Breach Extraction Attacks: Exposing and Addressing the Leakage in Second Generation Compromised Credential Checking Services*

Dario Pasquini† SPRING lab, EPFL dario.pasquini@epfl.ch

Danilo Francati† Aarhus University dfrancati@cs.au.dk

Giuseppe Ateniese George Mason University ateniese@gmu.edu

Evgenios M. Kornaropoulos George Mason University evgenios@gmu.edu

***Abstract*—Credential tweaking attacks use breached passwords to generate semantically similar passwords and gain access to victims' services. These attacks sidestep the first generation of compromised credential checking (C3) services. The second generation of compromised credential checking services, called "Might I Get Pwned" (MIGP), is a privacy-preserving protocol that defends against credential tweaking attacks by allowing clients to query whether a password or a semantically similar variation is present in the server's compromised credentials dataset. The desired privacy requirements include not revealing the user's entered password to the server and ensuring that no compromised credentials are disclosed to the client.**

In this work, we formalize the cryptographic leakage of the MIGP protocol and perform a security analysis to assess its impact on the credentials held by the server. We focus on how this leakage aids breach extraction attacks, where an honest-but-curious client interacts with the server to extract information about the stored credentials. Furthermore, we discover additional leakage that arises from the implementation of Cloudflare's deployment of MIGP. We evaluate how the discovered leakage affects the guessing capability of an attacker in relation to breach extraction attacks. Finally, we propose MIGP 2.0, a new iteration of the MIGP protocol designed to minimize data leakage and prevent the introduced attacks.

1. Introduction

In the evolving cyber threat landscape, attackers target user credentials, particularly those stored in plaintext, exploiting system vulnerabilities to compromise and post them online, thereby breaching user privacy and enabling *credential stuffing attacks* [1]. In these attacks, adversaries exploit widespread password reuse [2], [3], [4], [5] by using credentials exposed from a data breach to attempt unauthorized access to another unrelated domain. Services like "Have I Been Pwned" [6], Google Password Checkup [7], and Microsoft Password Monitor [8]—known as Compromised Credential Checking (C3) services— aim to alert users about the possibility of a credential stuffing attack. Specifically, they allow users to check if their active credentials appear in breach datasets. To accomplish this, C3 services use cryptographic tools to create a privacy-preserving protocol, ensuring that the queried password of the user (which may not be breached) is not disclosed to the server and the sensitive breached credentials are not shared with the client.

However, these services cannot cover an increasingly common type of attack: *credential tweaking attacks* [2], [9], [10]. In these attacks, cybercriminals employ sophisticated techniques to generate slight variants of known breached passwords, enabling them to make distinct educated password guesses towards unauthorized access to the target's services. Unfortunately, credential tweaking attacks are not covered by C3 services since they only check for *an exact match* against the breached credentials. To address these shortcomings, Pal et al. [11] proposed *Might I Get Pwned* (MIGP), a second-generation C3 service. MIGP extends the capabilities of conventional C3 by checking not only for exact password matches but also for semantic similarity with breached credentials. To achieve this, MIGP uses a password-generating function called τ to generate semantically similar passwords during the initialization phase.

The Role of Cryptographic Leakage in MIGP. It is important to note that the privacy-preserving design of MIGP serves (in part) the purpose of safeguarding the collection of breached credential data from being exposed to MIGP query issuers. Paradoxically, despite the MIGP server's data collection being labeled as "breached credentials," it can contain credentials that have been *breached but are not publicly available*. In April 2023 [12], the FBI took down a stolen identity marketplace that was selling non-publicly available breached credentials. To combat credential stuffing attacks, the FBI shared in confidence millions of non-publicly available compromised credentials with HIBP. Thus, real-world C3 services work with breached credentials that should not be exposed under any circumstances.[1]

Our work sheds light on an unexplored aspect of the MIGP protocol: the existence of *cryptographic leakage* over the stored credentials. This leakage is a controlled disclosure intentionally designed into the protocol. The term

**Appearing in the proceedings of the 45nd IEEE Symposium on Security and Privacy (S&P'24).*

†*Equal contribution.*

1. We note that if the breached credentials of the server are all considered public, then there is no point in deploying a privacy-preserving C3; the server can simply return a subset (i.e., a bucket) of the credentials in plaintext. This change enhances efficiency by forgoing cryptographic operations for non-interactive queries. Additionally, it fortifies defenses against tweaking attacks, allowing users to apply arbitrary similarity functions to leaked passwords.

breach extraction attack describes the attack vector in which an honest-but-curious user interacts with the MIGP server through its protocols and uses the leakage to infer the breach credentials stored within it. In this work, we focus on examining how MIGP's leakage impacts the security of the breached credentials stored in the MIGP server.

