Security Analysis of Master-Password-Protected Password Management Protocols

Abstract—**Password managers (PMs) are useful tools that help users manage their login credentials, alleviating the burden of memorizing an ever-increasing number of passwords. Master-password-protected password management (M3PM) protocols characterize the interaction between the client and the PM's server. In this protocol, the client uses the master password for authentication, and the server assists in retrieving credentials across devices. Given the ongoing PM data breaches and users' concerns about potential server misuse, it is crucial for the server to remain oblivious to both the master password and the credentials. The pivotal role of M3PM protocols underscores the need for a systematic and formal security analysis.**

In this paper, we, *for the first time*, present an extensive formal analysis of M3PM protocols. We identify the de facto M3PM protocols from 43 PMs in industry and academia by defining a methodology that includes documentation analysis, traffic analysis, and reverse engineering. To formalize the security properties of M3PM protocols, we propose a set of ideal functionalities within the universal composability (UC) framework. We categorize offline guessing attacks on master passwords into four types based on the knowledge of the adversary, and our analysis shows that 38 of the 43 PMs are vulnerable to at least one type of offline guessing attack, demonstrating the circumstances under which various M3PM protocols with single master password protection fail to resist such attacks. Additionally, we identify an oracle attack against the well-known open-source Passbolt and demonstrate that 1Password's dual-key mechanism provides strong protection for users' master passwords and credentials.

1. Introduction

Passwords stubbornly remain the most prevalent authentication method and are unlikely to be replaced in the foreseeable future [3], [5], [17], [18]. The average number of accounts that users need to maintain has increased to 80-107 [1], [28], [44]. This brings great burdens for users to manage (e.g., create/remember/type) passwords. In response, users often adopt vulnerable behaviors such as choosing popular passwords (e.g., `123456` and `password`) [56], reusing passwords [43], and creating passwords containing personally identifiable information (PII) [55], which makes password vulnerable to damaging password guessing attacks [36], [43], [56], [57]. Besides, there are usability issues, e.g., typos often occur when typing passwords, especially on mobile devices. In a nutshell, managing a large number of passwords is a great challenge for common human users.

Password managers (PMs), tools for helping users manage their credentials, are promising to alleviate this issue. Benefiting from the free of human memorization, users can generate strong (e.g., long and containing various characters) and unique passwords for each credential by PMs. Users can *store* their credentials into PMs, and unlock the PMs to *get* their passwords when they need. PMs help users avoid vulnerable behaviors, and reduce the cognitive burden caused by memorizing multiple passwords, striking a balance between password security and usability [22]. Therefore, PMs are highly recommended by standard bodies (e.g., NIST [4] and NCSC [2]) and security experts [29], [38], [63].

The security of PMs, given their centralization of sensitive user credentials, has become a serious concern for both industry [47], [53] and academia [24], [25], [35], [42], [51]. LastPass [7] experienced data breaches three times within three years [10], [52]. While LastPass asserts that no passwords were compromised by attackers in 2022, this breach has recently been found disastrous and linked to a theft of $4.4 million in cryptocurrency [10]. Furthermore, other potential security threats have been identified as deterrents for users in adopting PMs [11], [37], [41], [45], [46]. Users not only fear data breaches in PMs but also exhibit limited trust in entrusting all their credentials to a third party [11], [45]. To instill more trust, several PMs assert that they are "zero-knowledge" regarding users' password vaults [13], [33]. This term implies that the PMs only store encrypted vaults without possessing knowledge or access to the actual contents of users' vaults.

In general, users are required to create a master password to authenticate and gain access to manage password vaults. We characterize the interaction between the client and the PM's server as the *Master-Password-Protected Password Management (M3PM)* protocol. We define M3PM as a cryptographic protocol that authenticates the client and ensures the password vault remains oblivious to the server side. It is important to emphasize that secure encryption of password vaults does not necessarily ensure a secure M3PM protocol. The inappropriate combination of authentication and encryption mechanisms can introduce vulnerability (see Sec. 5.4), and M3PM protocols encompass broader security properties beyond secure encryption (see Sec. 4.1). To the best of our knowledge, this is the first study to systematically identify and analyze the security of M3PM protocols. We focus on three research questions (RQs):

- **RQ1**: What M3PM protocols have been adopted by existing password managers?

- **RQ2**: What defines a secure M3PM protocol?
- **RQ3**: Are existing password managers using M3PM protocols secure under our security definitions?

To address these questions, we have developed a methodology to understand the status quo of M3PM protocols. This involves collecting and refining information about M3PM protocols from 43 major password managers sourced from academia and industry. We summarize the security properties of M3PM protocols and formalize functionalities to describe the perspectives of M3PM protocols within the universal composability (UC) framework [19]. This enables a systematic analysis of the collected M3PM protocols to identify security vulnerabilities related to offline guessing attacks, distinguishability, and malleability. In summary, our contributions include:

- **A methodology for identifying M3PM protocols.** We introduce a systematic methodology for identifying M3PM protocols across 36 popular industrial PMs and seven representative academic PMs. Information about these protocols is sourced from three primary channels: documentation, data logs, and application code. We use an iterative process of collection and verification to accurately identify information on M3PM protocols, cryptographic algorithms, and metadata encryption.
- **Ideal functionalities of M3PM protocols.** We formalize the security properties of M3PM protocols and present an ideal functionality $\mathcal{F}_{M3PM}$ within the UC framework. During our analysis of the collected M3PM protocols, we iteratively revise and extend this functionality to distinguish between explicit authentication $\mathcal{F}_{M3PM-EA}$ and implicit authentication $\mathcal{F}_{M3PM-IA}$. Inspired by previous work on password-related protocols [9], [20], [23], [27], [31], we introduce a relaxed functionality $\mathcal{F}_{rM3PM}$ to address guessing attacks and present $\mathcal{F}_{adM3PM}$ to accommodate protocols with additional secrets.
- **Security analysis for M3PM protocols.** We categorize four types of offline master password guessing attacks, and find 38 of 43 PMs cannot resist at least one type of offline guessing attack. We summarize the circumstances that each M3PM protocol fail to resist offline guessing attacks and conclude that the current design of M3PM protocols, without additional secrets or devices, suffers from the low entropy of master passwords and heavily relies on the effectiveness of slow hash or memory-hard functions. Additionally, we find fully plaintext communication in Delinea Web, and uncover an oracle attack where a corrupted server can learn the encryption key of Passbolt. Overall, we give provable analysis of eight distinct M3PM protocols and prove the M3PM protocol of 1Password and Multipassword realize $\mathcal{F}_{adM3PM-EA}$.

2. Preliminary

2.1. Password managers

Password managers (PMs) are tools that help users manage their accounts. The basic management function of PMs is to memorize passwords so that users can effortlessly delegate their password management responsibilities to the PMs, retrieving passwords as needed. For modern PMs, the functions of managing users' passwords are diverse. PMs offer the capability to generate random passwords, and users can tailor password settings, including length and character composition. During the login process on respective websites, the autofill feature in PMs automatically populates users' credentials, alleviating the need for manual typing. This enhanced convenience attributed to autofill is a major driver behind the adoption of PMs [45]. Furthermore, PMs employ tools like the password strength meter, such as Zxcvbn [59], to assess and identify weak passwords. Additionally, PMs utilize compromised credential checking services, such as Have I Been Pwned [6], to identify and alert users about any passwords that may have been compromised or leaked.

In this work, we focus on the basic management function, i.e., memorizing passwords. Depending on the way of memorizing passwords, PMs can be categorized into two types [39]: (1) Retrieval PMs (e.g., 1Password and LastPass) store users' credentials in digital memory, commonly referred to as *password vaults*. These credentials are then retrieved as needed during the user authentication process; (2) Generative PMs (e.g., LessPass and SPHINX [49]) create users' credentials on demand by inputting the master password and additional information such as URLs and usernames into deterministic functions, such as OPRF [49] or Hash functions [48]. The result is the creation of distinct passwords for different user accounts. In essence, retrieval PMs store encrypted user-chosen credentials, like an encrypted database, whereas generative PMs generate passwords on behalf of different accounts.

2.2. Related work

In 2012, Gasti and Rasmussen [25] analyzed the storage format of password vaults. They gave two security definitions for password vaults: (1) Indistinguishability of databases game (IND-CDBA) corresponding to passive attackers; (2) Malleability of chosen database game (MAL-CDBA) corresponding to active attackers. According to their analysis, unencrypted metadata would leak users' privacy and give attackers the advantage of distinguishing users' password vaults and tampering vaults without integrity checks. Their work is the first rigorous analysis for PMs, but it analyzed the storage format instead of the M3PM protocol, which neglects authentication of retrieval password vaults and offline guessing attacks of the master passwords. Furthermore, their work analyzed 13 PMs in industry, while we give a more systematic analysis across 36 industrial PMs and seven academic PMs. In 2020, Oesch and Ruoti [42] replicated Gasti and Rasmussen's [25] study, finding PMs have improved in protecting metadata but still leave some plaintext vulnerable. Replicating their analysis, we find that after three years, 30 out of 43 PMs still leave plaintext metadata (details in Sec. 5.2).

In 2013, Zhao et al. [65] analyzed the storage of two cloud-based PMs and found that insecure storage methods (e.g., the insecure relationship between the master password and local authenticator) can leave the password vaults vulnerable. In the same year, Zhao and Yue [64] showed five browser-based PMs would be cracked once the encrypted information is stolen. These works pointed out the potential vulnerability of offline guessing attacks on master passwords and gave us great insights to conduct a systematic and rigorous analysis.

Bojinov et al. [15] and Chatterjee et al. [21] highlighted that encryption using the master password alone cannot withstand offline guessing attacks in the event of a leakage of the encrypted password vault. They proposed novel schemes, called honey vaults, to resist offline guessing attacks on leaked vaults. Inspired by their work, we formalize the process of offline guessing attacks and emphasize that the adversary's prior knowledge significantly influences the ability to attack honey vaults. (see details in Section 5.1)

In 2023, Davies et al. [23] formally analyzed WhatsApp's backup protocol and proposed a password-protected key retrieval (PPKR) functionality in the UC framework. Our M3PM functionalities share similarities with the PPKR functionality, but our definition encompasses three main differences from theirs: (1) The PPKR functionality employs one password to retrieve one key, whereas the M3PM functionality utilizes one master password mpw to retrieve multiple passwords. While this modification seems subtle, our functionalities are designed to ensure the uniqueness of passwords. This distinction is crucial as a secure M3PM protocol must safeguard one password even if other passwords are leaked. (2) The M3PM functionality distinguishes explicit and implicit authentication. (3) The PPKR functionality allows an adversary to guess offline ten times, while the relaxed M3PM functionality does not restrict the times specifically, which gives credit to the methods described in Appendix 6.1.

3. M3PM protocols collection

To address **RQ1:** *What M3PM protocols have been adopted by existing password managers?* we identify the diverse M3PM protocols in use. In this section, we elaborate on the process by which we select 43 PMs and collect information regarding the M3PM for each PM.

3.1. PMs selection

To comprehensively analyze various password management schemes, we exhaustively identify PMs from academic and industrial sources in September and October 2023.

For industrial PMs, we first collect PMs that are popular with users. We create core search terms (i.e., "password manager", "password vault", "password management", and "password store") and enter each core search term as a query into the search box on the Chrome Web Store, Google Play Store, Firefox Add-ons, Edge Add-ons, and Apple's App Store. We keep the top 30 PMs displayed in each query and remove duplicated PMs. To avoid overlooking popular PMs, we include those recommended annually by sites like passwordmanager.com, U.S. News, PCMag, Cybernews, CNET, Tom's Guide, Forbes, and Security.org.

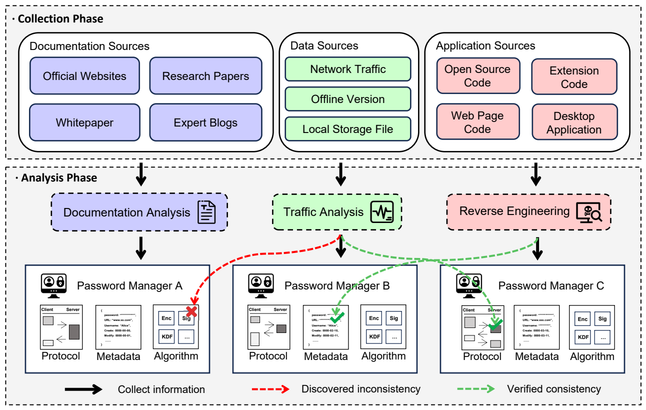


Figure 1: An overview of our methodology includes documentation analysis, traffic analysis, and reverse engineering. We collect information from documentation, data logs, and application codes. Documentation analysis helps gather initial data, while traffic analysis and reverse engineering are used to collect additional information and verify the consistency of the documentation.

Leveraging the aforementioned core search terms, we utilize Github to select relevant open-source PMs and retain the PMs with over one thousand stars. We consider PMs with a substantial number of stars on GitHub, as this metric indicates a stronger appeal and approval among developers. Given GitHub's community-driven nature, the number of stars reflects the PM's popularity and credibility within the developer community. All these above steps resulted in 145 PMs from the industry in total.

For academic PMs, we use the above core search terms to search in DBLP and Google Scholar, and we are only concerned about papers creating new M3PM protocols. Finally, we collect 15 PM schemes published in ICDCS, Securecomm, AsiaCCS, Computer&Security, ESORICS, IEEE S&P, SOUPS, WWW, Usenix Sec, TDSC, JCST, ACSAC, UbiComp, and NSPW.

In the course of our analysis, we refine the pool of candidate PMs by eliminating those with fewer than 10,000 installations, academic PMs with a primary focus other than security, and PMs that cannot be set up or launched (confirmed as vendor-related unavailability by all authors). This process results in a final selection of 43 PMs for in-depth research, and we summarize all PMs in Table 1.

3.2. Collection methodology

In Fig. 1, we present an overview of our collection methodology. We gather information from various sources, including documentation sources, data sources, and application sources. All this information is publicly available, and we do not employ it for any exploitation of PMs.

Documentation sources. For documentation sources, we use the names of corresponding PMs as core search terms in popular search engines, Google and Bing. According to the results of search engines, we collect every descriptive documentation of PMs, including industrial whitepapers, README files of open-source PMs, expert blogs, and other support documentation on PMs' official websites.

We first search for the descriptions about security and encryption on each documentation. Subsequently, we extract the relevant information about algorithms and protocols from the identified documentation. For example, Bitwarden has detailed information about its security architecture[1]. This includes specifics on how Bitwarden authenticates users and the encryption methods employed for storing users' credentials while excluding details on other security aspects like password sharing and single sign-on.

For PMs without sufficient public information, we collect the limited available information and formulate sensible and conservative predictions. For example, Norton PM's official website only reveals that they use AES-256 as an encryption method and encrypt the password vaults; Then, we record this encryption method and determine that Norton is a retrieval PM. All the information we extract is prepared for subsequent verification, and our goal is not an exhaustive collection of all online documentation but rather acquiring a sufficient amount for further in-depth research.

Data sources. For data sources, we use the web page version of PMs and install browser extensions on Edge and Chrome. For each PM, we first create an account (we delete these accounts after research to avoid unnecessary burdens for PM servers). Furthermore, we use Burp Suite[2] to record the HTTPS traffic. Following the account creation, we proceed with a series of essential operations, including logging in, opening the vault, adding credentials, changing credentials, deleting credentials, and logging out. In the subsequent content, we refer to these operations collectively as *PM operations*, considering them fundamental actions for collecting information on M3PM protocols.

Our collection of plaintext network traffic without TLS is aimed at discerning the communication patterns associated with M3PM protocols. Our lack of emphasis on the TLS serves two specific reasons: First, in the context of M3PM protocols, the server is assumed to be malicious, rendering the TLS between a client and a malicious server ineffective; Second, our results also give an understanding about the extent to which PMs can safeguard users' password vaults in the event of TLS compromise. Besides, we record crucial data fields for further code analysis. For example, we find a data field called "encrypted_username" from network traffic with LastPass, and then we search for this string in the source code to find where and how the algorithm encrypts usernames.

Furthermore, we install the desktop version of PMs on different platforms, such as Windows, Linux, and MacOS, and search for local storage files. We record the plaintext in data files, and we also record crucial data fields for further code analysis. Specifically, Zoho Vault provides an offline version, so we collect the offline version for further research.

Application sources. For application sources, we conduct static and dynamic analyses of applications. We gather code from open source code, extension code, and web page code on the client side. We use the information collected from documentation and data sources to form core terms (e.g., "AES256", "PBKDF2"), and locate the cryptographic algorithms in the source code.

For dynamic analysis, we use browser developer tools to set breakpoints on the location highlighted by the result of static analysis. Subsequently, we execute the *PM operations*, recording the precise algorithms associated with each operation. During dynamic analysis, we find the exact parameter values for cryptographic algorithms, such as iterations for key derivation functions. To address the challenges posed by JavaScript obfuscation and diverse front-end technology stacks, we conduct iterative dynamic analyses and validate the results collaboratively among the authors. Upon confirming all the gathered information, our results encompass M3PM protocols, the cryptographic algorithms employed within these protocols, and the encryption status of metadata.

3.3. Collection results

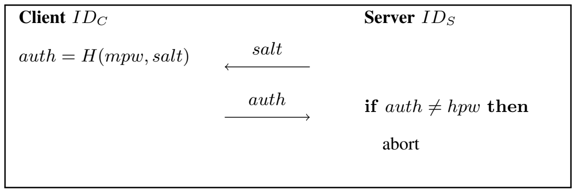
In this section, we answer the **RQ1** and give an overview of our collection results, including M3PM protocols, cryptographic algorithms in protocols, and metadata encryption.

