How the NSA Plans to Infect 'Millions' of Computers with Malware*. https://theintercept.com/2014/03/12/nsa-plans-infect-millions-computers-malware/. 2014.

258. Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Nov. 2015. DOI: 10.17487/RFC7693. URL: https://www.rfc-editor.org/info/rfc7693.

259. Matteo Scarlata, Matilda Backendal and Miro Haller. 'MFKDF: Multiple Factors Knocked Down Flat'. In: *USENIX Security 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/scarlata.

260. Matteo Scarlata, Matilda Backendal and Miro Haller. *MFKDF: Multiple Factors Knocked Down Flat*. Cryptology ePrint Archive, Report 2024/935. 2024. URL: https://eprint.iacr.org/2024/935.

261. Matteo Scarlata, Giovanni Torrisi, Matilda Backendal and Kenneth G. Paterson. 'Zero Knowledge (About) Encryption: A Comparative Security Analysis of Three Cloud-based Password Managers'. In: *35th USENIX Security Symposium (USENIX Security 2026)*. USENIX Association, Aug. 2026. URL: https://www.usenix.org/conference/usenixsecurity26.

262. Matteo Scarlata, Giovanni Torrisi, Matilda Backendal and Kenneth G. Paterson. *Zero Knowledge (About) Encryption: A Comparative Security Analysis of Three Cloud-based Password Managers*. Cryptology ePrint Archive, Report 2026/058. 2026. URL: https://eprint.iacr.org/2026/058.

263. Michel Schreiner and Thomas Hess. 'Examining the Role of Privacy in Virtual Migration: The Case of WhatsApp and Threema'. In: *21st Americas Conference on Information Systems, AMCIS 2015, Puerto Rico, August 13-15, 2015*. Association for Information Systems, 2015. URL: http://aisel.aisnet.org/amcis2015/ISSecurity/GeneralPresentations/33.

264. Security.org Team. *Password Manager Annual Report*. https://www.security.org/digital-safety/password-manager-annual-report/. Also titled: Password Manager Industry Report and Market Outlook (2023-2024). Accessed: 2024-11-14. 2024.

265. Adi Shamir. 'How to Share a Secret'. In: *Communications of the Association for Computing Machinery* 22.11 (Nov. 1979), pp. 612–613. DOI: 10.1145/359168.359176.

266. Tanusree Sharma, Vivek C Nair, Henry Wang, Yang Wang and Dawn Song. '"I Can't Believe It's Not Custodial!": Usable Trustless Decentralized Key Management'. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642464.

267. Yaron Sheffer, Dick Hardt and Michael B. Jones. *JSON Web Token Best Current Practices*. RFC 8725. Feb. 2020. DOI: 10.17487/RFC8725. URL: https://www.rfc-editor.org/info/rfc8725.

268. Victor Shoup. 'Lower Bounds for Discrete Logarithms and Related Problems'. In: *EUROCRYPT'97*. Ed. by Walter Fumy. Vol. 1233. LNCS. Springer, Berlin, Heidelberg, May 1997, pp. 256–266. DOI: 10.1007/3-540-69053-0_18.

269. Jochen Siegle. *Schutz beim Chatten: Threema wird noch sicherer*. de-CH. July 2019. URL: https://www.nzz.ch/digital/whatsapp-alternative-threema-wird-noch-sicherer-ld.1498576 (visited on 17/05/2022).

270. David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen and Collin Jackson. 'Password Managers: Attacks and Defenses'. In: *USENIX Security 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, Aug. 2014, pp. 449–464. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/silver.

271. Soatok. *Threema: Three Strikes, You're Out*. en-US. Nov. 2021. URL: https://soatok.blog/2021/11/05/threema-three-strikes-youre-out/ (visited on 09/09/2022).

272. Yuanming Song, Lenka Mareková and Kenneth G. Paterson. *Criminology: Refined Techniques for Compression Side-Channel Attacks*. Cryptology ePrint Archive, Paper 2026/1972. 2026. URL: https://eprint.iacr.org/2026/1972.

273. SRF. *Threema setzt sich durch - Schweizer Armee verbietet Whatsapp und Co.* de. Jan. 2022. URL: https://www.srf.ch/news/schweiz/threema-setzt-sich-durch-schweizer-armee-verbietet-whatsapp-und-co (visited on 17/05/2022).

274. Ben Stock and Martin Johns. 'Protecting users against XSS-based password manager abuse'. In: *ASIACCS 14*. Ed. by Shiho Moriai, Trent Jaeger and Kouichi Sakurai. ACM Press, June 2014, pp. 183–194. DOI: 10.1145/2590296.2590336.

275. Andris Suter-Dörig, Matteo Scarlata and Kenneth G. Paterson. 'Delegating Email Encryption to Gateways: Why Johnny Should Not'. In: *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. To appear. ACM, 2026.