On Cryptographic Leakage in Encrypted Systems. In order for the MIGP protocol to scale to the necessary performance requirements, i.e., resolving privacy-preserving queries on datasets with billions of breached credentials, the designers allowed some information to be revealed by design. This is not the first instance where a cryptographic primitive has allowed controlled disclosure; well-known examples include Searchable Encryption [13] and Structured Encryption [14]. Our work is inspired by the cryptanalysis of the leakage in searchable encryption [15], [16], [17], [18], [19], [20], where leakage attacks are mounted by the server to reconstruct the client's data through leakage. A promising step forward in this area is the use of a privacy quantification technique called *leakage inversion* [21] that has the ability to formally capture how leakage affects the reconstruction ability of an attacker. However, in our work, the leakage attacks are mounted by the client to extract information about the credentials stored on the server.

Where Does the Leakage Come From? Interestingly, the password-generating function τ can output the same password from different inputs. We call this phenomenon a τ-collision. Unfortunately, our findings show that the design of MIGP [11] reveals to the client several types of τ-collisions through the initialization of the protocol. When multiple τ-collisions occur, they reveal a network of connections between passwords, enabling the attacker to significantly reduce the search space. Simply switching to a different τ function is not a viable solution. The reason is that τ's role is to accurately predict password variations that a user would come up with; however, the more accurate the τ function becomes, the higher the occurrence of τ-collisions, leading to increased leakage in the protocol.

More Leakage from Cloudflare's Implementation. MIGP is currently deployed by Cloudflare and offers (1) a public-facing API similar to HIBP, as well as (2) a new breach alerting feature within Cloudflare's Web Application Firewall (WAF) product.[2] WAF acts on behalf of a client and communicates with the MIGP server to check if the queried credential (or a similar counterpart) has been compromised. However, our research shows that the current implementation of MIGP by Cloudflare [22] causes additional leakage, which assists in breach extraction attacks. We clarify that breach extraction attacks concern the data held by the MIGP server and not the query issued by the MIGP client.

Our Contributions. Our work advances the understanding of Compromised Credential Checking (C3) services by uncovering leakage-based vulnerabilities both within the MIGP protocol and in its implementation [22]:

- **Formalizing Leakage in MIGP:** We identified an unexplored cryptographic leakage over stored credentials in the MIGP protocol. This leakage, amplified by specific decisions in Cloudflare's MIGP implementation, reveals significant structural information about the stored credentials (Section 3 and Section 5).
- **Examination of τ-collisions:** We provide a taxonomy of τ-collisions (events where the password-generating function τ generates the same result from distinct inputs) and the means to identify them from the client's view of MIGP. The observed τ-collisions serve as the core ingredient towards breach extraction attacks.
- **Breach Extraction Attacks via Leakage:** We present three approaches that exploit the observed leakage to infer information about the breached credentials of the server. The first attack (Section 4) assumes knowledge of all but one password of the target user and uses cryptographic leakage to narrow down the possible guesses based on the observed pattern. The second attack (Section 6) assumes knowledge of a single password of the target user and leverages both cryptographic leakage and implementation leakage to guess the remaining passwords of the target user. The third attack (Section 7) assumes no knowledge about the target's passwords and uses the cryptographic leakage and the implementation leakage to infer a template of the unknown passwords, e.g., consisting of numbers.
- **Proposal of MIGP 2.0:** We propose MIGP 2.0 as a response to the identified security issues in the original MIGP protocol. This new protocol minimizes leakage and eliminates the risk of τ-collisions (Section 8).