All the results are open [8], and we record as many details as we can to enhance the reproducibility of our work. Overall, (1) 24 of 36 industrial password managers are analyzed using reverse engineering; (2) 32 managers provide useful information in local data files or network data packets; (3) Only six managers are primarily analyzed using documentation. In all, 27 managers collect sufficient information for in-depth analysis. Note that when the data logs provide sufficient information, we do not perform reverse engineering. For instance, during our network capture, we observed that Delinea Web does not encrypt any data, which eliminates the need for further static or dynamic analysis.

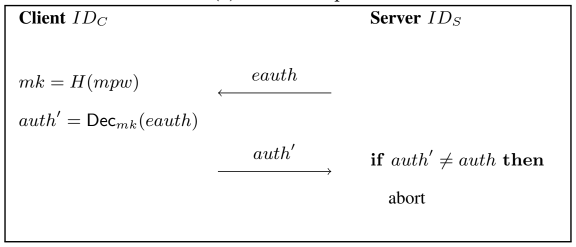
M3PM protocols. Throughout our collection process, we have identified that M3PM protocols can be categorized into two distinct stages: authentication and password processing. The authentication stage primarily facilitates explicit authentication for client-to-server communication. M3PM protocols adopt several approaches for explicit authentication: (1) Sending the hash of master passwords (*Hashed* mpw); (2) Using master passwords to decrypt protected authentication tokens and sending the tokens to authenticate (*Protected tokens*); (3) Using password-authenticated key exchange

1. https://bitwarden.com/help/bitwarden-security-white-paper/

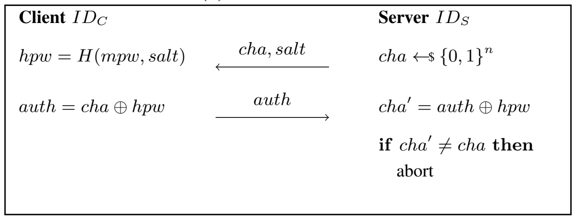
2. Burp Suite is a tool that can capture HTTP/HTTPS traffic on the browser. See more in https://portswigger.net/burp



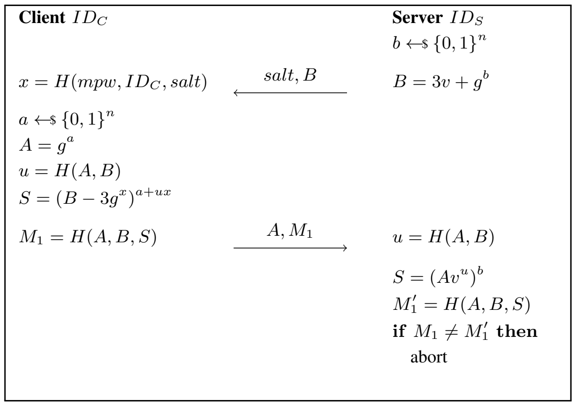
(a) Hashed mpw


Client ID_C
Server ID_S
$mk = H(mpw)$
$eauth$
$auth' = \mathsf{Dec}_{mk}(eauth)$
$auth'$
if $auth' \neq auth$ **then**
abort

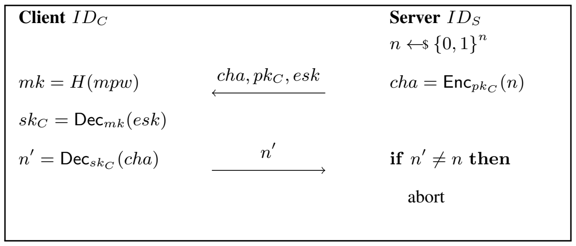
(b) Protected tokens



(c) mpw challenge



(d) SRP


Client ID_C
Server ID_S
$n \leftarrow\$ \{0,1\}^n$
$mk = H(mpw)$
cha, pk_C, esk
$cha = \mathsf{Enc}_{pk_C}(n)$
$sk_C = \mathsf{Dec}_{mk}(esk)$
$n' = \mathsf{Dec}_{sk_C}(cha)$
n'
if $n' \neq n$ **then**
abort

(e) pk challenge

Figure 2: The authentication stage of M3PM protocols. We omit the registration initialization. During registration, the client sends authenticators to the server through a trusted channel. For *Hashed* mpw, the server receives the hash of the master password $hpw = H(mpw, salt)$ and the salt; For *Protected tokens*, the server receives the encrypted and plaintext authenticators $eauth = \mathsf{Enc}_{H(mpw)}(auth)$ and $auth$; For mpw *challenge*, the server receives the hash of the master password $hpw = H(mpw, salt)$ and the salt; For *SRP*, the server receives the authenticator derived from the master password $v = g^{H(mpw, ID_C, salt)}$ and the salt; For the pk challenge, the server receives the public key pk_C and the encrypted private key $esk = \mathsf{Enc}_{H(mpw)}(sk_C)$.

(PAKE) protocols like *SRP*; (4) Using master passwords to answer the challenge from the server (mpw *challenge*); (5) Using public keys to answer the challenge from the server (pk *challenge*). We summarize the authentication phase of the protocol in Fig. 2, omitting the registration initialization to save space.

After successful authentication, the client and server initiate password processing. Methods include encrypting/decrypting passwords with a key derived from the master password (mpw *encrypt*) or with a key generated by devices (*device-key encrypt*), both of which involve explicit authentication. Implicit authentication methods include encoding passwords before encryption (*encode-then-encrypt*) and using metadata and the master password to generate passwords (*generative*). The password-processing phase is depicted in Fig. 3, omitting the vault registration for brevity. Detailed descriptions are provided in Appendix C, and the protocol syntax for each PM is summarized in Table 1.

Cryptographic algorithms. We summarize the cryptographic algorithms in M3PM protocols. These cryptographic algorithms encompass symmetric encryption, asymmetric encryption, hash functions, and key derivation functions. Within M3PM protocols, these cryptographic algorithms are presented as abstract algorithms, accompanied by specific security assumptions (see details in Appendix A). All choices of cryptographic algorithms and detailed information about their parameters are provided in Appendix D.

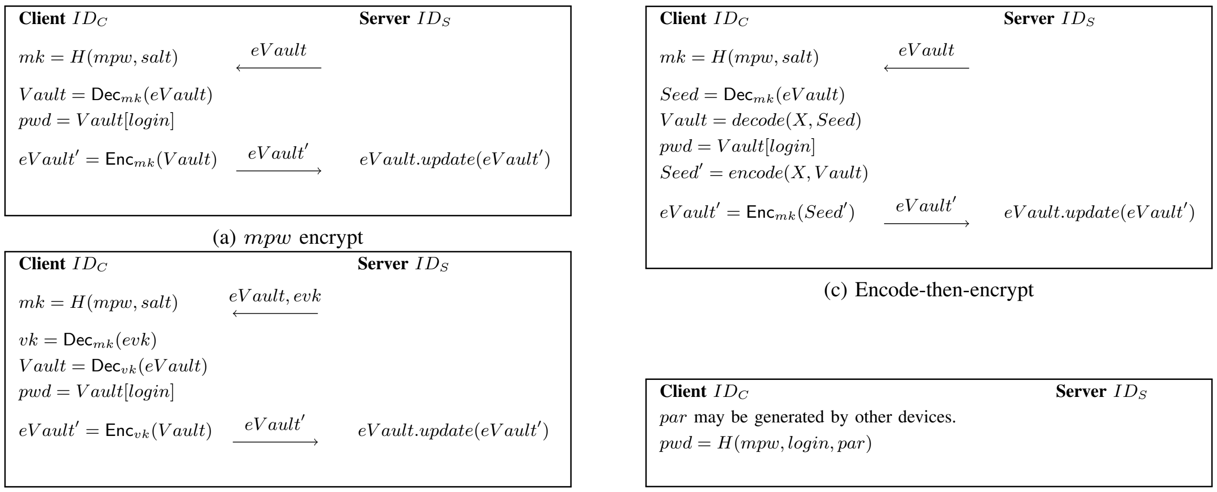
Metadata privacy. The encryption status of metadata not only impacts user privacy but also has implications for distinguishability and malleability [25], [42], which are discussed in Sections 5.2. In our work, we divide the data in password vaults into five different types: secret, login metadata, item state metadata, password hygienist metadata, and users' profiles. All details are shown in Appendix E.

4. Security definitions of M3PM protocols

To address **RQ2**: *What defines a secure M3PM protocol?*, we design a set of security definitions for the M3PM protocol and formalize these security goals within the UC framework [19]. We provide an informal description about our security goals and an overview of our formal security functionalities. These functionalities are illustrated in Fig. 4, with detailed descriptions provided in Appendix B.

4.1. Security goals

Generally, an adversary should not gain any knowledge of the master password mpw or the passwords pwd stored in password vaults. Given recent data breaches affecting PMs, such as the LastPass breach [10], we model the adversary as having the capability to compromise the server and access all stored data. Additionally, many PMs, including Bitwarden, assert that they have "zero knowledge" of users'


Client ID_C
Server ID_S
$mk = H(mpw, salt)$
$eVault$
$Vault = \mathsf{Dec}_{mk}(eVault)$
$pwd = Vault[login]$
$eVault' = \mathsf{Enc}_{mk}(Vault)$
$eVault'$
$eVault.update(eVault')$
Client ID_C
Server ID_S
$mk = H(mpw, salt)$
$eVault, evk$
$vk = \mathsf{Dec}_{mk}(evk)$
$Vault = \mathsf{Dec}_{vk}(eVault)$
$pwd = Vault[login]$
$eVault' = \mathsf{Enc}_{vk}(Vault)$
$eVault'$
$eVault.update(eVault')$
Client ID_C
Server ID_S
$mk = H(mpw, salt)$
$eVault$
$Seed = \mathsf{Dec}_{mk}(eVault)$
$Vault = decode(X, Seed)$
$pwd = Vault[login]$
$Seed' = encode(X, Vault)$
$eVault' = \mathsf{Enc}_{mk}(Seed')$
$eVault'$
$eVault.update(eVault')$
Client ID_C
Server ID_S
par may be generated by other devices.
$pwd = H(mpw, login, par)$

(a) mpw encrypt

(b) Device-key encrypt

(c) Encode-then-encrypt

(d) Generative

Figure 3: The password processing stage of M3PM protocols. We omit the vault initialization. During vault initialization, the client initializes the encrypted vault and sends it to the server. For *mpw encrypt*, the server receives the encrypted vault $eVault = \mathsf{Enc}_{H(mpw,salt)}(Vault)$; For *Device-key encrypt*, the server receives the encrypted key $evk = \mathsf{Enc}_{H(mpw,salt)}(vk)$ and the encrypted vault $eVault = \mathsf{Enc}_{vk}(Vault)$; For *Encode-then-encrypt*, the server receives the encoded-then-encrypted vault $eVault = \mathsf{Enc}_{H(mpw,salt)}(encode(X, Vault))$; For *Generative*, the server receive parameters par.

password. Therefore, we assume that a compromised server should not learn any information about the passwords pwd.

Furthermore, we summarize below the expected security properties we have formalized.

- **Indistinguishability of passwords** [25], [42]: The password initialized by an honest client remains unknown to the malicious server. In other words, any different passwords initialized by the client are indistinguishable to the malicious server.
- **Non-malleability of passwords** [25], [42]: The malicious server cannot tamper with the initialized password without the client's explicit notice.
- **Oblivious of master password** [15], [21], [34], [49], [64]: The initialization and retrieval phases do not leak any information about the master password to the malicious server, ensuring its confidentiality.
- **Pseudorandom generated passwords** [42]: The password generated in the initialization phase is pseudorandom and completely independent of any other previously initialized passwords, even if the server acts maliciously.
- **Resistance to password guessing attack** [15], [21], [49]: The adversary cannot conduct guessing attack on master password. This security properties is relatively strong to M3PM protocols, as users rely solely on the master password to protect their vaults (discussed further in Section 5.1). Therefore, in the functionality $\mathcal{F}_{rM3PM}$, we include a TESTMPW query to model the malicious server's ability to conduct guessing attacks on the master password. This functionality is practical because there are several effective methods beyond protocol design to mitigate guessing attacks (discussed in Appendix 6.1). Notably, we use a single query to capture both online and offline guessing attacks, as a M3PM protocol with $\mathcal{F}_{rM3PM}$ cannot resist both types of attacks.
- **Resistance to the leakage of partial passwords** [49]: In our functionalities, the security ensures that the leakage of partial passwords does not compromise the security of other passwords in the vault. Our functionalities capture this security property by modeling the initialization and retrieval process for one credential at a time.
- **Authentication** [23]: When retrieving passwords, the client must authenticate using the same master password that was used in initialization phase.

Remark 1. Previous work [25], [42] highlights the importance of indistinguishability and non-malleability of metadata. In our functionalities, we extend protection to metadata by modifying pwd to a tuple $(pwd, metadata)$. Our review shows that 30 of 43 PMs do not encrypt all metadata (details in Appendix E), which could limit our analysis of their innovative M3PM protocols. For instance, stopping our security analysis due to unencrypted metadata in 1Password would prevent us from recognizing the security of their dual-key mechanisms (see Fig. 5). We consider unencrypted metadata a straightforward issue to mitigate with encryption, so we present the functionalities without metadata protection.

4.2. Overview of functionalities

To cover the various M3PM protocols, we develop a set of functionalities and iteratively revise them during our analysis of M3PM protocols. Firstly, we present $\mathcal{F}_{M3PM}$ to cover the basic security properties of initializing and retrieving a credential. We then identify two types of authentication: explicit and implicit, each requiring different functionalities to capture their flow. For explicit authentication, when the

Functionality $\mathcal{F}_{M3PM}$

```
Initialization Phase
On input $(\textsc{InitC}, sid, ID_C, ID_S, mpw, login, pwd)$ from $ID_C$ (or $\mathcal{A}$ if $ID_C$ is corrupted):
  IC.1  If $pwd = \epsilon$ then: $pwd \leftarrow\$ \{0,1\}^n$, and mark it GENERATED //Generate a random password for users.
  IC.2  Record $\langle \textsc{InitC}, sid, ID_C, ID_S, mpw, login, pwd \rangle$, overwriting existing record $\langle \textsc{InitC}, sid, ID_C, ID_S, *, *, * \rangle$
  IC.3  Send $(\textsc{InitC}, sid, ID_C, ID_S)$ to $\mathcal{A}$ and $ID_S$
On input $(\textsc{InitS}, sid, ID_C, ID_S)$ from $ID_S$ (or $\mathcal{A}$ if $ID_S$ is corrupted):
  IS.1  Retrieve record $\langle \textsc{InitC}, sid, ID_C, ID_S, [mpw], [login], [pwd] \rangle$ //The client has already started initialization.
  IS.2  Record $\langle \textsc{Vault}, ID_C, ID_S, mpw, login, pwd \rangle$ and send $(\textsc{InitS}, sid, ID_C, ID_S)$ to $\mathcal{A}$
On input $(\textsc{CompleteInitC}, sid, ID_C, ID_S)$ from $\mathcal{A}$: //The client outputs a password and completes the protocol.
  CIC.1  Retrieve record $\langle \textsc{InitC}, sid, ID_C, ID_S, mpw, *, [pwd] \rangle$ and delete it
  CIC.2  If there is a record $\langle \textsc{Vault}, ID_C, ID_S, mpw', *, * \rangle$ marked STORED, then:
           a) If $mpw = mpw'$, send success to $ID_C$ and $\mathcal{A}$
           b) If $mpw \neq mpw'$, send failure to $\mathcal{A}$ and $ID_C$. Delete record $\langle \textsc{InitC}, sid, ID_C, ID_S, mpw, *, [pwd] \rangle$
On input $(\textsc{CompleteInitS}, sid, ID_C, ID_S)$ from $\mathcal{A}$: //The server stores a record and completes the protocol.
  CIS.1  Retrieve record $\langle \textsc{Vault}, ID_C, ID_S, [mpw], *, * \rangle$ not marked STORED
  CIS.2  If there is a record $\langle \textsc{Vault}, ID_C, ID_S, mpw', *, * \rangle$ marked STORED, then:
           a) If $mpw = mpw'$, mark $\langle \textsc{Vault}, ID_C, ID_S, mpw, login, pwd \rangle$ STORED and send success to $ID_S$ and $\mathcal{A}$
           b) If $mpw \neq mpw'$, send failure to $\mathcal{A}$ and $ID_S$, and delete $\langle \textsc{Vault}, ID_C, ID_S, mpw, *, * \rangle$
Retrieval Phase
On input $(\textsc{RetrC}, sid, ID_C, ID_S, mpw', login)$ from $ID_C$ (or $\mathcal{A}$ if $ID_C$ is corrupted):
  RC.1  Record $\langle \textsc{RetrC}, sid, ID_C, ID_S, mpw', login \rangle$, overwriting any record $\langle \textsc{RetrC}, sid, ID_C, ID_S, *, login \rangle$
  RC.2  Send $(\textsc{RetrC}, sid, ID_C, ID_S)$ to $\mathcal{A}$ and $ID_S$
On input $(\textsc{RetrS}, sid, ID_C, ID_S)$ from $ID_S$ (or $\mathcal{A}$ if $ID_S$ is corrupted):
  RS.1  Retrieve record $\langle \textsc{RetrC}, sid, ID_C, ID_S, [mpw'], [login] \rangle$
  RS.2  If no record $\langle \textsc{Vault}, ID_C, ID_S, [mpw], *, * \rangle$ is marked STORED, send $(\textsc{RetrResult}, sid, \textsc{Fail})$ to $ID_S$
  RS.3  Append $mpw$ to $\langle \textsc{RetrC}, sid, ID_C, ID_S, mpw', login \rangle$, overwriting any $\langle \textsc{RetrC}, sid, ID_C, ID_S, *, *, * \rangle$
  RS.4  Send $(\textsc{RetrS}, sid, ID_C, ID_S, mpw \overset{?}{=} mpw')$ to $\mathcal{A}$
On input $(\textsc{CompleteRetrC}, sid, ID_C, ID_S)$ from $\mathcal{A}$:
  CRC.1  Retrieve $\langle \textsc{RetrC}, sid, ID_C, ID_S, [mpw'], [login], [mpw] \rangle$. Mark it RETRIEVED. If it is RETRIEVED, delete it.
  CRC.2  If $mpw = mpw'$ then retrieve the record $\langle \textsc{Vault}, ID_C, ID_S, mpw', login, [pwd] \rangle$ and set $pwd' \leftarrow pwd$
         and send $(\textsc{RetrResult}, sid, \textsc{Success})$ to $\mathcal{A}$ if $ID_C$ is not corrupted
  CRC.3  If $mpw \neq mpw'$, determine the output with two options:
           a) If $pwd$ is marked GENERATED then set $pwd' \leftarrow\$ \{0,1\}^n$. Otherwise, set $pwd' \leftarrow\$ Zipf(n)$
           b) Set $pwd' \leftarrow \textsc{Fail}$, and send $(\textsc{RetrResult}, sid, \textsc{Fail})$ to $\mathcal{A}$ if $ID_C$ is not corrupted
  CRC.4  Send $(\textsc{RetrResult}, sid, pwd')$ to $ID_C$
On input $(\textsc{CompleteRetrS}, sid, ID_C, ID_S)$ from $\mathcal{A}$:
  CRS.1  Retrieve $\langle \textsc{RetrC}, sid, ID_C, ID_S, [mpw'], [login], [mpw] \rangle$. Mark it RETRIEVED. If it is RETRIEVED, delete it.
  CRS.2  If $mpw = mpw'$, send $(\textsc{RetrResult}, sid, ID_C, ID_S, \textsc{Succ})$ to $ID_S$ (or $\mathcal{A}$ if $ID_S$ is corrupted)
  CRS.3  If $mpw \neq mpw'$, send $(\textsc{RetrResult}, sid, ID_C, ID_S, \textsc{Fail})$ to $ID_S$ (or $\mathcal{A}$ if $ID_S$ is corrupted)
On input $(\textsc{TestMpw}, ID_C, ID_S, mpw)$ from $\mathcal{A}$ if server $ID_S$ is corrupted:
  TM.1  If there exists a $\langle \textsc{Vault}, ID_C, ID_S, mpw, *, * \rangle$, then send CORRECT to $\mathcal{A}$. Otherwise, send WRONG to $\mathcal{A}$
```