276. The Matrix.org Team. *Post-mortem and Remediations for Apr 11 Security Incident*. Matrix.org blog. Attacker gained access to Matrix production servers and databases via a Jenkins vulnerability, and defaced the project website. May 2019. URL: https://matrix.org/blog/2019/05/08/post-mortem-and-remediations-for-apr-11-security-incident/.

277. Threema GmbH. *Threema for iOS: Preview of Upcoming Multi-Device Functionality*. Threema Blog. Dec. 2022. URL: https://threema.com/en/blog/ios-test-multi-device.

278. Karim Toubba. *Security Incident Update and Recommended Actions*. 2023.

279. Kien Tuong Truong, Simon-Philipp Merz, Matteo Scarlata, Felix Günther and Kenneth G. Paterson. 'Breaking and Fixing Content-Defined Chunking'. In: *ACM CCS 2025*. Ed. by Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie and Véronique Cortier. ACM Press, Oct. 2025, pp. 2294–2308. DOI: 10.1145/3719027.3744870.

280. Kien Tuong Truong, Noemi Terzo and Kenneth G. Paterson. 'Signal Lost (Integrity): The Signal App is More than the Sum of its Protocols'. In: *IACR Cryptol. ePrint Arch.* 2026 (2026), p. 484. URL: https://eprint.iacr.org/2026/484.

281. Andreas Tsouloupas. 'Breaking Cryptography in the Wild: Double-Ratchet Mutations'. Master's thesis. ETH Zurich, Sept. 2023. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/masters-thesis_andreas-tsouloupas.pdf.

282. Filippo Valsorda. *XAES-256-GCM*. C2SP specification. June 2024. URL: https://c2sp.org/XAES-256-GCM.

283. Mathy Vanhoef and Frank Piessens. 'All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS'. In: *USENIX Security 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 97–112. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/vanhoef.

284. Serge Vaudenay. 'Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS...' In: *EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. LNCS. Springer, Berlin, Heidelberg, Apr. 2002, pp. 534–546. DOI: 10.1007/3-540-46035-7_35.

285. WhatsApp. *WhatsApp Help Center - Answering your questions about WhatsApp's January 2021 Privacy Policy update*. en. Jan. 2021. URL: https://faq.whatsapp.com/general/security-and-privacy/answering-your-questions-about-whatsapps-privacy-policy/?lang=en (visited on 17/05/2022).

286. Wilson Center. *The Bybit Heist: What Happened, What Now?* Wilson Center commentary. Attack executed via malicious JavaScript served from the compromised Safe{Wallet} front-end, so that cold-wallet signers approved a transaction different from the one displayed. 2025. URL: https://www.wilsoncenter.org/article/bybit-heist-what-happened-what-now.

287. Xiaowen Xin. *Titan M makes Pixel 3 our most secure phone yet*. https://blog.google/products/pixel/titan-m-\makes-pixel-3-our-most-secure-phone-yet/. Accessed on 2023-10-19. Oct. 2018.

288. Frances F. Yao and Yiqun Lisa Yin. 'Design and Analysis of Password-Based Key Derivation Functions'. In: *CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. LNCS. Springer, Berlin, Heidelberg, Feb. 2005, pp. 245–261. DOI: 10.1007/978-3-540-30574-3_17.

289. Yuval Yarom, Daniel Genkin and Nadia Heninger. 'CacheBleed: A Timing Attack on OpenSSL Constant Time RSA'. In: *CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. LNCS. Springer, Berlin, Heidelberg, Aug. 2016, pp. 346–367. DOI: 10.1007/978-3-662-53140-2_17.

290. Yubico. *Using Your YubiKey with KeePass*. https://support.yubico.com/hc/articles/360013779759-Using-Your-YubiKey-with-KeePass. Accessed on 2024-01-11. May 2020.

291. Wenyi Morty Zhang, Annie Dai, Keegan Ryan, Dave Levin, Nadia Heninger and Aaron Schulman. 'Don't Look Up: There Are Sensitive Internal Links in the Clear on GEO Satellites'. In: *ACM CCS 2025*. Ed. by Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie and Véronique Cortier. ACM Press, Oct. 2025, pp. 3960–3974. DOI: 10.1145/3719027.3765198.

292. Rui Zhao, Chuan Yue and Kun Sun. 'A Security Analysis of Two Commercial Browser and Cloud Based Password Managers'. In: *International Conference on Social Computing, SocialCom 2013, SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013, Washington, DC, USA, 8-14 September, 2013*. IEEE Computer Society, 2013, pp. 448–453. DOI: 10.1109/SOCIALCOM.2013.70.

293. Yuliang Zheng. 'Digital Signcryption or How to Achieve Cost(Signature & Encryption) ≪ Cost(Signature) + Cost(Encryption)'. In: *CRYPTO'97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. LNCS. Springer, Berlin, Heidelberg, Aug. 1997, pp. 165–179. DOI: 10.1007/BFb0052234.

294. Jacob Ziv and Abraham Lempel. 'A Universal Algorithm for Sequential Data Compression'. In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

CV

Acknowledgments