Vulnerability Disclosure. We disclosed our findings about the leakage (see Sections 3 and 5) and our techniques for using the leakage toward breach extraction attacks (see Sections 4, 6, and 7) to the Cloudflare team on May 24th, 2023. Cloudflare acknowledged receipt the next day. On August 2nd, the company confirmed the validity of our findings. The Cloudflare team informed us that the public-facing API of the breach-extraction-prone implementation [22] is considered a demo, and there is no threat with respect to the WAF as it only acts as the client part of the protocol and, thus, does not leak any information. We, therefore, consider the presented attacks orthogonal to Cloudflare's WAF.

Ethical Considerations The MIGP server deployed by CloudFlare uses only publicly known breached credentials from the 2017 "BreachCompilation" dataset (4IQ); thus, in this instance, our extraction techniques cannot expose private data due to its absence. To carry out our security analysis, we used plaintext credentials from the 4IQ leak, raising ethical concerns. Yet, our analysis does not further the damage initially inflicted on users by the 4IQ leak. We refrained from linking the 4IQ data with other publicly available information or preprocessing it in a manner that could offer more insights to potential adversaries.

2. Overview of the MIGP Protocol

The following exposition describes the "Might I get Pwned" (MIGP) protocol that was originally proposed in [11]. This protocol is run between a client and a server.

2. A running demo can be found at: https://migp.cloudflare.com.

The desired functionality is that the client wants to check if (i) her input password currently appears in a breached dataset, (ii) a password that is "similar" to her original password appears in a breached dataset, or (iii) none of the above. The server holds a collection of breached credentials (augmented by tweaked passwords) and assists in this computation by returning the client an encoded subset of the password collection. The desired privacy guarantees are (A) the server does not infer the queried credential, and (B) the client does not infer the encoded credentials (which may not be publicly available) received from the server. Jumping ahead, the next section describes the security-efficiency trade-off as *leakage functions* in order to make the system scalable.

Notation. The term $\mathbb{G}$ denotes a cyclic group of prime order, and $\mathbb{Z}_p$ is the set of integers modulo p. The notation $x \leftarrow\$ \mathcal{X}$ denotes sampling at random from the set $\mathcal{X}$. The term $|\mathcal{X}|$ denotes the cardinality of the set $\mathcal{X}$. We define as $\mathcal{D} = \{(u_1, \tilde{w}_1), \ldots, (u_{|\mathcal{D}|}, \tilde{w}_{|\mathcal{D}|})\}$ a dataset of breached username-password pairs (i.e., credentials). The term $\mathcal{Z}$ denotes a set (i.e., bucket) computed by the server. The term τ_n defines an algorithm with input a user's password (also called an original/real password) and output n synthetic passwords that are *similar* to the input (under a notion of similarity). On input the i-th password $\tilde{w}_i$ of the user u, $\tau_n(\tilde{w}_i)$ outputs n passwords $(w_i^1, \ldots, w_i^n)$ that are similar to the real password $\tilde{w}_i$. For several users, say u_1, u_2, then the notation for the i-th password of u_1 and the j-th password of u_2 becomes $\tilde{w}_{1,i}$ where $\tau_n(\tilde{w}_{1,i})$ outputs $(w_{1,i}^1, \ldots, w_{1,i}^n)$ and for $\tilde{w}_{2,j}$ we have $\tau_n(\tilde{w}_{2,j})$ that outputs $(w_{2,j}^1, \ldots, w_{2,j}^n)$.

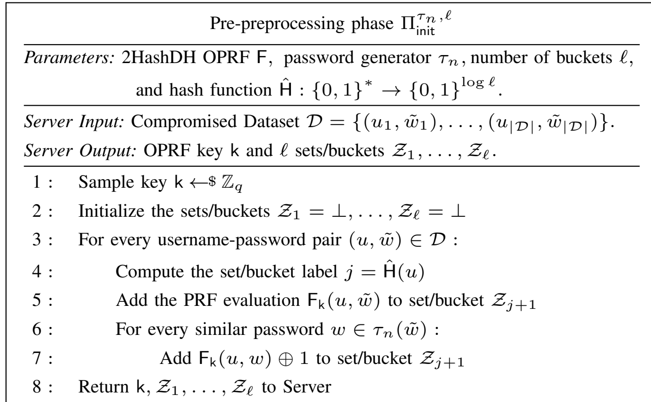


Figure 1: Pre-processing phase of MIGP. The OPRF F corresponds to the 2HashDH construction. τ_n corresponds to an arbitrary function that, on input an original password $\tilde{w}$, it generates n similar passwords $(w_1, \ldots, w_n)$.