Figure 4: Ideal functionality $\mathcal{F}_{M3PM}$ for master-password-protected password management protocols. The boxed code describes the explicit authentication $\mathcal{F}_{M3PM-EA}$, and we can drop it to have the implicit authentication $\mathcal{F}_{M3PM-IA}$. The dashed box is for relaxed functionality $\mathcal{F}_{rM3PM}$. The $\mathcal{F}_{adM3PM}$ needs change all mpw into a tuple $(mpw, SecretK)$. The notation "*" means any value, and "[]" means retrieving a value.

client fails to authenticate, the server responds with a FAIL signal. For implicit authentication, the server responds with a random and independent incorrect password to the client upon authentication failure. Therefore, we extend $\mathcal{F}_{M3PM}$ to $\mathcal{F}_{M3PM-EA}$ and $\mathcal{F}_{M3PM-IA}$.

Our analysis shows that 34 of the 43 M3PM protocols (see Table 1) cannot fulfill $\mathcal{F}_{M3PM-EA}$ or $\mathcal{F}_{M3PM-IA}$ because they fail to resist offline guessing attacks when the server is compromised. However, Table 2 shows that 35 PMs use PBKDF2 or Argon2 to mitigate offline guessing attacks. These functions can reduce the successful rate of offline guessing master passwords. Therefore, we introduce a relaxed functionality $\mathcal{F}_{rM3PM}$ (containing $\mathcal{F}_{rM3PM-EA}$ and $\mathcal{F}_{rM3PM-IA}$) in Fig. 4 which gives credit to these functions, and we will discuss more technologies on mitigating offline guessing attacks in Sec. 6.1.

Finally, we find 1Password and Multipassword adopt additional secret value and achieve the strong security notion against all kind of offline guessing attack in Table 1. Therefore, we introduce another functionality $\mathcal{F}_{adM3PM}$ which captures additional secret value enhanced M3PM protocols.

Remark 2. In our functionalities, we only consider two parties: a client and a server. In fact, there are also M3PM protocols containing one or three parties. For one party, we designate the local PM as the entity through which users store and retrieve password vaults on a specific device without involving any communication through the network. Strictly speaking, this scenario may not qualify as a M3PM protocol, but we can analyze this scenario by assigning with a server to facilitate synchronization. For the three-parties protocol, there is another assistant device. In our functionality, we only consider two participants for two reasons: Two-party protocols are beneficial for UC framework [19]; Two-party protocols can be extended to three within the UC framework, and one-party protocols can be modeled as a secure functionality. We leave the work of modeling additional devices as a future work.

5. Security analysis

To answer the **RQ3:** *Are existing password managers secure under our security definitions?* we give a security analysis of all M3PM protocols. The results are summarized in Table 1.

In our analysis, we show that only 1Password and Multipassword achieve $\mathcal{F}_{adM3PM-EA}$ which is owed to the entropy of the secret key (they adopt the same protocol, and the security of 1Password is shown in Sec. 5.3). Other protocols at most achieve $\mathcal{F}_{rM3PM}$ because they all suffer offline guessing attacks. In total, there are eight distinct protocols. We provide the proof for 1Password and demonstrate how to extend the proof to other *mpw encrypt*, *device-key encrypt*, and *encode-then-encrypt* protocols. For the *generative* scheme, we give the proof of SPHINX [49] in Sec. 5.5.

Furthermore, Delinea Web PM does not adopt any cryptographic method on the client side. Passbolt faces an oracle attack which we describe in Sec. 5.4. Six PMs without a provable security result cannot be proven secure in our model: Norton PM, Chrome PM, and Passwarden do not have enough information for protocol analysis; Amnesia [58], BluePass [34], and Tapas [40] have three parties in protocols.

5.1. Offline guessing attack

The underlying vulnerability of offline guessing attacks comes from two facts: low entropy and offline verification. The low entropy of passwords brings unavoidable brute force guessing, and Zipf-distribution compresses the guessing space of passwords [54]. Compared with common encryption algorithms, such as AES, a password commonly has 2^{20-22} bit entropy [16], while AES has three options of key length, i.e., 128 bits, 192 bits, and 256 bits. Consequently, successfully guessing a password carries a non-negligible probability compared to guessing an ordinary random key.

Another reason causing offline guessing attacks is that adversaries find a way to verify the success of password guessing offline. For example, the adversary may exploit the database of the server and then obtain all password authenticators $hpw = H(mpw, salt)$. Then the adversary can guess passwords mpw' and compare the hash results $hpw' = H(mpw', salt)$ with the authenticators to verify the correctness of the password guesses.

Formally, we formalize this verification process as an algorithm $GuessVerify$. The $GuessVerify$ takes three inputs: a string m which is the message for verify, a distribution X which gives the probabilities of the m to be a correct guess, and a threshold σ which represents the confidence for the adversary about the guess. If the probability of m in X is larger than σ then the algorithm returns $\delta = 1$ representing that the verification is correct. For example, the adversary knows passwords follow Zipf-distribution [54], then she sets the distribution X to be Zipf-distribution, and she believes that the probability more than 0.1 represents a password, then she sets the threshold σ to be 0.1. Therefore, when the adversary decrypts the vault, a value `123456abc` has a high probability of being a password in Zipf-distribution, while `Q!='GhEuk76!WH+92Rf` has a low probability of being a password even it is generated by a password generator. We use distribution X to give a general description for the process of verification, and for a specific situation, such as the hashed password, the distribution can be $\Pr[hpw = H(mpw', salt)] = 1$ and $\Pr[hpw \neq H(mpw', salt)] = 0$ (here we omit the collision of the hash functions).

We give our analysis based on our taxonomy in Sec. C. For *hashed* mpw, the corrupted server ID_S has $hpw = H(mpw, salt)$ and the $salt$, then ID_S can guess a master password mpw' and calculates $hpw' = H(mpw', salt)$. The ID_S can validate the correctness of guesses by checking $hpw \overset{?}{=} hpw'$. For *protected tokens*, the ID_S has $eauth = \mathsf{Enc}_{H(mpw)}(auth)$ and $auth$, then ID_S can guess the master password mpw' and decrypt the $eauth$

TABLE 1: Overview of security in password managers. †

Password manager@Version	Protocol Syntax		Resist offline guessing*				Metadata privacy	Enough details	Provable Security
	Authentication	Password processing	I	II	III	IV			
Bitwarden@2023.9.1	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
1Password@8.10.16	SRP+two-key‡	device-key encrypt	●	●	●	●	○	●	$\mathcal{F}_{adM3PM-EA}$
LastPass@4.123.0	hashed mpw	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Dashlane@6.2342.0	protected tokens	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
RoboForm@9.5.2	hashed mpw	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Keeper@16.10.9	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
NordPass@5.8.21	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Zoho Vault@3.8	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
KeePass@2.55	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
KeePassXC@2.7.6	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
Dropbox PM@3.26.0	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
DualSafe PM@1.4.21	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
Enpass@6.9.1	hashed mpw	mpw encrypt	●	○	●	○	○	○	$\mathcal{F}_{rM3PM-EA}$
Norton PM@8.1.0.73	mpw challenge	mpw encrypt	●	○	●	○	●	○	-
Avira PM@2.19.13.44521	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
McAfee True Key@4.3.1.9339	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Sticky PM@8.8.3.1629	protected tokens	mpw encrypt	●	○	●	○	○	○	$\mathcal{F}_{rM3PM-EA}$
Password Boss@5.5.5138.0	hashed mpw	mpw encrypt	●	○	●	○	○	○	$\mathcal{F}_{rM3PM-EA}$
Multipassword@0.93.36	SRP+two-key‡	device-key encrypt	●	●	●	●	○	●	$\mathcal{F}_{adM3PM-EA}$
Passwarden@1.3.3	protected tokens	mpw encrypt	●	○	●	○	○	○	-
Delinea Web PM@3.7	-	-	●	○	●	○	○	●	✗
LogMeOnce@7.8.0	hashed mpw	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Passbolt@4.4.2	pk challenge	mpw encrypt	●	○	●	○	○	●	✗
IronVest@9.8.15	hashed mpw	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
SafeInCloud PM@22.3.4	hashed mpw	mpw encrypt	●	○	●	○	○	○	$\mathcal{F}_{rM3PM-EA}$
Password Safe@3.64.1	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
Bitdefender PM@3.21.1247	hashed mpw	mpw encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
iCloud Keychain@Ventura 13.5.2	hashed mpw	device-key encrypt	●	○	●	○	●	○	$\mathcal{F}_{rM3PM-EA}$
Chrome PM@119.0.6045.160	-	device-key encrypt	●	○	●	○	○	●	-
KeeWeb@1.18.7	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
Lesspass@9.1.9	-	generative	●	●	○	○	●	●	$\mathcal{F}_{rM3PM-IA}$
Gopass@1.15.10	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Padloc@4.3.0	SRP	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Authpass@1.9.9	hashed mpw	mpw encrypt	●	○	●	○	●	●	$\mathcal{F}_{rM3PM-EA}$
Gokey@0.1.2	-	generative	●	●	○	○	●	●	$\mathcal{F}_{rM3PM-IA}$
TeamPass@3.0.10	hashed mpw	device-key encrypt	●	○	●	○	○	○	$\mathcal{F}_{rM3PM-EA}$
SPHINX [49]	-	generative	●	●	●	○	●	●	$\mathcal{F}_{rM3PM-IA}$
BluePass [34]	-	device-key encrypt	●	●	●	●	○	●	-
NoCrack [21]	-	encode-then-encrypt	●	●	●	○	○	●	$\mathcal{F}_{rM3PM-IA}$
Amnesia [58]	-	generative	●	●	●	●	●	●	-
Kamouflage [15]	-	encode-then-encrypt	●	●	●	○	○	●	-
CSF-BPM [64]	hashed mpw	device-key encrypt	●	○	●	○	○	●	$\mathcal{F}_{rM3PM-EA}$
Tapas [40]	-	device-key encrypt	●	●	●	●	●	●	-

† ● : The PM fulfills this security property; ○ : The PM does not fulfill this security property. The ✗ in a column of provable security means this protocol has a security flaw. 1Password and Multipassword achieve $\mathcal{F}_{adM3PM-EA}$. The "-" symbol in columns of protocol syntax means the PM does not use any cryptographic method, and the "-" symbol in a column of provable security means this protocol cannot be proven in the current model. It worth note that Enpass, Sticky PM, Password Boss, SafeInCloud, iCloud Keychain, and TeamPass do not have enough information details for determining the correctness of the documentation, so we analyze their M3PM protocols from documentation descriptions. We do not have enough information about Norton PM and Passwarden for protocol analysis. Chrome PM, BluePass, Amnesia, Kamouflage, and Tapas are beyond our current model.
* We divide offline guessing attacks into four types: Type-I refers to the adversary attacking the protocol without additional leaked information; Type-II refers to the adversary attacking the protocol with a corrupted server; Type-III refers to the adversary knowing partial leaked passwords in the password vault; Type-IV refers to the adversary both corrupts the server and knows partial leaked passwords.
‡ SRP+two-key means the user uses two secrets: a master password and a random string to authenticate and retrieve password vaults.

by $auth' = \mathsf{Dec}_{H(mpw')}(eauth)$, then ID_S can verify the guess by checking $auth \overset{?}{=} auth'$. For *SRP*, the corrupted server ID_S has $v = g^{H(mpw, ID_C, salt)}$ and $salt$, then ID_S can guess the master password mpw', calculate $v' = H(mpw', ID_C, salt)$, and verify $v \overset{?}{=} v'$. For mpw *challenge*, the server has the plaintext of the master password, and for pk *challenge* the server ID_S guesses the master password and decrypts the private key $sk'_C = \mathsf{Dec}_{H(mpw')}(esk)$, and uses the public key pk_C to validate the correctness of the guess.

In the password-processing stage, the situation is more complicated. For mpw *encrypt*, the server ID_S has the encrypted password vault $eVault$. Then ID_S guesses a master password mpw' and calculates the key $mk' = H(mpw, salt)$. The ID_S uses the key mk' to decrypt the encrypted vault and verify the output. If the encryption algorithm is an authentication encryption, such as AES-GCM, then the wrong key leads to a failed decryption which eliminates a wrong guess. If the wrong key leads to a wrong plaintext instead of a failed decryption, the adversary can use the $GuessVerify$ algorithm to determine the correctness of the plaintext. For example, if the decrypted result gets jumbled e.g., containing non-ASCII characters and more than 50 characters, it is a wrong decryption with high probability. The overall offline guessing attack is formally described in Algorithm 1.

Therefore, the mpw *encrypt* and *device-key encrypt* retrieval mechanism cannot resist an offline guessing attack.

Algorithm 1: Offline guessing attack on password vaults.

```
Input: The encrypted password vault $eVault$, the expected random variable for master password $X_{mpw}$, the expected random variable for password $X_{pwd}$, and the guess threshold $\sigma$.
Output: The passwords vault $Vault$.
1  $Guess\_mpw\_set = X_{mpw}.sortPr(order = descend)$; //Use $X_{mpw}$ to generate a guess dictionary.
2  $\mathcal{K} = \varepsilon$; //Initialize a variable.
3  $Vault = \phi$;
4  while $!Guess\_mpw\_set.empty()$ do
5      $Guess\_mpw = Guess\_mpw\_set.pop()$;
6      $Guess\_mk = KDF(Guess\_mpw)$; //Derive the key by following the specific protocol.
7      $Guess\_vault = \mathsf{Dec}(Guess_m k, eVault)$;
8      $\delta = GuessVerify(Guess\_vault, X_{pwd}, \sigma)$
9      if $\delta = 1$ then
10         $pwd' = Guess\_vault[login]$; //Select one credential.
11         if $OnlineTest(login, pwd')$ then
12             $\mathcal{K} = Guess\_mpw$; //If the credential logs in successfully, the vault is decrypted.
13             $Vault = Guess\_vault$;
14             return $Vault$;
15 return $Vault$
```

What's worse, the *encode-then-encrypt* mechanism which is designed to resist offline guessing attacks [15], [21] cannot restrict the attack into negligible probability. Briefly reviewing the mechanism of *encode-then-encrypt*, the encoding algorithm relies on a distribution X to the decryption of a wrong key following the distribution X. Then, if the adversary inputs the same distribution X into the $GuessVerify$ algorithm, a wrong decrypted password will have a high probability of passing the $GuessVerify$ algorithm. Therefore, the adversary needs to perform $OnlineTest$, then the offline guessing attack is transformed into a less threatening online guessing attack.

However, the distribution in the encoding algorithm cannot perfectly deny the $GuessVerify$ algorithm, for the adversary can adjust its verification distribution adaptively. For example, if one credential is decrypted into `abc12345` which has a high probability in Zipf-distribution, but the corresponding website forces all passwords having at least one special character, then `abc12345` has zero probability of becoming a correct password. This defect seemingly can be mitigated by adopting a more accurate distribution for each credential. However, the mitigation is endless, for the adversary may know one correct password in the vault and then the adversary verifies the guess with a high probability.

Generally, the generative M3PM can resist offline guessing, for the adversary cannot verify the guess offline without any value for verification and has to resort to online verification. Because all passwords generated by generative M3PM follow uniform distribution, the adversary cannot gain an advantage by gaining knowledge on distribution. However, if one generated password is leaked, the adversary gains the advantage of offline guessing and verifying the guess.

In Table 1, we summarize the result of our security analysis on offline guessing. We categorize offline guessing into four types depending on the ability of the adversary. Type-I adversaries can interact with the protocol but have no additional information about users. All PMs can resist this kind of adversary because they adopt TLS to protect communication channels. Type-II adversary can corrupt the server and learn the internal state of the corrupted server. 1Password and Multipassword adopt two secrets, i.e., master password and random secret key, to resist a guessing attack. Only if the adversary gets the secret key can she conduct a guessing attack, which is a strong capability for an adversary to steal users' secret keys. Lesspass, Gokey, SPHINX, and Amnesia do not store any value about password vaults. NoCrack and Kamouflage use the *encode-then-encrypt* method to resist offline guessing upon leakage. BluePass, CSF-BPM, and Tapas use another device, and only a corrupted server cannot give the adversary the ability to guess the password.

In type III, the adversary learns partially leaked passwords in password vaults, and LessPass and Gokey suffer from this kind of adversary. They do not rely on any storage with additional secrets, so the adversary can guess the master password and generate the password for verification. If the adversary can learn partially leaked passwords and corrupt the server, i.e., type-IV adversary, only another secret key (1Password and Multipassword) or additional device (BluePass, CSF-BPM, and Tapas) can resist this offline guessing attack.

Insights. Our results show that all PMs capable of resisting offline guessing attacks introduce another high-entropy secret. For instance, 1Password and Multipassword require users to keep an additional secret string, while BluePass [34], Amnesia [58], and Tapas [40] introduce another device to store secret keys. This approach makes sense because if a password vault is protected by a single master password, the master password alone cannot provide sufficient entropy for protection, and some verifier, such as password vaults or generated passwords, can be obtained by the adversary. Therefore, without an additional high-entropy secret, M3PM protocols heavily rely on technologies such as slow hash.

5.2. Metadata privacy

In the works of Gasti et al. [25] and Oesch and Ruoti [42], the assessment of metadata privacy focuses on the encryption of metadata and the encryption algorithms used. Overall, 30 of the 43 PMs have unencrypted metadata (see Tables 1 and 3), which violates the principles of indistinguishability and non-malleability of metadata.

The malleability of metadata has more severe negative effects than that of secret passwords. While password malleability primarily results in usability issues, such as users being unable to log in with tampered passwords and potentially needing to reset them, tampered login metadata poses a significant security risk. An adversary could change a website's URL to a phishing URL. If users unknowingly access phishing websites, PMs might autofill passwords into these sites. Additionally, PMs often provide click-to-jump features for domains, allowing users to navigate to the domain with a click. We verified that the offline version of Zoho Vault has a vulnerability where an adversary

Protocol OP

Input: In the initialization phase, the client inputs a master password $mpw \in Zipf(n)$, a secret key $SecretK \in \{0,1\}^n$, login metadata $login$, and password pwd. In the retrieval phase, the client inputs a master password mpw', a secret key $SecretK'$, and login metadata $login$. The server inputs nothing; the common input is the security parameter n.
Output: The client outputs a password pwd in the initialization and retrieval phases. The server outputs a success/failure.
Protocol:

I **Initialization Phase.**

(a) [Registration] The client ID_C utilizes master password mpw and secret key $SecretK$ to generate a master authenticator $x = H_1(mpw, ID_C) \oplus H_1(SecretK, ID_C)$ and a master key $mk = H_2(mpw, ID_C) \oplus H_2(SecretK, ID_C)$. Subsequently, the client generates a vault key vk and private key sk, and initializes an empty vault $Vault$. The client employs the mk to encrypt the private key $esk = AE.\mathsf{Enc}_{mk}(sk)$, utilizes the private key sk to encrypt the vault key $evk = PKE.\mathsf{Enc}_{sk}(vk)$, and finally, uses the vault key vk to encrypt the vault $eVault = AE.\mathsf{Enc}_{vk}(Vault)$. If there is no record $(\text{FILE}, ID_C, ID_S, x)$ in ID_S, ID_C querys $(\text{STOREPWDFILE}, sid, ID_C, x)$ to the functionality $\mathcal{F}_{pwAuth}$, and transmits $(evk, esk, eVault)$ to the server.

(b) [Authentication] The client sends $(\text{NEWSESSION}, sid, ID_C, ID_S, x, \text{Client})$ to $\mathcal{F}_{pwAuth}$, and the server sends $(\text{NEWSESSION}, sid, ID_S, ID_C, \text{Server})$.

(c) [Vault initialization] Upon receive SUCCESS from $\mathcal{F}_{pwAuth}$, the server sends $(\text{INIT}, evk, esk, eVault)$ to the client. The client uses the mk to decrypt the key and vault. Subsequently, the client initializes the password pwd with the login metadata $login$. Finally, the client outputs the pwd, re-encrypts the vault $eVault, evk, esk$, and sends $(eVault, evk, esk)$ to the server.

II **Retrieval Phase.**

(a) [Authentication] The client and server use the same authentication method in the initialization phase.

(b) [Vault retrieval] Upon receive SUCCESS from $\mathcal{F}_{pwAuth}$, the server sends $(\text{RETR}, evk, esk, eVault)$ to the client. Then, the client uses the master key mk to decrypt the key and vault. Finally, the client uses login metadata $login$ to retrieve the password pwd and outputs it.

Figure 5: The complete M3PM protocol of 1Password. The client and server execute the SRP protocol with master authenticator x. The details of SRP are in Fig. 2, and we use functionality $\mathcal{F}_{pwAuth}$ in this protocol.

can tamper with the URL, and users may click the jump button, redirecting them to a phishing URL. This feature makes phishing attacks even more dangerous. Furthermore, tampering with user settings can be fatal. An adversary could change settings to an insecure state, such as reducing the iterations of the key derivation function (KDF) or disabling the requirement for a master password.

It is worth noting that the *encode-then-encrypt* password-processing scheme struggles to achieve indistinguishability and non-malleability of metadata. Indeed, if the client employs a key derived from the master password to encrypt the metadata, *encode-then-encrypt* would also need to operate on the metadata. The distribution of the metadata needs to be encoded, which introduces an additional layer of complexity and considerations for *encode-then-encrypt* in scenarios. Moreover, neither metadata nor secret passwords can achieve non-malleability because the honey encryption mechanism never indicates a failure, even with incorrect decryption.

5.3. Case study 1: 1Password

We show the details of the M3PM protocol of 1Password in Fig. 5. 1Password adopts *SRP* [60] for authentication, and *device-key encrypt* for password-processing. According to the above analysis, *SRP* and *device-key encrypt* cannot resist offline guessing attacks under corruption. However, 1Password adopts a two-key mechanism in that users remember a master password mpw and save a 44-character random string called secret key $SecretK$. Both authenticator $x = H_1(mpw, ID_C) \oplus H_1(SecretK, ID_C)$ and encryption key $mk = H_2(mpw, ID_C) \oplus H_2(SecretK, ID_C)$ are derived by the combination of the master password and secret key so that the adversary cannot conduct offline guessing attack without the knowledge of the secret key.

Before our proof, we establish some necessary assumptions. First, we assume that the hash functions in the protocol are modeled as random oracles, the public encryption algorithm satisfies IND-CPA, and symmetric encryption uses an AEAD algorithm. Additionally, while the SRP protocol lacks a formal security proof, no security flaws have been identified to date [62]. Our work does not aim to address the challenge of proving SRP's security. Therefore, we use a functionality $\mathcal{F}_{pwAuth}$ based on the work of Canetti et al. [20] and Groce and Katz [27] to replace the SRP protocol. Details of our preliminary assumptions are provided in Appendix A.

Theorem 1. *Let the client and server use an authentication sub-protocol UC-realizing $\mathcal{F}_{pwAuth}$. Let H_1, H_2 be random oracles, $PKE = (PKE.KeyGen, PKE.\mathsf{Enc}, PKE.\mathsf{Dec})$ be an IND-CPA-secure public key encryption scheme, and $AE = (AE, KeyGen, AE.\mathsf{Enc}, AE.\mathsf{Dec})$ be an authenticated encryption scheme. Let OP be the M3PM protocol of 1Password in Fig. 5. Then the OP UC-realizes* *the M3PM functionality $\mathcal{F}_{M3PM-EA}$ of Fig. 4 in $\mathcal{F}_{pwAuth}$-hybrid model.*

PROOF. The proof can be found in Appendix F. Here we provide a brief sketch of the main idea in the proof.

Our analysis is based on sequence-of-games proof strategy [50]. We define a series of games from the real-world execution to ideal-world execution and prove that the environment $\mathcal{Z}$ can distinguish each two games in a negligible probability, which means the environment cannot distinguish the real world and the ideal world. The simulator for the M3PM protocol of 1Password is depicted in Fig. 9.

The challenge of constructing a simulator is that, without knowledge of the correct master password, the simulator cannot determine whether the master password sent by the adversary is correct. To address this, the simulator records every initialization query sent by the adversary and rejects any retrieval query from the adversary that lacks a corresponding initialization. This solution is effective because it is negligible for the adversary to guess the correct authenticator.

The simulator records every initialization by an honest client and stores a “garbage” vault and key, which are random values. This storage maintains consistency, ensuring that the adversary can verify the consistency of an honest client’s storage without gaining any knowledge or tampering with it. For every encrypted vault and key sent by the adversary, the simulator checks whether the vault has been previously stored (leveraging the INT-CTXT security reduction for authenticated encryption). If a stored vault exists, the simulator encrypts a “garbage” vault and sends it to the adversary (leveraging the IND-CPA security reduction for authenticated encryption).

Remark 3. We briefly mention the proof for other schemes. Without an additional high-entropy secret key, the adversary has a non-negligible chance of correctly guessing the master password. Therefore, the simulator needs to program the random oracle and record the queries for any master passwords. Then, the simulator uses the TESTMPW query to test the correctness of the master password. By doing so, the simulator can respond correctly to the adversary.

5.4. Case study 2: Passbolt

The details of Passbolt’s M3PM protocol are depicted in Fig. 6. Passbolt uses *pk challenge* for authentication, and *device-key encrypt* for retrieval. Specifically, Passbolt employs the OpenPGP standard [32] for encryption, introducing a public key pair. The client stores an encrypted secret key esk, and both the client and server have the public key pk_C of the client ID_C. The server uses the public key pk_C and samples a random nonce $b \leftarrow\$ \{0,1\}^n$ to challenge the client for authentication $cha = PKE.\mathsf{Enc}_{pk_C}(b)$. Then, the client uses the master password mpw to decrypt the encrypted secret key $sk_C = AE.\mathsf{Dec}_{H(mpw)}(esk)$ and uses the secret key to respond to the challenge. The secret key sk is also used to decrypt the vault key $vk = PKE.\mathsf{Dec}_{sk_C}(evk)$ and the vault key is sampled freshly in each encryption.

The protocol of Passbolt has an oracle attack in that the corrupted server ID_S can learn the vault key vk of the client ID_C. The security flaw comes from the fact that the secret key sk_C is also used to both decrypt the challenge and the encrypted vault key. The corrupted server sends the encrypted vault key evk as challenge $cha' = evk$ instead of sampling a new random nonce. Then the honest client will decrypt the challenge and respond with the vault key to the server $evk' = PKE.\mathsf{Dec}_{sk_C}(cha)$. The client uses sk_C to answer the challenge, which gives the adversary an oracle to use sk_C for decryption. Despite the client updating the vault key in each initialization, the server can gain the vault key in each authentication of the protocol.

5.5. Case study 3: SPHINX

SPHINX is proposed in Shirvanian et al.’s work [49], and it is built upon the device-enhance password-authenticated key exchange (DE-PAKE) protocol [30]. The details of SPHINX are in Fig. 7. The cryptographic primitive during this process is called oblivious pseudorandom function (OPRF) which is used in many cryptographic protocols [23], [30], [31].

In the initialization phase, the client ID_C samples a random value $\rho \leftarrow\$ \mathbb{Z}_q$ and uses it to blind the hash of master password mpw and login domain $login$. Then the client sends the blinded value $\alpha = (\mathsf{H}'(mpw, login))^\rho$ to the server ID_S. The server receives the value α and samples a new random secret $k \leftarrow\$ \{0,1\}^n$ which is stored in the server for further retrieval. Then the server sends $\beta = \alpha^k$ to the client. Upon receiving the β, the client unblind the ρ by $\zeta = \beta^{\frac{1}{\rho}}$ and then generate the password pwd by the coordination of the client and the server $pwd = \mathsf{H}(mpw, login, \zeta)$. The retrieval phase is similar to the initialization phase except that the server does not sample a new secret k but retrieves the stored one.

The security of SPHINX is proved in [30] with the one-more gap DH assumption [12]. In our work, we prove the following theorem in our security definitions.

Theorem 2. *Let $\mathbb{G}$ be a group under one-more gap DH assumption, and let H_1, H_2 be random oracles. Let $SPHINX$ be the M3PM protocol of SPHINX in Fig. 7. Then the $SPHINX$ UC-realizes the $\mathcal{F}_{rM3PM-IA}$.*

PROOF. The proof sketch is provided in Appendix G. We present the simulator and sequence of games in the Appendix.

6. Discussion

6.1. Mitigation on offline guessing attacks

Now we discuss three main methods: slow hash/memory-hard function, password policy, and honey encryption. These

Protocol PB

Input: In the initialization phase, the client inputs a master password $mpw \in Zipf(n)$, login metadata $login$, and password pwd. In the retrieval phase, the client inputs a master password mpw', login metadata $login$.

Output: The client outputs a password pwd in the initialization and retrieval phases. The server outputs a success/failure.

Protocol:

I **Initialization Phase.**

(a) [Registration-Auth] The client ID_C generates a public key pair and sends the public key pk_C to the server. Subsequently, the client utilizes the master password to encrypt the private key $esk = AE.\mathsf{Enc}_{H(mpw)}(sk_C)$.

(b) [Registration-Vault] The client ID_C generates a vault key $vk \leftarrow\!\!\$ \{0,1\}^n$ and encrypts it with private key $evk = PKE.\mathsf{Enc}_{pk_C}(vk)$. Finally, the client uses the vault key to encrypt an empty vault $eVault = AE.\mathsf{Enc}_{vk}(Vault)$, sending $evk, eVault$ to the server.

(c) [Authentication] The client sends $kf = H(pk_C)$ to start the session, and the server checks $kf \overset{?}{=} H(pk_C)$. Then the server generates a random value $b \leftarrow\!\!\$ \{0,1\}^n$ and utilizes the public key to challenge the client $cha = PKE.\mathsf{Enc}_{pk_C}(b)$. If the client can decrypt and answer the b correctly, then authentication is successful.

(d) [Vault initialization] If the server authenticates the client, then the server sends $(\textsc{Init}, eVault, evk)$ to the client. Then, the client uses the master key $mk = H(mpw)$ to decrypt the key and vault. Subsequently, the client can initialize the password pwd with the login metadata $login$. Finally, the client outputs the password pwd, re-encrypts the vault $eVault'$ with new key vk', and sends $eVault', evk'$ to the server.

II **Retrieval Phase.**

(a) [Authentication] The client and server use the same authentication method in the initialization phase.

(b) [Vault retrieval] If the server authenticates the client, then the server sends $(\textsc{Retr}, eVault, evk)$ to the client. Then the client uses the master key $mk = H(mpw)$ to decrypt the key and vault. Finally, the client uses login metadata $login$ to retrieve the password pwd, and outputs it.

Figure 6: The complete M3PM protocol of Passbolt. There is a security flaw: The compromised server has the capability to transmit the encrypted vault key evk as $cha = PKE.\mathsf{Enc}_{pk_C}(vk)$ instead of sampling a new challenge to obtain the vault key $b' = vk$, and the client remains unaware.

Protocol $SPHINX$

Input: In the initialization phase, the client inputs a master password $mpw \in Zipf(n)$, login metadata $login$. In the retrieval phase, the client inputs a master password mpw' and login metadata $login$. The server inputs nothing; the common input is the security parameter n.

Output: The client outputs pwd in the initialization and retrieval phases. The server outputs a success/failure.

Protocol:

(a) [OPRF-Blind] The client samples a random value for blinding $\rho \leftarrow\!\!\$ \{0,1\}^n$, then uses it to blind the query $\alpha = (H_1(mpw, login))^\rho$. The client sends α to the server.

(b) [OPRF-Generate] In the initialization phase, the server generates a random value k and stores it for the retrieval phase. Then the server generates the secret $\beta = \alpha^k$ to the client.

(c) [OPRF-Disclose] The client discloses the secret $zeta = \beta^{\frac{1}{\rho}}$, and finally gets the generated password $pwd = H_2(mpw, login, \zeta)$.

Figure 7: The complete M3PM protocol of SPHINX.

methods cannot reduce the probability of offline guessing attacks to negligible, but they have great benefits in defending offline guessing attacks in practice.

Slow hash and memory hard function. One practical approach to weakening the effectiveness of offline guessing attacks is to slow down the speed of each guess, thereby limiting the number of guesses an adversary can perform within a given period. This can be achieved through the use of slow hash functions and memory-hard functions as key derivation functions (KDFs). Slow hash functions deliberately increase the computational time required for each guess, making it more difficult for adversaries to perform a high number of iterations quickly. Memory-hard functions, on the other hand, require substantial memory resources in addition to computational time, further hindering the adversary's ability to efficiently carry out offline guessing attacks. Both of these methods are employed to effectively slow down step 6 in Algorithm 1, thereby enhancing the security of the password managers against offline guessing attempts.