2HashDH OPRF of Jarecki et al. [23], [24]. A PRF F is an efficiently computable function that produces pseudorandom outputs. Let $\mathbb{G}$ be a cyclic group of prime order p and $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}, \mathsf{H}_2 : \{0,1\}^* \times \mathbb{G} \rightarrow \{0,1\}^\lambda$ be two hash functions. An *oblivious* PRF (OPRF) F is a PRF with an efficient protocol Π_{oprf} which allows a client (holding an input x) and a server (holding a key k) to compute $\mathsf{F}_\mathsf{k}(x)$ without revealing any information about the input of the other party. The 2HashDH OPRF [23], [24] construction $\mathsf{F} : \mathbb{Z}_p \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is defined as $\mathsf{F}_\mathsf{k}(x) = \mathsf{H}_2(x, \mathsf{H}_1(x)^\mathsf{k})$ where $x \in \{0,1\}^*$ and k is a random PRF key $k \leftarrow\$ \mathbb{Z}_p$. The protocol Π_{oprf} is defined in Figure 13 (see Section A). We abstract OPRF calls as invocations of the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$.

Pre-processing Phase $\Pi_{\mathsf{init}}^{\tau_n,\ell}$ (Server-side Only). Let ℓ be an integer denoting the number of buckets and τ_n be a fixed and publicly known algorithm that computes similar passwords. The MIGP protocol starts with an offline pre-processing phase $\Pi_{\mathsf{init}}^{\tau_n,\ell}$ in which the server process the dataset $\mathcal{D} = \{(u_1, \tilde{w}_1), \ldots, (u_{|\mathcal{D}|}, \tilde{w}_{|\mathcal{D}|})\}$ composed of breached username and password pairs. The protocol $\Pi_{\mathsf{init}}^{\tau_n,\ell}$ is depicted in Figure 1. First, the server samples a random PRF key $\mathsf{k} \leftarrow\$ \mathbb{Z}_p$ for the OPRF F and initializes ℓ empty buckets $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$, implemented as sets (i.e., no repetitions). Then, for every pair $(u, \tilde{w}) \in \mathcal{D}$, it encodes $(u, \tilde{w})$ by evaluating $\mathsf{F}_\mathsf{k}(u, \tilde{w})$ and it adds the latter to the $(j+1)$-th bucket $\mathcal{Z}_{j+1}$ selected as $j = \hat{\mathsf{H}}(u)$ where $\hat{\mathsf{H}} : \{0,1\}^* \rightarrow \{0,1\}^{\log \ell}$ is an hash function. We note here that buckets $\mathcal{Z}$ are indexed from 1 to ℓ, while the hash function outputs bit labels from $\{0,1\}^{\log \ell}$, which is why we assign each user to the $(j+1)$-th bucket (as opposed to j-th). In order to deal with similar passwords, the server holds a password generator algorithm $\tau_n(\cdot)$ (we assume $\tau_n(\cdot)$ is a publicly known parameter of the protocol) such that on input, a real password $\tilde{w}$, it generated n synthetic variants $(w_1, \ldots, w_n) = \tau_n(\tilde{w})$ of $\tilde{w}$. This algorithm is applied to every original password $\tilde{w}$ contained in $\mathcal{D}$ in order to generate variants that, in turn, will be encoded as described before, except that *the last bit of the PRF's output is flipped* in order to denote that this is a similar password (see Line 7 of Figure 1). Then, the server stores the PRF key k and the pre-processed buckets $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$.