Blocki et al. [14] analyzed the strategies and capabilities of offline guessing attackers, finding that slow hash functions are ineffective against offline guessing attacks, whereas memory-hard functions are more effective. In our research, we consider these "slowing" strategies as mechanisms that effectively expand the password guess space. This is because the time required for one guess with these strategies might be equivalent to 2^{15} guesses without them. However, it is crucial to note that these strategies do not affect the password distribution. In simpler terms, if users choose a weak master password like `123456`, the effectiveness of slow hash and memory-hard functions is limited.

Password creation policy. Another way to weaken the ability of an offline guessing attack is to reduce the probability of guessing the correct master password, then the adversary needs to conduct more guesses. PMs adopt different password creation policies for master passwords. For example, LastPass requires master passwords to be longer than 12 and have at least one lowercase, one uppercase, and one digit character, while Dashlane does not set explicit requirements. Though there is no research showing which password creation policy benefits most, we believe that adopting a password creation policy and blocking simple passwords, such as `123456` and `password`, can mitigate offline guessing attacks.

Honey encryption. We now discuss honey encryption [21], a specific encryption algorithm used in the *encode-then-encrypt* password-processing method. The encoding algorithm is designed to render the $GuessVerify$ algorithm ineffective, forcing the adversary to rely on online verification, which is less efficient and more likely to be detected by websites. As previously analyzed, while the encoding process cannot completely eliminate the chance of a correct guess by $GuessVerify$, it does increase the adversary's knowledge requirements regarding the user's passwords.

In the M3PM functionality (see Fig. 4), we acknowledge the effectiveness of honey encryption by instructing $\mathcal{F}_{rM3PM-IA}$ to return a random password following a Zipf distribution upon authentication failure. This approach means the adversary would need additional knowledge about the user's personal information to successfully break the honey encryption.

However, honey encryption presents other challenges. In Sec. 5.2, we discuss the necessity of encrypting metadata and the difficulties honey encryption faces in this context. Another challenge lies in integrating honey encryption with explicit authentication. If the server stores an authenticator for the master password, it becomes vulnerable to offline guessing attacks in the event of a server compromise. Therefore, the presence of an authenticator for explicit authentication can undermine the effectiveness of honey encryption.

6.2. Recommendations

According to our analysis in Sec. 5.1, all the authentication methods in Fig. 2 cannot resist offline guessing attacks upon the compromised server. In essence, for authentication with only one master password, the server has to store such an authenticator because this is the only secret between the user and the server. However, the low entropy of the master password makes the authenticator vulnerable to guessing attacks. Therefore, we recommend PMs adopt two-factor authentication. We highlight these two factors must be incorporated into authentication and encryption.

A direct and low-cost method to address the problem of guessing attacks is to introduce another high-entropy secret into the protocols, similar to 1Password and Multipassword. This additional secret may initially seem like a management burden for users, but proper policies can mitigate this. For example, 1Password advises users to store the additional secret key on a piece of paper and keep it safe. Typically, users do not need to use the secret key frequently, as it is stored on their primary device. Only when using PMs on a new device do users need to enter this secret key.

Moreover, an additional secret can offset the costs associated with mitigating offline guessing attacks, such as the time cost of slow hashing or the memory cost of memory-hard functions. Besides, the entropy provided by the additional secret allows users to create relatively simple, easy-to-remember passwords without compromising security. However, as highlighted by the security flaw in Passbolt, simply adding another secret does not necessarily enhance security. The protocol must be carefully designed to ensure robustness.

7. Conclusion

In this paper, we present a systematic security analysis of master-password-protected password management (M3PM) protocols in password managers. For the first time, we identify M3PM protocols from 43 password managers from both academia and industry, define their functionalities in the UC framework, and conduct a security analysis. Our analysis reveals several security issues in current M3PM protocols, and we provide recommendations to address these issues. Given the crucial role of password managers, we believe our work significantly advances research on their protocols.

References

[1] *The 3rd Annual Global Password Security Report*, https://www.lastpass.com/state-of-the-password/global-password-security-report-2019.

[2] *Password managers: using browsers and apps to safely store your passwords*, Dec. 2021, https://www.ncsc.gov.uk/collection/top-tips-for-staying-secure-online/password-managers.

[3] *Why the password isn't dead quite yet*, Jul. 2021, https://arstechnica.com/information-technology/2021/07/why-the-password-isnt-dead-quite-yet/.

[4] *NIST SP 800-63: Digital Identity Guidelines Frequently Asked Questions.*, Mar. 2022, https://pages.nist.gov/800-63-FAQ/.

[5] *The Password Isn't Dead Yet. You Need a Hardware Key*, 2022, https://www.wired.com/story/hardware-security-key-passwords-passkeys/.

[6] *Have I been pwned?*, 2023, https://haveibeenpwned.com/.

[7] *LastPass Technical Whitepaper*, Aug. 2023, https://support.lastpass.com/download/lastpass-technical-whitepaper.

[8] *Full version of this paper and open supplementary materials*, 2024, https://anonymous.4open.science/r/pm-analysis-4661/.

[9] M. Abdalla, M. Barbosa, T. Bradley, S. Jarecki, J. Katz, and J. Xu, "Universally composable relaxed password authenticated key exchange," in *Proc. CRYPTO 2020*, vol. 12170, pp. 278–307.

[10] L. Abrams, *LastPass breach linked to theft of $4.4 million in crypto*, 2023, https://www.bleepingcomputer.com/news/security/lastpass-breach-linked-to-theft-of-44-million-in-crypto/.

[11] N. Alkaldi and K. Renaud, "Why do people adopt, or reject, smartphone password managers?" in *Proc. EuroUSEC 2016*.

[12] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko, "The one-more-rsa-inversion problems and the security of chaum's blind signature scheme," *J. Cryptol.*, vol. 16, no. 3, pp. 185–215, 2003.

[13] Bitwarden, *Bitwarden Security Whitepaper - Vault Data*, 2023, https://bitwarden.com/help/vault-data/.

[14] J. Blocki, B. Harsha, and S. Zhou, "On the economics of offline password cracking," in *Proc. IEEE S&P 2018*, pp. 35–53.

[15] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *Proc. ESORICS 2010*.

[16] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *Proc. IEEE S&P 2012*.

[17] J. Bonneau, C. Herley, P. van Oorschot, and F. Stajano, "Passwords and the evolution of imperfect authentication," *Commun. ACM*, vol. 58, no. 7, pp. 78–87, 2015.

[18] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE S&P 2012*, pp. 553–567.

[19] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proc. FOCS 2001*, pp. 136–145.

[20] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie, "Universally composable password-based key exchange," in *Proc. EUROCRYPT 2005*, vol. 3494, pp. 404–421.

[21] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart, "Cracking-resistant password vaults using natural language encoders," in *Proc. IEEE S&P 2015*, pp. 481–498.

[22] Y. Choong, "A cognitive-behavioral framework of user password management lifecycle," in *Proc. HAS 2014*, pp. 127–137.

[23] G. T. Davies, S. H. Faller, K. Gellert, T. Handirk, J. Hesse, M. Horváth, and T. Jager, "Security analysis of the whatsapp end-to-end encrypted backup protocol," in *Proc. CRYPTO*, 2023.

[24] A. Gangwal, S. Singh, and A. Srivastava, "Autospill: Credential leakage from mobile password managers," in *Proc. CODASPY 2023*.

[25] P. Gasti and K. B. Rasmussen, "On the security of password manager database formats," in *Proc. ESORICS 2012*, pp. 770–787.

[26] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, and J. P. Richer, *NIST SP 800-63B*, Jun. 2017, https://pages.nist.gov/800-63-3/sp800-63b.html.

[27] A. Groce and J. Katz, "A new framework for efficient password-based authenticated key exchange," in *Proc. ACM CCS 2010*, pp. 516–525.

[28] A. Hanamsagar, S. S. Woo, C. Kanich, and J. Mirkovic, "Leveraging semantic transformation to investigate password habits and their causes," in *Proc. ACM CHI 2018*, pp. 1–10.

[29] I. Ion, R. Reeder, and S. Consolvo, "'...No one can hack my mind': Comparing expert and non-expert security practices," in *Proc. SOUPS 2015*, pp. 327–346.

[30] S. Jarecki, H. Krawczyk, M. Shirvanian, and N. Saxena, "Device-enhanced password protocols with optimal online-offline protection," in *Proc. AsiaCCS 2016*, pp. 177–188.

[31] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks," in *Proc. EUROCRYPT 2018*, vol. 10822, pp. 456–486.

[32] C. Jon, D. Lutz, F. Hal, S. Daphne, and T. Rodney, "Openpgp message format," Nov. 2007, https://datatracker.ietf.org/doc/html/rfc4880.

[33] LastPass, *An Encryption Model that Prioritizes your Security*, 2023, https://www.lastpass.com/security/zero-knowledge-security.

[34] Y. Li, H. Wang, and K. Sun, "Bluepass: A secure hand-free password manager," in *Proc. SecureComm 2017*, pp. 185–205.

[35] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *Proc. USENIX SEC 2014*, pp. 465–479.

[36] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proc. IEEE S&P 2014*, pp. 689–704.

[37] P. Mayer, C. W. Munyendo, M. L. Mazurek, and A. J. Aviv, "Why users (don't) use password managers at a large educational institution," in *Proc. USENIX SEC 2022*, pp. 1849–1866.

[38] P. Mayer, Y. Zou, F. Schaub, and A. J. Aviv, "'Now i'm a bit angry: ' Individuals' awareness, perception, and responses to data breaches that affected them," in *Proc. USENIX SEC 2021*, pp. 393–410.

[39] D. McCarney, "Password managers: Comparative evaluation, design, implementation and empirical analysis," Master's thesis, Carleton University, 2013.

[40] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: Design, implementation, and usability evaluation of a password manager," in *Proc. ACM ACSAC 2012*, pp. 89–98.

[41] C. W. Munyendo, P. Mayer, and A. J. Aviv, ""i just stopped using one and started using the other": Motivations, techniques, and challenges when switching password managers," in *Proc. CCS 2023*.

[42] S. Oesch and S. Ruoti, "That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers," in *Proc. USENIX SEC 2020*.

[43] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart, "Beyond credential stuffing: Password similarity models using neural networks," in *Proc. IEEE S&P 2019*, pp. 417–434.

[44] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proc. ACM CCS 2017*, pp. 295–310.

[45] S. Pearman, S. Zhang, L. Bauer, and N. Christin, "Why people (don't) use password managers effectively," in *Proc. SOUPS 2019*.

[46] H. Ray, F. Wolf, R. Kuber, and A. J. Aviv, "Why older adults (don't) use password managers," in *Proc. USENIX SEC 2021*, pp. 73–90.

[47] B. Reviews, *Which Password Managers Have Been Hacked?*, 2023, https://password-managers.bestreviews.net/faq/which-password-managers-have-been-hacked/.

[48] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions." in *Proc. USENIX SEC 2005*, pp. 1–15.

[49] M. Shirvanian, N. Saxena, S. Jarecki *et al.*, "Building and studying a password store that perfectly hides passwords from itself," *IEEE Trans. Dependable Secur. Comput.*, vol. 16, no. 5, pp. 770–782, 2019.

[50] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *IACR Cryptol. ePrint Arch.*, 2004.

[51] D. Silver, S. Jana, D. Boneh, E. Y. Chen, and C. Jackson, "Password managers: Attacks and defenses," in *Proc. USENIX SEC 2014*.

[52] K. Toubba, *Notice of Recent Security Incident: The LastPass Blog*, 2022, https://blog.lastpass.com/2022/12/notice-of-recent-security-incident/.

[53] J. Wallen, *Six of the most popular Android password managers are leaking data*, 2023, https://www.zdnet.com/article/six-of-the-most-popular-android-password-managers-are-leaking-data/.

[54] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's law in passwords," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 11, pp. 2776–2791, 2017.

[55] D. Wang, P. Wang, D. He, and Y. Tian, "Birthday, name and bifacial-security: Understanding passwords of Chinese web users," in *Proc. USENIX SEC 2019*, pp. 1537–1555.

[56] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proc. ACM CCS 2016*, pp. 1242–1254.

[57] D. Wang, Y. Zou, Y.-A. Xiao, S. Ma, and X. Chen, "PASS2EDIT: A multi-step generative model for guessing edited passwords," in *Proc. USENIX SEC 2023*, pp. 1–18.

[58] L. Wang, Y. Li, and K. Sun, “Amnesia: A bilateral generative password manager,” in *Proc. IEEE ICDCS 2016*, pp. 313–322.

[59] D. Wheeler, “zxcvbn: Low-budget password strength estimation,” in *Proc. USENIX SEC 2016*, pp. 157–173.

[60] T. Wu, “A real-world analysis of kerberos password security,” in *Proc. NDSS 1999*, pp. 13–22.

[61] T. Wu, “The secure remote password protocol,” in *Proc. NDSS 1998*.

[62] T. Wu, “Srp-6: Improvements and refinements to the secure remote password protocol,” *http://srp.stanford.edu/srp6.ps*, 2002.

[63] A. Zaharia, *Have I Been Hacked? How To Recognize & Recover From a Hack*, 2023, https://www.aura.com/learn/have-i-been-hacked.

[64] R. Zhao and C. Yue, “All your browser-saved passwords could belong to us: A security analysis and a cloud-based new design,” in *Proc. CODASPY 2013*, pp. 333–340.

[65] R. Zhao, C. Yue, and K. Sun, “Vulnerability and risk analysis of two commercial browser and cloud based password managers,” *ASE Science Journal*, vol. 1, no. 4, pp. 1–15, 2013.

Appendix

A. Basic cryptographic security assumptions

IND-CPA security for public key encryption. For public key encryption $PKE = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, the indistinguishability of ciphertext under chosen plaintext attacks (IND-CPA) security refers to that adversary can choose plaintext for encryption but still cannot distinguish ciphertext. Formally, the adversary $\mathcal{A}$ can use the public key pk generated by KGen for encryption, and give the challenger two messages m_0, m_1. The challenger selects a bit $b \leftarrow\$ \{0,1\}$ randomly and encrypts one of the messages $c = \mathsf{Enc}_{pk}(m_b)$. Upon receiving the ciphertext c, the adversary $\mathcal{A}$ tries to distinguish which message is encrypted. The advantage of the adversary $\mathcal{A}$ for a correct distinguish b' is defined as $\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathcal{A},PKE}(n) = \Pr[b' = b] - \frac{1}{2}$.

IND-CPA security for authenticated encryption. For an authenticated encryption $AE = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, the indistinguishability of ciphertext under chosen plaintext attacks (IND-CPA) security refers to that adversary can choose plaintext for encryption but still cannot distinguish ciphertext. Formally, the adversary $\mathcal{A}$ is given an encryption oracle $\mathsf{Enc}_K()$ where the key K is generated by KGen, and sends challenger two message m_0, m_1. The challenger selects a bit $b \leftarrow\$ \{0,1\}$ randomly and encrypts one of the messages $c = \mathsf{Enc}_p k_C(m_b)$. Upon receiving the ciphertext c, the adversary $\mathcal{A}$ tries to distinguish which message is encrypted. The advantage of the adversary $\mathcal{A}$ for a correct distinguish b' is defined as $\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathcal{A},AE}(n) = \Pr[b' = b] - \frac{1}{2}$.

INT-CTXT security for authenticated encryption. For an authenticated encryption $AE = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, the integrity of ciphertext (INT-CTXT) security refers to the fact that the adversary cannot create a new correct ciphertext. Formally, the adversary $\mathcal{A}$ is given an encryption oracle $\mathsf{Enc}_K()$ where the key K is generated by KGen and gives the challenger a new ciphertext c which has not been generated by $\mathsf{Enc}_K()$ oracle. The advantage of the adversary $\mathcal{A}$ for a correct distinguish b' is defined as $\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathcal{A},AE}(n) = \Pr[\mathsf{Dec}_K(c) \neq \perp]$.

The one-more gap DH assumption [12]: Let $\mathbb{G}$ represent a group of prime order q and k be a random value in $\mathbb{Z}_q$. The one-more gap DH problem provides two oracles: DH_k which outputs g^k on input $g \in \mathbb{G}$, and DDH_k which outputs whether $b = a^k$ on input (a, b). And given q_{DH} times DH_k oracle queries and q_{DDH} times DDH_k oracle queries, the one-more gap DH problem is to output $q_{DH} + 1$ different pairs (g, g^k) on input a set of random element R and each $g \in R$. The one-more gap DH assumption is that any probabilistic polynomial time adversary $\mathcal{A}$ can solve the one-more gap DH problem at most a negligible probability $\mathsf{Adv}^{\mathrm{omg-dh}}_{\mathcal{A},q_{DH},q_{DDH}}(n)$.

The functionality $\mathcal{F}_{pwAuth}$. The M3PM protocol of 1Password uses SRP protocol to authenticate the client, and not use the generated session key for further communication (all communication is protected by TLS). Therefore, we slightly change the functionality of Jarecki et al.’s [31] and Groce and Katz’s work [27], and give a functionality called $\mathcal{F}_{pwAuth}$. The details of functionality $\mathcal{F}_{pwAuth}$ are depicted in Fig. 8. In the M3PM protocol of 1Password, the client and server use SRP to authenticate the client and then the server sends the encrypted vault to the authenticated client. Therefore, we give the functionality only with client-to-server authentication and neglect the part of generating the session key.

B. Security functionality

We describe our ideal functionality in Fig. 4. The functionality $\mathcal{F}_{M3PM}$ represents the ideal scenario for a two-party M3PM protocol executed by a client and a server.

Generally, $\mathcal{F}_{M3PM}$ maintains a lookup table for authentication and password retrieval. This lookup table, encoded in VAULT, contains the user’s identity ID_C, the server’s identity ID_S, the master password mpw, login metadata $login$, and the password pwd. The functionality $\mathcal{F}_{M3PM}$ uses ID_C and ID_S to look up vaults, $login$ to look up passwords, and mpw for authentication. The functionality $\mathcal{F}_{M3PM}$ includes both initialization and retrieval phases, and any queries for a non-existent record will be ignored by $\mathcal{F}_{M3PM}$.

Initialization phase. In initialization phase, a client ID_C and a server ID_S initialize a password pwd for one credential $login$. During initialization, the client sets a master password mpw for authentication in the retrieval phase.

A client ID_C starts a new initialization and calls INITC query with master password mpw, login metadata $login$, and password pwd. If the password pwd is ϵ, the functionality $\mathcal{F}_{M3PM}$ will generate a random password $pwd \leftarrow\$ \{0,1\}^n$ for client, and marks the password GENERATED (IC.1). Then, $\mathcal{F}_{M3PM}$ records this initialization session and makes sure only one record for each session (IC.2). Finally, the functionality sends a message to inform adversary $\mathcal{A}$ and server ID_S that the client has initialized a new

Functionality $\mathcal{F}_{pwAuth}$

On input $(\textsc{StorePwdFile}, sid, ID_C, pw)$ from ID_S (or $\mathcal{A}$ if ID_S is corrupted):

- If there is the first StorePwdFile for ID_C, record $\langle\textsc{File}, ID_C, ID_S, pw\rangle$ and mark it Uncompromised

On input $(\textsc{StealPwdFile}, sid, ID_C)$ from $\mathcal{A}$:

- If there is no File record for ID_C, then return "no password file" to $\mathcal{A}$
- If there is File record marked Uncompromised, then mark it Compromised
- If there is a record $\langle\textsc{Offline}, pw\rangle$, then send pw and to $\mathcal{A}$. Else, send "password file stolen" to $\mathcal{A}$

On input $(\textsc{OfflineTestPwd}, sid, pw')$ from $\mathcal{A}$:

- If $\langle\textsc{File}, ID_C, ID_S, pw\rangle$ is marked Compromised, then if $pw' = pw$, return Correct to $\mathcal{A}$; else Wrong
- Else record $\langle\textsc{Offline}, pw\rangle$

On input $(\textsc{NewSession}, sid, ID_C, ID_S, pw, \text{Client})$ from ID_C (or $\mathcal{A}$ if ID_C is corrupted):

- Store a record $(ID_C, ID_S, sid, pw, \text{Client})$ labeled Fresh
- Send $(\textsc{NewSession}, sid, ID_C, ID_S, \text{Client})$ to the adversary $\mathcal{A}$

On input $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ from ID_S (or $\mathcal{A}$ if ID_S is corrupted):

- Retrieve record $\langle\textsc{File}, ID_C, ID_S, pw\rangle$
- If a $(ID_C, ID_S, sid, pw, \text{Client})$ exists, store $(ID_S, ID_C, sid, pw, \text{Server})$ labeled Fresh
- Send $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ to the adversary $\mathcal{A}$

On input $(\textsc{TestPwd}, sid, ID_C, pw')$ from the adversary $\mathcal{A}$:

- If there is no stored record $(ID_C, ID_S, sid, pw, \text{role})$ marked Fresh, then abort. Otherwise, retrieve it
- If $pw = pw'$, label the record Compromised and reply $\mathcal{A}$ with Correct guess
- If $pw \neq pw'$, label the record Interrupted and reply $\mathcal{A}$ with Wrong guess

On input $(\textsc{CompleteAuth}, sid, ID_S, ID_C)$ from adversary $\mathcal{A}$:

- Retrieve the record $(ID_S, ID_C, sid, pw, \text{Server})$, and if it is marked Completed, then abort
- If the record is Compromised, or either ID_C or ID_S are corrupted, send Success to ID_S and $\mathcal{A}$
- If the record is Interrupted, send Fail to ID_S and $\mathcal{A}$
- Mark the record $(ID_S, ID_C, sid, pw, \text{Server})$ as Completed

Figure 8: Ideal functionality $\mathcal{F}_{pwAuth}$ for password-based authentication.

password (IC.3). If ID_C is corrupted, the adversary $\mathcal{A}$ can send the InitC query.

If the server ID_S agrees to incorporate into the initialization session, the ID_S sends an InitS query to $\mathcal{F}_{M3PM}$. The functionality retrieves the record that the client has initialized, and gets the value of the master password mpw, login metadata $login$, and password pwd (IS.1). If there exists such a record, then the $\mathcal{F}_{M3PM}$ records a tuple $\langle\textsc{Vault}, ID_C, ID_S, mpw, login, pwd\rangle$ and makes sure that one client only has one password for credentials (IS.2). Finally, the $\mathcal{F}_{M3PM}$ informs adversary $\mathcal{A}$ that the server ID_S has incorporated into the session (IS.3). If ID_S is corrupted, the adversary $\mathcal{A}$ can send the InitS query.

We let adversary $\mathcal{A}$ determine when to complete the protocol for modeling the ability that adversary can interrupt the message [19], [23]. The adversary can send a CompleteInitC query to determine the finish of the client. The functionality $\mathcal{F}_{M3PM}$ searches for a record for client initialization. If $\mathcal{F}_{M3PM}$ finds this record, it means that the client is ready to finish the session, then the $\mathcal{F}_{M3PM}$ deletes the record for session completed (CIC.1). Next, the $\mathcal{F}_{M3PM-EA}$ functionality checks whether there are vault records for ID_C and ID_S. If there exists a record $\langle\textsc{Vault}, ID_C, ID_S, mpw', *, *\rangle$, the functionality verifies the correctness of mpw (CIC.2). This step captures the practical scenario where, after registering a vault in password managers, the M3PM protocol with explicit authentication will authenticate the master password and then initialize passwords.

Similarly, the adversary $\mathcal{A}$ can send a CompleteInitS query, and the functionality will store a record represented by the password vault (CIS.1), and the server ID_S checks whether the master password is correct (CIS.2). During the CompleteInitC and CompleteInitS processes, the adversary gains no information about mpw or pwd. This ensures that the obliviousness of the master password and the password vault are maintained throughout these processes.

Remark 4. In $\mathcal{F}_{M3PM-EA}$, we do not support updating the master password. Each time the client ID_C initializes a master password with ID_S, it cannot initialize passwords with a different master password later. This issue can be resolved by introducing an update phase that modifies the vault record in the functionality, which we leave for future work.

Retrieval phase. In retrieval phase, a client ID_C retrieves the password pwd for credential $login$ with the corperation of a server ID_S by inputting the correct master password mpw. If the wrong master password is input, the server ID_S will receive a FAIL message. The output of the functionality $\mathcal{F}_{M3PM}$ for a wrong password depends on whether the M3PM protocol has explicit or implicit authentication. A FAIL message is for explicit authentication $\mathcal{F}_{M3PM-EA}$ and a random password is for implicit authentication $\mathcal{F}_{M3PM-IA}$.

A client ID_C starts a new retrieval session by a RETRC query with master password mpw' and login metadata $login$. The functionality records this query and the master password mpw' and makes sure that only one session has started (RC.1). Then the functionality sends a message to inform adversary $\mathcal{A}$ and server ID_S that the client has started a new retrieval session (IC.2). Specially, the adversary $\mathcal{A}$ can also send RETRC query for online guessing.

After the client ID_C finishes the RETRC query, the server ID_S can incorporate into the session sid by sending a RETRS query. The functionality $\mathcal{F}_{M3PM}$ first retrieves the record that the client has stored in RETRC query (RS.1). Then, the functionality checks for stored vault records, and non-existence represents the failure of retrieval (RS.2). If there exists such a record, the functionality will append the correct master password mpw into the RETRC record for the further check (RS.3). The functionality sends the adversary $\mathcal{A}$ about whether the retrieval is a success (RS.4).

Same with the initialization phase, the adversary determines the completion of the retrieval session. The adversary sends a COMPLETERETRC query to functionality, and the functionality retrieves for the RETRC record. The functionality marks the RETRC record RETRIEVED to make sure this record will be deleted after the client and server both retrieve it (CRC.1). After that, the functionality checks two master passwords $mpw \overset{?}{=} mpw'$. If they are equal, the functionality prepares to send the correct password. And the functionality $\mathcal{F}_{M3PM-EA}$ will alert the correct result to the $\mathcal{A}$ (CRC.2). Otherwise, the functionality responds with two options. For explicity authentication $\mathcal{F}_{M3PM-EA}$, the functionality returns a FAIL message (CRC.3(b)); For implicit authentication $\mathcal{F}_{M3PM-IA}$, the functionality returns a random password. If the password was generated by functionality, then return a uniformly distributed password, else return a Zipf-distributed [54] password (CRC.3(a)). Finally, the functionality sends the result to the client or adversary (CRC.4).

The adversary can send a COMPLETERETRS query. First, the functionality retrieves the RETRC record and does the same process as COMPLETERETRC query (CRS.1). The response of the functionality is simple: if the master password is checked successfully, then return SUCC to the server ID_S (CRS.2), else return FAIL to the server (CRS.3). If the server ID_S is corrupted, we allow $\mathcal{A}$ to have a TESTMPW query in $\mathcal{F}_{rM3PM}$. The adversary $\mathcal{A}$ guesses a mpw, and the functionality checks the correctness of the guess (TM.1). If the guess mpw is correct, the functionality sends the adversary the correct password pwd, else the functionality generates a wrong pwd' which is a random value or FAIL (TM.2).

In the $\mathcal{F}_{rM3PM}$ functionality, the adversary $\mathcal{A}$ can make a TESTMPW query to capture the threat of a guessing attack. Our analysis shows that an M3PM protocol that cannot resist offline guessing attacks when the server is compromised also cannot resist online guessing attacks. Therefore, we provide only one query API. The adversary tests the correctness of the master password (mpw), and the functionality checks and responds accordingly. Additionally, we emphasize that $\mathcal{F}_{adM3PM}$ can be achieved by replacing all instances of mpw with a tuple $(mpw, SecretK)$.

Remark 5. In our functionality, we omit the scenario where a correct master password is used, but non-existent login metadata is retrieved. For simplicity, this situation can be addressed by directly sending a FAIL message to the client.

C. The taxonomy of M3PM protocols

M3PM protocols consist of two main stages: authentication and password processing. During authentication, the system checks for the correct master password and explicitly outputs a failure if incorrect. Upon successful authentication, the client and server proceed to the password-processing stage to initialize and retrieve passwords.

Authentication stage. The details of the explicit authentication stage are depicted in Fig. 2. There are two participants, i.e., client and server, in the protocols. The client refers to the PM client application, and the client has a unique identity ID_C which can be users' emails. The client possesses users' master passwords mpw, and the client initializes the protocols by sending users' identities at first (we omit this message). Server refers to the PM cloud service, and the server has a unique identity ID_S, i.e., domain. For different authentication methods, the server possesses different items, but all are for authenticating master passwords.

For *hashed* mpw, the server sends the client salt, and the client uses the salt $salt$ and master password mpw to derive an authenticator $auth = H(mpw, salt)$ by hash functions $H : \{0,1\}^* \rightarrow \{0,1\}^n$. The server receives the authenticator $auth$ and compares its own store $hpw = H(mpw, salt)$ which is created at the registration stage (for simplicity, we omit the registration stage and assume the registration stage is under a secure channel). For *protected tokens*, the server sends encrypted authenticator $eauth = \mathsf{Enc}_{H(mpw)}$, and the client uses master password mpw to derive the master key $mk = H(mpw)$ for decrypting the encrypted authenticator $auth' = \mathsf{Dec}_{mk}(eauth)$. The server receives the authenticator $auth'$ and verifies whether $auth' = auth$. If the client stores the encrypted authenticator $eauth$, it can send the authenticator directly which can save one communication round. For the mpw *challenge*, the server generates a uniformly random challenge value $cha \leftarrow\$ \{0,1\}^n$ with length n and sends it to the client. The client uses master password mpw to encrypt the challenge $auth = auth \oplus H(mpw, salt)$, and the server receives the authenticator $auth$ to verify whether $auth \oplus hpw = cha$.

For *SRP*, the server and client run a secure remote password (SRP) protocol. The server samples a uniformly random $b \leftarrow\$ \{0,1\}^n$, generates value $B = 3v + g^b$, and sends the salt $salt$ and B to the client. On receiving $salt, B$ from the server, the client uses the master password to generate a secret value $x = H(mpw, ID_C, salt)$. Specifically, 1Password and Multipassword use a secret key SK (a random string generated by the server and sent to the users for keeping it) to derive secret value $x = H(mpw \oplus SK, ID_C, salt)$. Next, the client picks a random $a \leftarrow\$ \{0,1\}^n$ and generates $A = g^a$. For message integrity, the client generates $u = H(A, B)$ and generates the exchanged secret $S = (B - 3g^x)^{a+ux}$. Finally, the client outputs A and $M_1 = H(A, B, S)$ for authentication. On receiving (B, M_1), the server computes the same value $S = (Av^u)^b$, and verifies whether $M_1 = H(A, B, S)$. SRP is a mature password-authenticated key exchange (PAKE) protocol, and we omit the underlying principle for SRP and recommend readers for more details on [61], [62].

For the *pk challenge*, the server chooses a random value $n \leftarrow\$ \{0,1\}^n$ and uses the user's public key pk_C to generate challenge $cha = \mathsf{Enc}_{pk_C}(n)$. On receiving cha, pk_C, esk, the client first uses master password mpw to derive master key mk, and then decrypts the private key $sk_C = \mathsf{Dec}_{mk}(esk)$ to verify the challenge $n' = \mathsf{Dec}_{sk_C}(cha)$. Upon receiving the value n', the server checks for the possession of the master password by verifying if $n' = n$.

Remark 6. To be more precise, the authentication stage encompasses a registration phase. During registration, the client submits the authenticator to the server for enrollment. Specifically, for *hashed* mpw and mpw *challenge*, the client gets the $salt$ and sends $hpw = H(mpw, salt)$ to the server; For *protected tokens*, the client gets the $auth$ and sends $eauth = \mathsf{Enc}_{H(mpw)}(auth)$ to the server; For *SRP*, the client gets $salt$ and send $v = g^{H(mpw, ID_C, salt)}$; For *pk challenge*, the client generates a public key pair, encrypts the private key, and sends server $pk_C, esk = \mathsf{Enc}_{H(mpw)}(sk_C)$. In Fig. 2, we omit the process of registration for simplicity.

Password processing stage. M3PM protocols have two main methods for password-processing, i.e., storing and generating. The storing method makes a PM like an encrypted database, and users can store their passwords (human-created or password-manager-created) and retrieve them on demand. The encrypted database can be encrypted by master passwords (mpw *encrypt*), or device keys (*device-key encrypt*). And further, the database can be encoded first and then encrypted (*encode-then-encrypt*). For *generating* PMs, PMs take the master password, and login data fields as input, and generate unique secret passwords for each login credential.

The detailed descriptions of the password-processing stage of M3PM are depicted in Fig. 3. The server stores encrypted password vaults $eVault$ for mpw *encrypt*, *device-key encrypt*, and *encode-then-encrypt*. It also stores encrypted keys evk for password vaults in *device-key encrypt*. Before retrieving passwords, the M3PM protocol must initialize them. Initialization corresponds to real-world scenarios where users register credentials on a website, allowing password managers to manage those credentials.

After authentication, the server sends the encrypted vault $eVault$ to the client. For mpw *encrypt*, the client uses master password mpw to derive master key $mk = H(mpw, salt)$ and uses the key mk to decrypt the vaults $Vault = \mathsf{Dec}_{mk}(eVault)$. In the initialization phase, the client stores the password pwd which can be input by humans or generated by the client into the vault $Vault.set(login, pwd)$ with corresponding login metadata $login$. Finally, the client encrypts the vault $eVault' = \mathsf{Enc}_{mk}(Vault)$ and sends it back to the server. While in the retrieval phase, the client receives the encrypted vaults $eVault$ and decrypts it. The client uses login metadata $login$ to retrieve password pwd.

The process is similar to *device-key encrypt* and *encode-then-encrypt*, and we point out the difference between them. For *device-key encrypt*, the server also stores encrypted vault key evk and sends it to the client. The client uses the master key $mk = H(mpw, salt)$ to decrypt the vault key $vk = \mathsf{Dec}_{mk}(evk)$ and uses the vault key vk to decrypt the encrypted vault $Vault = \mathsf{Dec}_{mk}(eVault)$. The encrypted vault key can be a series of encrypted keys. For example, 1Password and NordPass use master password mpw to encrypt a secret key sk and use the secret key sk to encrypt the vault key vk. Multiple keys can be a benefit for usability: (1) Encrypting vaults with the device key helps circumvent the need to re-encrypt all vaults when changing the master password; (2) The key, encrypted by the private key, can be shared with other users, enabling the sharing of vaults. We do not focus on the usability of M3PM, so we will leave these functions for future work.

For *encode-then-encrypt*, the client adopts an encoding algorithm before encrypting the vaults. The encoding algorithm $encode(X, Vault)$ takes the vault $Vault$ and a distribution X as input, and outputs the seed $Seed$ for the following encryption. The core function of the encoding algorithm is to map the message from the distribution X to the plaintext space of the encryption algorithm uniformly. In decryption, the client decrypts the seed $Seed = \mathsf{Dec}_{mk}(eVault)$ and maps the seed to the plaintext $Vault = decode(X, Seed)$. The encoding/decoding algorithm is designed for resisting offline guessing which we will discuss in Section 5.1.