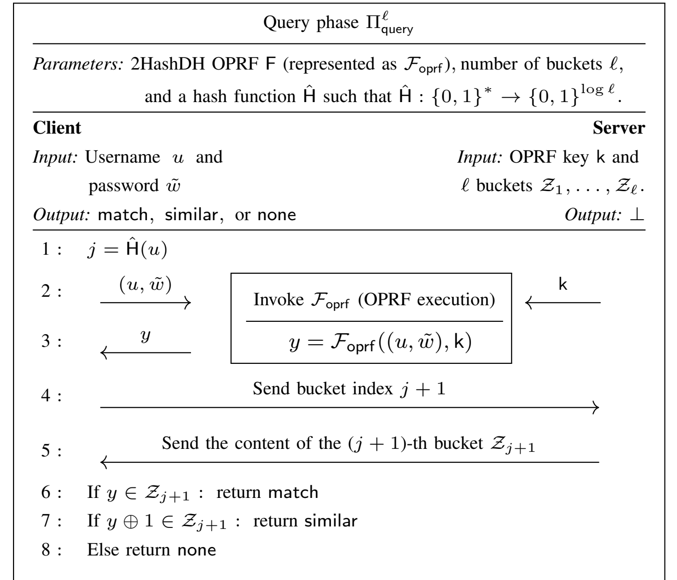


Figure 2: The query phase of MIGP. $\mathcal{F}_{\mathsf{oprf}}$ is the ideal OPRF functionality that is instantiated with 2HashDH.

Query Phase $\Pi_{\mathsf{query}}^{\ell}$. The query phase is executed each time a client wants to check whether her password w (associ-

ated to an username u) appears in the pre-processed server's buckets. The client and the server invoke the ideal OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$ on client's input $(u, \tilde{w})$ and server's input k (this is denoted as $y = \mathcal{F}_{\mathsf{oprf}}((u, \tilde{w}), \mathsf{k})$ in Figure 2). As a result, the client will obtain $y = \mathsf{F}_{\mathsf{k}}(u, \tilde{w})$. Moreover, the client sends the bucket index $j+1$ corresponding to her username u, i.e., $j = \hat{\mathsf{H}}(u)$. In turn, the server will reply with the entire $(j+1)$-th bucket $\mathcal{Z}_{j+1}$. With $y = \mathsf{F}_{\mathsf{k}}(u, \tilde{w})$ and $\mathcal{Z}_{j+1}$, the client can locally check if $y \in \mathcal{Z}_{j+1}$. If this is the case, the client outputs match, meaning that her password $\tilde{w}$ appears in the password dataset of the server. On the other hand, if $y \notin \mathcal{Z}_{j+1}$, the client checks if $\tilde{w}$ matches a similar password w_i generated by the server using her password generator function τ_n. This can be done by checking that $y \oplus 1 \in \mathcal{Z}_{j+1}$ (recall that similar passwords are encoded with the last bit flipped). If true, then the client outputs similar. If all the above checks fail, then the client outputs none, meaning that $\tilde{w}$ does not match any of the original and similar passwords. We simplified [11] by excluding client-side generation of similar passwords (see Section B).

Similarity Function τ. To model similarity among passwords, `MIGP` relies on models originally designed to perform credential *tweaking* [2], [9]. These generative models take a password as an input and produce a set of similar passwords that are variations that the user may come up with (e.g., "*pass0word*" $\rightarrow$ "*Pass0word!*"). Formally, function $\tau_n(\tilde{w})$ generates a set of n similar passwords to $\tilde{w}$, with $w \notin \tau_n(w)$. We note that τ is not necessarily symmetric, that is if $w_1 \in \tau_n(w_2)$ it doesn't mean that $w_2 \in \tau_n(w_1)$. Pal et al. [11] considers two similarity functions.

`Das-r` model uses a set of hand-crafted mangling rules introduced by Das et al. [2]. The above rules encode basic string transformations based on deleting, appending, and substituting characters. Pal et al. [11] choose to rank the output of the mangling rules with respect to their effectiveness, an approach denoted as "*Das-r*". For reference, in Table 2, we report the first 10 entries. `Das-r` model is fast and accurate enough [11], which is why `MIGP` chooses `Das-r` as its τ function in the main implementation [22].

`P2P` model uses a neural network [9]. Specifically, an encoder-decoder RNN-based architecture is employed to model credential tweaking attacks. Given a target password as input, the network conditionally defines a set of transformations by composing over a vocabulary of predefined atomic instructions [9]. While this model generally performs better than simpler rule-based approaches such as `Das-r`, this is currently not employed by any of the `MIGP` implementations, given its inherent computational complexity.