For *generative*, the server stores nothing about the user's password vault, and the user needs master password mpw and login metadata $login$ to generate unique passwords pwd for each credential. The user can select the parameters par for generation. For example, the user can set a counter ctr to generate multiple passwords for one credential; SPHINX [49] uses a device to store parameter k for each credential and generate passwords pwd by OPRF function. Generally, the user uses hash function $H : \{0,1\}^* \to \{0,1\}^n$ to generate passwords $pwd = H(mpw, login, par)$, and the user first generates the password pwd for initialization and then does the same process for retrieval.

D. The algorithms of PMs

As seen in Table 2, 36 of 43 PMs use Advanced Encryption Standard (AES) as a symmetric encryption algorithm and only CSF-BPM [64] uses AES with 128-bit key length instead of 256-bit key length. NordPass adopts XChaCha20 for easier implementation [3], and Password Safe chooses Twofish to encrypt password vaults. NoCrack [21] uses DTE to encode password vaults before encrypting them, and it utilizes AES-256 for the encryption of the vaults. Lesspass, SPHINX [49], and Amnesia [58] do not adopt symmetric encryption. BluePass [34] and Kamouflage [15] do not mention their specific symmetric algorithm selection.

Nine PMs use modern Authentication Encryption with Associated Data (AEAD) primitives which encrypt and authenticate messages in one algorithm. Nine PMs use the Galois/Counter mode of AES, and fourteen PMs use HMAC with hash functions to authenticate messages. PMs without AEAD or HMAC do not authenticate messages, and PMs with Cipher Block Chaining (CBC) mode (14 PMs) and Counter (CTR) mode (four PMs) have indistinguishability security under chosen-plaintext attack (IND-CPA). NordPass, Dropbox PM, Norton PM, McAfee True Key, Sticky PM, Delinea Web PM, LogMeOnce, SafeInCloud PM, and Gopass do not give details about encryption mode.

Beyond the ones highlighted in Table 2, other PMs incorporate asymmetric encryption, primarily for password sharing. Bitwarden, for example, utilizes RSA to facilitate password sharing. Because we only focus on password retrieval, we do not record asymmetric algorithms for password sharing. 1Password and Padloc adopt RSA-OAEP, NordPass adopts X25519-XSalsa20, Passwarden adopts ECDH-secp384r1, Passbolt adopts RSA-2048, and Gopass adopts Elgamal Diffie-Hellman. All the above PMs utilize asymmetric encryption to encrypt vault keys. Besides, SPHINX [49] employs elliptic curve NIST P-256 for constructing oblivious pseudorandom functions (OPRF), and BluePass [34] uses RSA for encrypting password vaults.

All PMs except Chrome PM, SPHINX [49], BluePass [34], Amnesia [58], Kamouflage [15], and Tapas [40] use key derivation functions (KDF). 22 PMs adopt a password-based key derivation function (PBKDF2), and iCloud Keychain uses PBKDF2 with 10 million iterations to derive a secure key for keybag (iCloud Keychain calls the set of keys "keybag"). In NIST standard 800-63B [26], PBKDF2 iterations are recommended to be at least 10,000 iterations. However, Avira PM, McAfee True Key, Password Safe, TeamPass, and NoCrack adopt iteration numbers of no more than 10,000, which is relatively low compared with other PMs.

In Blocki et al.'s work [14], they conclude that PBKDF2 is not secure enough for password hashing, and they recommend memory-hard functions (MHF), such as Argon2. Dashlane, NordPass, Dropbox PM, Passwarden, IronVest, Bitdefender PM, KeeWeb, and Authpass adopt Argon2. Specifically, there are other PMs supporting MHF, such as Bitwarden and KeePass. However, we only collect the default settings of each PM because we make an assumption that most ordinary users will not change the encryption settings. We do not find the Argon2id parameters for NordPass or the KDF information about Norton PM, SafeInCloud PM, and Delinea Web PM. Passbolt and Gopass adopt OpenGPG [32], a message encryption standard with symmetric and asymmetric hybrid encryption without KDF. We only survey the local version of Chrome PM, and the local Chrome PM does not use master passwords for encryption, so it does not need a KDF.

[3] NordPass gives four reasons for choosing XChaCha20 at https://nordpass.com/features/xchacha20-encryption/

E. The metadata of PMs

All the information about metadata is described in Table 3. The *secret* refers to the stored passwords, and *login* metadata refers to other information about the credentials, i.e., websites' domains (including websites' icons) and accounts' usernames. The *item state* metadata encompasses details about a credential stored in password vaults, including the creation time, the modification time of the item, the last usage timestamp, and the count of how many times the item has been used. The *password hygienism* metadata refers to the evaluation results of each password, such as whether the password is weak. The *users' profile* refers to users' emails which are used to register a PM account, and other users' settings such as iterations of the key derivation function.

Compared with Gasti et al.'s [25] and Oesch and Ruoti's findings [42], we give a broader view of the encryption on metadata and gain more insights on PMs. In all, KeePass, KeePassXC, DualSafe PM, Norton PM, Password Safe, KeeWeb, and Authpass store all metadata encrypted, and LessPass, Gokey, SPHINX [49], and Amnesia [58] are generative PMs without storing any metadata.

39 PMs encrypt *login* metadata, 12 of 39 PMs store websites' URLs in plaintext, and 10 of 39 PMs store accounts' usernames in plaintext. A plaintext *login* metadata gives adversaries the advantage of distinguishing password vaults and leaking users' privacy, but also has vulnerabilities in malleability which we will discuss in Section 5.2. The *item state* metadata is used to control the validity of the stored credential items. For example, Bitwarden does not remove the information of deleted items but labels them as deleted to give users a chance to recover them. 16 of 43 PMs store the creation time in plaintext, and 14 of 43 PMs store the modification time in plaintext. These time-related state metadata are useful for synchronization, and the cloud servers can refer to the time to update the password vaults. Therefore, we conjecture that the PMs store time-related state metadata encrypted and may create temporary information for synchronization.

There are 11 PMs that store information about *password hygiene*, with five of them storing this information in plaintext. For example, Padloc records whether a password is weak or reused in the metadata. This type of unencrypted metadata poses a significant security threat: plaintext *password hygiene* data not only allows an adversary to

TABLE 2: Overview of encryption algorithms used in password managers. [†]

Password manager@Version	Symmetric encryption			Asymmetric encryption	Hash function		Key derivation function	
	Algorithm	Length	Mode	Algorithm	Algorithm	Length	Algorithm	Parameters
Bitwarden@2023.9.1	AES	256	CBC	-	HMAC-SHA256	256	PBKDF2	600,000
1Password@8.10.16	AES	256	GCM	RSA-OAEP	HMAC-SHA256	256	PBKDF2	650,000
LastPass@4.123.0	AES	256	CBC	-	SHA256	256	PBKDF2	600,000
Dashlane@6.2342.0	AES	256	CBC	-	Argon2	256	Argon2d	3 rounds / 32 Mib Memory
RoboForm@9.5.2	AES	256	CBC	-	HMAC-SHA256	256	PBKDF2	100,000
Keeper@16.10.9	AES	256	GCM	-	HMAC-SHA256	256	PBKDF2	1,000,000
NordPass@5.8.21	XChaCha20	256		X25519-XSalsa20	Argon2	256	Argon2id	
Zoho Vault@3.8	AES	256	CTR	-	HMAC-SHA256	256	PBKDF2	310,000
KeePass@2.55	AES	256	CBC	-	HMAC-SHA256	256	AES-KDF	
KeePassXC@2.7.6	AES	256	CBC	-	HMAC-SHA256	256	AES-KDF	
Dropbox PM@3.26.0	AES	256		-	Argon2	256	Argon2id	2 rounds / 256 Mib Memory
DualSafe PM@1.4.21	AES	256	CBC	-	SHA256	256	PBKDF2	100,000
Enpass@6.9.1	AES	256	CBC	-	HMAC-SHA256	256	PBKDF2	320,000
Norton PM@8.1.0.73	AES	256		-				
Avira PM@2.19.13.44521	AES	256	CBC	-	HMAC-SHA256	256	PBKDF2	1,000
McAfee True Key@4.3.1.9339	AES	256		-	HMAC-SHA256	256	PBKDF2	10,000
Sticky PM@8.8.3.1629	AES	256		-			PBKDF2	several thousands
Password Boss@5.5.5138.0	AES	256	CBC	-	HMAC-SHA1	160	PBKDF2	64,000
Multipassword@0.93.36	AES	256	GCM	-	SHA256	256	PBKDF2	100,000
Passwarden@1.3.3	AES	256	GCM	ECDH-secp384r1	SHA512	512	Argon2id	4 rounds / 16 Mib Memory
Delinea Web PM@3.7	AES	256		-	HMAC-SHA256	256	PBKDF2	
LogMeOnce@7.8.0	AES	256		-	HMAC-SHA512	512	PBKDF2	600,000
Passbolt@4.4.2	AES	256	GCM	RSA-2048			-	-
IronVest@9.8.15	AES	256	CBC	-	SHA512	512	Argon2id	3 rounds / 19 Mib Memory
SafeInCloud PM@22.3.4	AES	256		-				
Password Safe@3.64.1	Twofish	128	CBC	-	SHA256	256	PBKDF2	2,048
Bitdefender PM@3.21.1247	AES	256	CCM	-	SHA512	512	Argon2id	3 rounds / 64 Mib Memory
iCloud Keychain@Ventura 13.5.2	AES	256	GCM	-			PBKDF2	10,000,000
Chrome PM@119.0.6045.160	AES	256	GCM	-	-	-	-	-
KeeWeb@1.18.7	AES	256	CBC	-	-	-	Argon2	2 rounds / 1 Mib Memory
Lesspass@9.1.9	-	-	-	-	SHA256	256	PBKDF2	100,000
Gopass@1.15.10	AES	256		Elgamal Diffie-Hellman	SHA1	160	-	-
Padloc@4.3.0	AES	256	GCM	RSA-OAEP	HMAC-SHA256	256	PBKDF2	1,000,000
Authpass@1.9.9	AES	256	CBC	-	-	-	Argon2	2 rounds / 1 Mib Memory
Gokey@0.1.2	AES	256	CTR	-	SHA56	256	PBKDF2	4,096
TeamPass@3.0.10	AES	256	CTR	-	SHA256	256	PBKDF2	100,000
SPHINX [49]	-	-	-	elliptic curve NIST P-256	SHA256	256	-	-
BluePass [34]	-	-	-	RSA	-	-	-	-
NoCrack [21]	AES + DTE	256	CTR	-	SHA256	256	PBKDF1	100
Amnesia [58]	-	-	-	-	SHA256	256	-	-
Kamouflage [15]	-	-	-	-	-	-	-	-
CSF-BPM [64]	AES	128	CCM	-	SHA1	160	PBKDF2	-
Tapas [40]	AES	256	GCM	-	-	-	-	-

[†] PMs may adopt various algorithms. For instance, Bitwarden may use PBKDF2 and Argon2 for key derivation, but we only record the default function. We make this decision because we believe that ordinary users may not change the default settings on cryptographic algorithms without sufficient expertise, and it is the responsibility of PMs to ensure the security of the algorithms they use. The blank items indicate that we do not have enough information on these cryptographic algorithms or parameters. The ”-” items denote that the PMs do not require this cryptographic algorithm.

distinguish between password vaults but also helps identify weak or reused passwords. Consequently, if an adversary gains access to the password vaults, they can use this metadata to aid in guessing passwords.

During our analysis, 22 of 31 PMs store unencrypted users' emails. 14 of 29 PMs store users' settings in plaintext, and this information contains the iterations of key derivation functions, whether users unlock vaults with a master password or the time of auto-lock. All these give adversaries more information to conduct attacks on PMs. We do not determine which attack the adversary can conduct, but we are sure that the leak of users' settings is harmful.

F. Proof of 1Password

Proof. Let $Real_{\mathcal{Z},\mathcal{A},OP}$ be the probability that environment $\mathcal{Z}$ interacting with adversary $\mathcal{A}$ and real-world protocol OP thinks it interacts with the real-world protocol, i.e., output 1, and let $Ideal_{\mathcal{Z},SIM_{OP},\mathcal{F}_{M3PM-EA}}$ be the probability that environment $\mathcal{Z}$ interacting with simulator SIM_{OP} and ideal-world functionality $\mathcal{F}_{M3PM}$ outputs 1. Our analysis is based on sequence-of-games proof strategy [50]. We define a series of games from the real-world execution to ideal-world execution and prove that the environment $\mathcal{Z}$ can distinguish each two games in a negligible probability, which means the environment cannot distinguish the real world and the ideal world. The simulator for the M3PM protocol of 1Password is depicted in Fig. 9, and now we give the details of the games.

Game G_0: This game is in the real world and the adversary $\mathcal{A}$ interacts with the real players following the protocol OP.

$$Real_{\mathcal{Z},\mathcal{A},OP} = \Pr[G_0] \tag{1}$$

Game G_1: (Simulate real world). In this game, we perform simulation on the real players to make the adversary $\mathcal{A}$ and environment $\mathcal{Z}$ interact with the simulator. The simulation in this game is exactly the same as the real game, except the simulator stores records to simulate random oracles and all transcripts between the client ID_C and the server ID_S. In all, this change is just for turning the real game into a game in the UC framework with only syntactical change.

$$\Pr[G_1] = \Pr[G_0] \tag{2}$$

TABLE 3: Overview of metadata in password managers. †

Password manager@Version	Login			Item state				Password hygienism	User's profile	
	URL	Icon	Username	Creation	Modification	Last use	Fill count	Password security	User's email	User's settings
Bitwarden@2023.9.1	●	●	●	○	○				○	○
1Password@8.10.16	●	●	●	●	●				○	●
LastPass@4.123.0	●	●	●	●	●				○	○
Dashlane@6.2342.0	●	●	●	●	●	●	●		○	●
RoboForm@9.5.2	●	●	●	●	●	●	●	○	○	●
Keeper@16.10.9	●	●	●	●	●	●	●		○	●
NordPass@5.8.21	●	●	●	○	○				○	○
Zoho Vault@3.8	○		○	○					○	○
KeePass@2.55	●	●	●	●	●	●	●		●	●
KeePassXC@2.7.6	●	●	●	●	●	●	●		●	●
Dropbox PM@3.26.0	●	●	●	●	●	●	●	●	○	●
DualSafe PM@1.4.21	●	●	●	●	●	●	●	●	●	●
Enpass@6.9.1	●	●	●	●	●	●	●	●		○
Norton PM@8.1.0.73	●	●	●	●	●	●	●	●	●	●
Avira PM@2.19.13.44521	●	●	●	○	○				○	
McAfee True Key@4.3.1.9339	○		○	○	○		○		○	○
Sticky PM@8.8.3.1629	●	●	●	○	○			●	●	●
Password Boss@5.5.5138.0	●	●	●	○	○	●	●	●	○	●
Multipassword@0.93.36	●	●	●	○	○				○	○
Passwarden@1.3.3	●	●	●	○	○					
Delinea Web PM@3.7	○	○	○						○	○
LogMeOnce@7.8.0	○	○	●							
Passbolt@4.4.2	○	○	○	○	○			○	○	
IronVest@9.8.15	●		●	○	○				○	○
SafeInCloud PM@22.3.4	○	○	○	○	○	○	○	○	○	○
Password Safe@3.64.1	●	●	●	●	●	●			●	●
Bitdefender PM@3.21.1247	●	○	●	○	○		●		○	○
iCloud Keychain@Ventura 13.5.2	●	●	●	●	●					
Chrome PM@119.0.6045.160	○	●	○	○	○	○	○	○	●	●
KeeWeb@1.18.7	●	●	●	●	●	●	●		●	●
Lesspass@9.1.9										
Gopass@1.15.10	○									
Padloc@4.3.0	●		●	○	○			○	○	○
Authpass@1.9.9	●	●	●	●	●	●	●		●	●
Gokey@0.1.2										
TeamPass@3.0.10	○	○	○	○					○	○
SPHINX [49]										
BluePass [34]	○		○						○	
NoCrack [21]	○		○							
Amnesia [58]										
Kamouflage [15]	○		○							
CSF-BPM [64]	●		●						○	○
Tapas [40]	●	●	●							

† ● : The metadata is encrypted; ○ : The metadata is in plaintext; Empty: The password manager does not store this kind of metadata.

Game G_2: (Record initialization). In this game, the simulator acts like $\mathcal{F}_{pwAuth}$ to $\mathcal{A}$ to keep the consistency of the $\mathcal{A}$ acting like a normal user. And the simulator stores the record from $\mathcal{F}_{adM3PM-EA}$ for consistency. For CompleteAuth query, the simulator check whether this query is made from $\mathcal{F}_{adM3PM-EA}$ or $\mathcal{A}$ by checking the record. All changes in this game are internal and unseeable to the environment, so it is the same with G_1.

$$\Pr[G_2] = \Pr[G_1] \tag{3}$$

Game G_3: (Random key for incorrect guess). In this game, the simulator records the password file $\langle \text{mFile}, ID_C, ID_S, x \rangle$ for the adversary. For any $(\text{OfflineTestPwd}, sid, x)$ query, the simulator looks for the record $\langle \text{mFile}, ID_C, ID_S, x \rangle$. If such a record exists, it returns Correct to $\mathcal{A}$. Otherwise, it returns Wrong. For any corrupted password file on the server, the simulator generates a random value x for the adversaries. In this game, the probability of distinguishing between G_1 and G_2 is the probability that the adversary correctly guesses x.