3. The Cryptographic Leakage Profile

The `MIGP` protocol is designed to be efficient in practice. This efficiency comes with the price of weakening the overall security; this tradeoff is captured by a *leakage function*. The security definition guarantees that nothing else is revealed to the participating parties except these well-defined leakage functions (also called *leakage profile*). We note that the original `MIGP` protocol [11] was not presented using a leakage-based security definition (but previous works on compromised credentials checking services did follow this security framework [25]). In this section, we *refactor* the same security guarantees of the original `MIGP` under a leakage-based framework so as to (i) formally identify the shortcomings of the proposed leakage profile [11], and (ii) use our cryptanalysis to motivate a new leakage profile (and a protocol) that is not vulnerable to our attacks.

The `MIGP` Leakage Profile. We say that the `MIGP` protocol is $(\mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}})$-secure if the server and the client (holding inputs $\mathcal{D}$ and $(u, \tilde{w})$, respectively) obtain no more information than the leakage $\mathsf{L}_{\mathsf{client}}(\mathcal{D}, (u, \tilde{w}))$ (i.e., a disclosure function over the server's dataset $\mathcal{D}$) and $\mathsf{L}_{\mathsf{server}}(\mathcal{D}, (u, \tilde{w}))$ (i.e., a disclosure function over the client's queried credentials $(u, \tilde{w})$), respectively. This is formalized by requiring that there exists an efficient simulator that simulates the protocol by using only the corresponding leakage. To capture a more realistic interaction with `MIGP`, the simulator must be able to simulate q rounds of query phase executions, i.e., $(u_1, \tilde{w}_1), \ldots, (u_q, \tilde{w}_q)$. Hence, in the case of multiple queries, we say that the `MIGP` protocol is $(q, \mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}})$-secure where the leakage is a function computed over $\mathcal{D}$ and $\{(u_i, \tilde{w}_i)\}_{i \in [q]}$ (i.e., the q inputs of the client), i.e., $\mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i \in [q]})$ and $\mathsf{L}_{\mathsf{server}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i \in [q]})$. The discussed security definition (the formal treatment of which can be found in Section C) captures the case of semi-honest security.

A first look at the design of `MIGP` (ignoring the client-side similar password generation) may imply that the best possible security that *any*[3] second-generation C3 service can offer is defined by the following two leakage functions:

$$\begin{aligned} \mathsf{L}_{\mathsf{server}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i \in [q]}) &= (\hat{\mathsf{H}}(u_i) + 1)_{i \in [q]} \\ \mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i \in [q]}) &= (\mathsf{result}_i, |\mathcal{Z}_{\hat{\mathsf{H}}(u_i)+1}|)_{i \in [q]} \end{aligned} \quad (1)$$

where $\mathcal{D}$ is the server's dataset, $(u_i, \tilde{w}_i)$ is the i-th credential queried by the client, $\hat{\mathsf{H}}(u_i) + 1$ is the bucket index queried by the client during the i-th execution, $|\mathcal{Z}_{\hat{\mathsf{H}}(u_i)+1}|$ is the size of the bucket returned by the server during the i-th execution, and result_i is the final output of the i-th query phase execution which is defined as follows:

$$\mathsf{result}_i = \begin{cases} \mathsf{match} & \text{, if } (u_i, \tilde{w}_i) \in \mathcal{D} \\ \mathsf{similar} & , \exists (u_i, \tilde{w}_j) \in \mathcal{D} : \tilde{w}_i \in \tau_n(\tilde{w}_j) \\ \mathsf{match} + \mathsf{similar} & \text{, if } (u_i, \tilde{w}_i) \in \mathcal{D} \text{ and} \\ & \quad \exists (u_i, \tilde{w}_j) \in \mathcal{D} : \tilde{w}_i \in \tau_n(\tilde{w}_j) \\ \mathsf{none} & \text{, otherwise} \end{cases} \quad (2)$$

We emphasize that the result "match + similar" is not explicitly mentioned in [11] as part of their desired functionality. However, it can be inferred from the design choices of their protocol, as the same password w' may appear as both $\mathsf{F}_{\mathsf{k}}(u, w')$ and $\mathsf{F}_{\mathsf{k}}(u, w') \oplus 1$ in the same bucket.

τ-Collisions: A Subtle Disclosure of `MIGP`. The `MIGP` protocol models each bucket as a *set*. It is known that every member of a set must be unique, i.e., no two members are

3. In Section 4 we show that the leakage profile from Equation (1) is vulnerable to attacks and in Section 8 we show that there exists a leakage profile that reveals significantly less and is not vulnerable to our attacks.

identical. We stress that this modeling is a design choice, not an inherent characteristic of the functionality.

One implication of this design choice is the following: inserting a duplicate PRF evaluation in bucket $\mathcal{Z}$ leads to creating a "hole"[4]. Specifically, it is possible that Line 7 of Figure 1 attempts to insert the same synthetic password multiple times to $\mathcal{Z}$. Thus, a curious client can detect the absence of a password, given that both τ and n are public. With the term *τ-collision*, we refer to the phenomenon of τ generating the same password for two different inputs. The next subsection details the different ways that τ-collisions may occur. A τ-collision is illustrated in Figure 3 for passwords "*mkm123*" and "*123123001*". Applying the τ function to both passwords generates the same entry "*123123*" once for τ("*mkm123*") and once for τ("*123123001*").

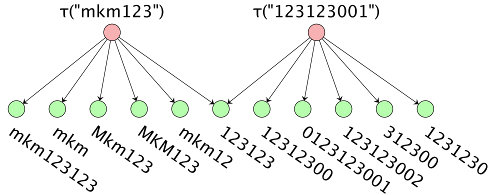


Figure 3: Example of a τ-collision between two passwords. Function τ generates the same output, namely `123123`, for two different inputs, namely `mkm123` and `123123001`.

When Do We Have τ-collisions? A τ-collision between two original passwords occurs when the passwords are semantically similar. Informally, the very nature of the τ function is to enable the arrow direction "*semantically similar passwords*"$\Rightarrow$"*collision*". To better understand the implications of this leakage, we need to analyze whether an attacker can traverse this relation in the reverse direction, i.e., "*semantically similar passwords*"$\Leftarrow$"*collision*", i.e., recover one original password by observing a collision.

3.1. Taxonomy of τ-collisions

In our analysis, we identify three types of τ-collisions. Each of them can be detected using a different mechanism. Hereafter, we refer to them as *Type-0*, *Type-1a*, and *Type-1b*.

Type-0 Collisions. Type-0 collisions do not appear very often (according to our experiments), but they disclose a lot of information about the structure of colliding passwords. On a high level, this collision inserts the same password w twice in the bucket without leaving a "hole". A Type-0 collision takes place when a user has at least one real password $\tilde{w}_i$ that is identical to a password generated by τ with $\tilde{w}_j$ as an input. This can be formally expressed as:

$$\exists (u, \tilde{w}_i), (u, \tilde{w}_j) \in \mathcal{D} : \tilde{w}_i \in \tau(\tilde{w}_j). \quad (3)$$

Collision Detection Mechanism. Type-0 collisions are revealed in an explicit way, i.e., $\mathcal{Z}$ has a Type-0 collision, if it contains two PRF outputs with the last bit flipped:

$$\exists \mathsf{F}_\mathsf{k}(u, \tilde{w}_i), \mathsf{F}_\mathsf{k}(u, w_j^t) \in \mathcal{Z} : \mathsf{F}_\mathsf{k}(u, \tilde{w}_i) = (\mathsf{F}_\mathsf{k}(u, w_j^t) \oplus 1).$$

4. Let q be the number of real passwords in $\mathcal{Z}$. With the term "hole" we refer to the fact that there is a discrepancy between $|\mathcal{Z}|$ and $q \cdot n$. We do not imply that the memory allocation of the bucket is fragmented.

Indeed, following Line 7 of the protocol Figure 1, every similar password generated through τ is flagged by `XOR`ing the last bit with 1 before inserting it in the bucket. Given that the two PRF outputs are not identical (i.e., the last bit is different), this "repetition" is not eliminated (see Figure 4a).