In our work, we assume passwords follow a Zipf distribution [54], so the probability of correctly guessing passwords is bounded by $C' \cdot (Q_{init} + Q_{retr})^{s'}$, where C' and s' are parameters of the Zipf distribution. We also consider the $SecretK$, determined by the security parameter n, as a random n-length string. The probability that the environment $\mathcal{Z}$ can distinguish between G_2 and G_3 is determined by the likelihood of simultaneously guessing the master password mpw and $SecretK$, which is negligible.

Remark 7. For other schemes, such as "hashed mpw" and "protected tokens," we need to record the queries to the random oracle and use these records to track whether the adversary tests a correct mpw. In this situation, we require the TestMpw query in $\mathcal{F}_{rM3PM}$. In other words, we can divide the functionality into two separate functionalities—an authentication functionality and a password processing functionality—and use the universal composition theorem to prove the security of their combination. For simplicity, we omit the similar details for these schemes.

$$|\Pr[G_3] - \Pr[G_2]| \leq C' \cdot (Q_{init} + Q_{retr})^{s'} \cdot \frac{1}{2^n} \tag{4}$$

Game G_4: (Unless vault key suite). In this game, the simulator replaces the encrypted secret vault key esk and vault key evk with an encryption of 0. We reduce distinguishing

Simulator SIM_{OP}

On input $(\textsc{InitC}, sid, ID_C, ID_S)$ from $\mathcal{F}_{adM3PM-EA}$:

- If there is no record $\langle \textsc{Vault}, ID_C, ID_S, *, *, * \rangle$, then generate random value $eVault, evk, esk$, store $\langle \textsc{Vault}, ID_C, ID_S, eVault, evk, esk \rangle$, and send $(\textsc{Init}, eVault, evk, esk)$ to $\mathcal{A}$
- Send $(\textsc{NewSession}, sid, ID_C, ID_S, \text{Client})$ to $\mathcal{A}$

On input $(\textsc{InitS}, sid, ID_C, ID_S)$ from $\mathcal{F}_{adM3PM-EA}$:

- If there is no record $\langle \textsc{Vault}, ID_C, ID_S, *, *, * \rangle$, then generate random vaule $eVault, evk, esk$, and store $\langle \textsc{Vault}, ID_C, ID_S, eVault, evk, esk \rangle$, and send $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ to $\mathcal{A}$

On input $(\textsc{RetrC}/\textsc{RetrS}, sid, ID_C, ID_S)$ from $\mathcal{F}_{adM3PM-EA}$:

- For $\textsc{RetrC}$, store a record $(\textsc{RetrC}, sid, ID_C, ID_S)$, and send $(\textsc{NewSession}, sid, ID_C, ID_S, \text{Client})$ to $\mathcal{A}$
- For $\textsc{RetrS}$, store an record (sid, ID_C, ID_S, res), and send $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ to $\mathcal{A}$

On input $(\textsc{StealPwdFile}, sid, ID_C)$ from $\mathcal{A}$:

- If there exists $\langle \textsc{mFile}, ID_C, ID_S, x \rangle$, then return x to the $\mathcal{A}$
- Otherwise, corrupt the server, generate a random value x' for existing ID_C and return to the $\mathcal{A}$

On input $(\textsc{NewSession}, sid, ID_C, ID_S, x, \text{Client})$ from $\mathcal{A}$:

- Store a $(ID_C, ID_S, sid, x, \text{Client})$ labeled $\textsc{Fresh}$, and send $(\textsc{NewSession}, sid, ID_C, ID_S, \text{Client})$ to $\mathcal{A}$

On input $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ from $\mathcal{A}$:

- Retrieve record $(\textsc{mFile}, ID_C, ID_S, x)$, and send $(\textsc{NewSession}, sid, ID_S, ID_C, \text{Server})$ to $\mathcal{A}$
- If a $(ID_C, ID_S, sid, x, \text{Client})$ exists, then store $(ID_S, ID_C, sid, x, \text{Server})$ labeled $\textsc{Fresh}$

On input $(\textsc{CompleteAuth}, sid, ID_S, ID_C)$ from $\mathcal{A}$:

- If there is no record $(ID_S, ID_C, sid, pw, \text{Server})$, retrieve record (sid, ID_C, ID_S, res). If it does not exist, retrieve and check $(\textsc{mFile}, ID_C, ID_S, [x])$ stored by $\mathcal{A}$
- If $res = 1$ then send $\textsc{Success}$ to ID_S and $\mathcal{A}$, else send $\textsc{Fail}$
- If $res = 1$ then retrieve $\langle \textsc{Vault}, ID_C, ID_S, eVault, evk, esk \rangle$ and send $(\textsc{Retr}, eVault, evk, esk)$ to the $\mathcal{A}$

On input $(\textsc{OfflineTestPwd}, sid, x')$ from $\mathcal{A}$:

- If there is a record $\langle \textsc{mFile}, ID_C, ID_S, x' \rangle$, then return $\textsc{Correct}$. Otherwise, return $\textsc{Wrong}$ to the $\mathcal{A}$

On input $(\textsc{Init}/\textsc{Retr}, eVault, evk, esk)$ from $\mathcal{A}$:

- Retrieve record $\langle \textsc{Vault}, ID_C, ID_S, eVault, evk, esk \rangle$, if there do not exist such a record, then abort.
- If this is a $\textsc{Retr}$ query, then send $(\textsc{CompleteRetrS}, sid, ID_C, ID_S)$

On input $(eVault', evk', esk')$ from $\mathcal{A}$:

- Update the record $\langle \textsc{Vault}, ID_C, ID_S, eVault', evk', esk' \rangle$. //The simulator maintains storage consistency.
- Retrieve $(\textsc{InitC}, sid, ID_C, ID_S)$, delete it, and send $(\textsc{CompleteInitC}, sid, ID_C, ID_S)$ to the $\mathcal{F}_{adM3PM-EA}$

Figure 9: Simulator for M3PM protocol of 1Password.

G_4 and G_3 on the IND-CPA security of authentication encryption AE and PKE. In G_3, we make sure that no adversary can guess both master password mpw and $SecretK$ correctly. Therefore, the master key mk is random and uniform for the adversary.

If there exists an adversary $\mathcal{D}$ can break IND-CPA of AE with probability $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n)$, then the environment $\mathcal{Z}$ can use the adversary $\mathcal{D}$ to distinguish G_4 and G_3 within probability $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n)$. Similarly, if there exists an adversary $\mathcal{D}'$ can break IND-CPA of PKE with probability $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D}',PKE}(n)$, then the environment $\mathcal{Z}$ can use the adversary $\mathcal{D}$ to distinguish G_4 and G_3 within probability $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},PKE}(n)$.

$$|\Pr[G_4] - \Pr[G_3]| \leq \mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n) + \mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D}',PKE}(n) \quad (5)$$

Game G_5: (Client encrypts "garbage"). In this game, the client encrypts 0 instead of password vaults. We reduce distinguishing G_5 and G_4 on the IND-CPA security of authentication encryption AE. If there exists an adversary $\mathcal{D}$ can break IND-CPA of AE with probability $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n)$, then the environment $\mathcal{Z}$ can use the adversary $\mathcal{D}$ to distinguish G_3 and G_2 within probability $Q_{init}\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n)$ where Q_{init} is the times of initialization.

$$|\Pr[G_5] - \Pr[G_4]| \leq Q_{init} \cdot \mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{D},AE}(n) \quad (6)$$

Game G_6: (Abort upon ciphertext forgery). In this game, the simulator keep a local storage for the consistency of $eVault, evk, esk$. Upon receiving $\textsc{InitC}$ query, the simulator keep the storage $(\textsc{Vault}, ID_C, ID_S, eVault, evk, esk)$. Therefore, the adversary cannot distinguish the game from the inconsistency of the encrypted storage or keys. If the simulator receives a new ciphertext created by adversary $\mathcal{A}$, then the simulator returns FAIL to the adversary.

The environment $\mathcal{Z}$ distinguishes G_5 and G_6 if adversary $\mathcal{A}$ forges a ciphertext succeeding to decrypt by AE. Then, we reduce the distinguishability to the ciphertext integrity (INT-CTXT) of AE. If there exists an adversary $\mathcal{D}$ can break INT-CTXT of AE with probability $\mathsf{Adv}_{\mathcal{D},AE}^{\mathrm{int-ctxt}}(n)$, then the environment $\mathcal{Z}$ can use the adversary $\mathcal{D}$ to distinguish G_6 and G_5 within probability $(Q_{init} + Q_{retr}) \cdot \mathsf{Adv}_{\mathcal{D},AE}^{\mathrm{int-ctxt}}(n)$ where Q_{init} and Q_{retr} are the times of initialization and retrieval.

$$|\Pr[G_6] - \Pr[G_5]| \leq (Q_{init} + Q_{retr}) \cdot \mathsf{Adv}_{\mathcal{D},AE}^{\mathrm{int-ctxt}}(n) \quad (7)$$

Game G_7: (Use $\mathcal{F}_{M3PM-EA}$ functionality interface). In this game, we modify the simulator to interact with the functionality $\mathcal{F}_{M3PM}$ through the required interface. This change cannot be distinguished by the environment $\mathcal{Z}$, as it is internal. The final simulator is described in Fig. 9, and we use appropriate formal descriptions to balance accuracy and understandability.

$$\Pr[G_7] = \Pr[G_6] \quad (8)$$

In the game G_7, the simulator mimics ideal world perfectly, so we have:

$$Ideal_{\mathcal{Z},SIM_{OP},\mathcal{F}_{M3PM-EA}} = \Pr[G_7] \quad (9)$$

In conclusion, we find that the advantage of environment $\mathcal{Z}$ in distinguishing between the real world and the ideal world is negligible.

□

G. Proof of SPHINX

Proof. Let $Real_{\mathcal{Z},\mathcal{A},SPHINX}$ be the probability that the environment $\mathcal{Z}$ interacting with adversary $\mathcal{A}$ and the real-world protocol $SPHINX$ outputs 1, and let $Ideal_{\mathcal{Z},SIM_{SPHINX},\mathcal{F}_{rM3PM-IA}}$ be the probability that the environment $\mathcal{Z}$ interacting with simulator SIM_{SPHINX} and the ideal-world functionality $\mathcal{F}_{rM3PM-IA}$ outputs 1. We define a series of games from the real-world execution to the ideal-world execution and prove that the environment $\mathcal{Z}$ cannot distinguish between the real and ideal worlds.

Game G_0: This game is in the real world, and the adversary $\mathcal{A}$ interacts with the real players following the protocol.

$$Real_{\mathcal{Z},\mathcal{A},SPHINX} = \Pr[G_0] \quad (10)$$

Game G_1: (Simulate real world). In this game, we perform simulation on the real players to make the adversary $\mathcal{A}$ and environment $\mathcal{Z}$ interact with the simulator. The simulation in this game is exactly the same as the real game, except the simulator stores records to simulate random oracles and all transcripts between the client ID_C and the server ID_S. In all, this change is just for turning the real game into a game in the UC framework with only syntactical change.

$$\Pr[G_1] = \Pr[G_0] \quad (11)$$

Game G_2: (Avoid nonce collision). In this game, the simulator avoids the collision of random nonce. The simulator tracks the nonce α and k, and when a new α', k' is the same as an old nonce α, k, the simulator aborts. The nonce k is only generated in the initialization phase with Q_{init} times, and α is generated in both the initialization phase with Q_{init} times and retrieval phase with Q_{retr} times.

$$|\Pr[2] - \Pr[1]| \neq \binom{Q_{init}}{2} \cdot 2^{-n} + \binom{Q_{retr}}{2} \cdot 2^{-n+1} \quad (12)$$

Game G_3: (Random generated password for passive adversary). In this game, if $\mathcal{A}$ sends an α generated by the simulator, the simulator outputs a random password when the adversary queries H_2. The environment $\mathcal{Z}$ can distinguish between G_3 and G_2 only if $\mathcal{A}$ can calculate ζ using only α and β. Without the random value ρ, ζ is independent of α and β.

$$\Pr[G_3] = \Pr[G_2] \quad (13)$$

Game G_4: (Random generated password if adversary guesses wrongly). In this game, the simulator tracks the random oracles H_1 and H_2. For each H_1 query by $\mathcal{A}$, the simulator asks $\mathcal{F}_{rM3PM-IA}$ to test whether the master password is correct and records the answer. When $\mathcal{A}$ queries the H_2 oracle, the simulator checks whether $\mathcal{A}$ used a correctly guessed master password. If so, the simulator queries $\mathcal{F}_{rM3PM-IA}$ for the correct password; otherwise, it returns a random password to $\mathcal{A}$. The environment $\mathcal{Z}$ can distinguish between G_3 and G_4 only if $\mathcal{A}$ can generate a new ζ without sending an α.

We reduce distinguishing G_3 and G_4 to the one-more gap DH assumption. In the reduction, the simulator gets a random element set R containing $g_1, ..., g_{Q_{retr}}$ and returns g_i for each H_1 query of $\mathcal{A}$. The adversary $\mathcal{A}$ interacts Q_{retr} times with the server ID_S using DH_k queries which calculate $\zeta = g_i^k$. For each H_2 query with $(mpw, login, \zeta)$, the simulator first checks whether $\mathcal{A}$ has queried H_1 with $(mpw, login)$ and get the answer g_i. If so, the simulator uses DDH_k to check whether $\zeta = g_i^k$, and outputs a pair (g_i, ζ) if it is. Therefore, if the adversary $\mathcal{A}$ can generate a new ζ without using a DH_k oracle, then the simulator can output $Q_{retr} + 1$ pair of (g_i, g_i^k).

$$|\Pr[G_4] - \Pr[G_3]| \leq \mathsf{Adv}_{\mathcal{A},Q_{init},Q_{RO}}^{\mathrm{omg-dh}}(n) \quad (14)$$

Game G_5: (Random α). In this game, the simulator sends adversary $\mathcal{A}$ a random value instead of a result of $(H_1(mpw, login))^\rho$. Because ρ is uniformly random sampled, the α is a one-time pad of the $H_1(mpw, login)$. Therefore, a random value in $\mathbb{G}$ is information-theoretically indistinguishable from the correct α.

$$\Pr[G_5] = \Pr[G_4] \quad (15)$$

Simulator SIM_{SPHINX}

On input $(\textsc{InitC}, sid, ID_C, ID_S)$ from $\mathcal{F}_{rM3PM-IA}$:

- Store a record $(\textsc{InitC}, sid, ID_C, ID_S)$, and send $\alpha \leftarrow\!\$\, \mathbb{G}$ to $\mathcal{A}$.

On input $(\textsc{RetrC}, sid, ID_C, ID_S)$ from $\mathcal{F}_{rM3PM-IA}$:

- Store a record $(\textsc{RetrC}, sid, ID_C, ID_S)$, and send $\alpha \leftarrow\!\$\, \mathbb{G}$ to $\mathcal{A}$.

On input β from $\mathcal{A}$:

- Do nothing. //The malicious server outputs nothing, so the simulator does not respond.

On input α from $\mathcal{A}$:

- Retrieve $\langle \textsc{Vault}, ID_C, ID_S, k\rangle$. If no record exists, create one with a random $k \leftarrow\!\$\, \{0,1\}^n$.
- Send $\beta = \alpha^k$ to $\mathcal{A}$. //The simulator only maintains the storage consistency for a malicious client.

On input $H_1(mpw, login)$ from $\mathcal{A}$:

- Perform like a random oracle against $\mathcal{A}$.
- If server ID_S is corrupted, then send $(\textsc{TestMpw}, ID_C, ID_S, mpw, login)$ to $\mathcal{F}_{rM3PM-IA}$ and get the (r, pwd). Then store a record $\langle \textsc{OfflineGuess}, sid, ID_C, ID_S, mpw, login, pwd\rangle$.
- Otherwise, send $(\textsc{RetrC}, sid, ID_C, ID_S, mpw, login)$ to the $\mathcal{F}_{M3PM}$, and get the response. Then store a record $\langle \textsc{Guess}, sid, ID_C, ID_S, mpw, login\rangle$.

On input $H_2(mpw, login, \zeta)$ from $\mathcal{A}$:

- If there exists such a record $\langle \textsc{OfflineGuess}, sid, ID_C, ID_S, mpw, login, [pwd]\rangle$, then send pwd to the $\mathcal{A}$
- Otherwise, retrieve a record $\langle \textsc{Guess}, sid, ID_C, ID_S, mpw, login\rangle$. If there does not exist such a record, then send $\mathcal{A}$ a random value and record the value like a random oracle.
- Send $\mathcal{F}_{rM3PM-IA}$ a query $(\textsc{CompleteRetrC}, sid, ID_C, ID_S)$ and get the output pwd. Then send $\mathcal{A}$ the pwd.

Figure 10: Simulator for M3PM protocol of SPHINX.

Game G_6: (Use $\mathcal{F}_{rM3PM-IA}$ functionality interface). In this game, we modify the simulator to interact with the functionality $\mathcal{F}_{rM3PM-IA}$ in the required interface. This change cannot be distinguished by environment $\mathcal{Z}$, for the change is internal. The final simulator is described in Fig. 10.

$$\Pr[G_6] = \Pr[G_5] \tag{16}$$

In the game G_6, the simulator mimics an ideal world with functionality perfectly, so we have:

$$Ideal_{\mathcal{Z}, SIM_{SPHINX}, \mathcal{F}_{rM3PM-IA}} = \Pr[G_6] \tag{17}$$

Finally, we conclude that the advantage of environment $\mathcal{Z}$ for distinguishing the real world and the ideal world is negligible.

$\square$