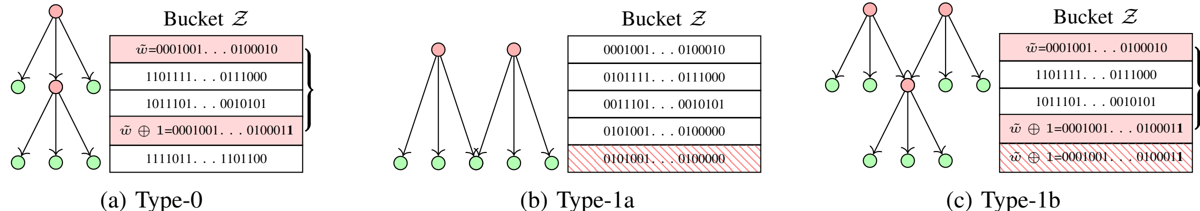
Type-1 Collisions. Type-1 collisions are the most prevalent form of τ-collisions. We differentiate between two types of Type-1 collisions, i.e., Type-1a and Type-1b. On a high level, both types of collisions leave an "hole" in the bucket due to inserting the same PRF evaluation twice into a set. Type-1a collision occurs when (at least) two passwords associated with the same user generate the same similar password through the application of τ. Formally, we have a Type-1a τ-collision in a bucket $\mathcal{Z}$ given a user u when:

$$\exists (u, \tilde{w}_i), (u, \tilde{w}_j) \in \mathcal{D} : |\tau(\tilde{w}_i) \cap \tau(\tilde{w}_j)| \neq 0. \quad (4)$$

An illustration of Type-1a is in Figure 4b. On the other hand, Type-1b collisions occur when two passwords associated with the same user generate the same similar password w' through the application of τ, and additionally, this password w' also appears as a real password, see Figure 4c.

Collision Detection Mechanism. In the case of a Type-1 collision, the initialization of the bucket in Figure 1 results in the existence of at least two different PRF evaluations of synthetic passwords $w_i^k \in \tau(\tilde{w}_i)$ and $w_j^t \in \tau(\tilde{w}_j)$, such that $\mathsf{F}_\mathsf{k}(u, w_i^k) = \mathsf{F}_\mathsf{k}(u, w_j^t)$ for $i \neq j$. Given that each bucket is modeled as a set, only one entry eventually appears in the bucket, which ultimately affects the total number of PRF evaluations in the bucket. The above observation holds for both Type-1a and Type-1b. The difference between Type-1a and Type-1b is that for the case of Type-1a, the repeated password w appears *only as* $\mathsf{F}_\mathsf{k}(u, w) \oplus 1$ in the bucket, while for the case Type-1b the repeated password w appears *both as* $\mathsf{F}_\mathsf{k}(u, w) \oplus 1$ and as $\mathsf{F}_\mathsf{k}(u, w)$ in the bucket. Let q be the number of real passwords in $\mathcal{Z}$. Formally, the sum of Type-1a and Type-1b collisions is disclosed as $q \cdot n - |\mathcal{Z}|$, where $|\mathcal{Z}|$ is the number of PRF evaluations present in $\mathcal{Z}$. From an adversarial point of view, the absence of passwords from the bucket reveals a Type-1 collision, but it does not reveal whether the cause is a Type-1a or a Type-1b collision.

Successful Choice of $\tau \Rightarrow$ Increased Leakage. There is an intrinsic tension between choosing an accurate similar-password generating function τ and minimizing leakage. Recall that the objective of the τ function is to take as input a single real password and guess a collection of passwords that are likely to be chosen by this user. Suppose that a user u has multiple compromised credentials, then if τ serves its purpose, it would generate synthetic passwords, all of which were also generated by user u and, potentially, were breached at some point in the past. Thus, all the configurations (choice of n and τ) for which `MIGP` provides **strong protection against tweaking attacks** are also the configurations that maximize the number of collisions, and consequently, **increase the leakage**. Therefore, the proposed leakage profile [11] cannot admit a parameterization of τ_n that simultaneously offers strong protection against tweaking attacks and minimal risk of exposing the stored credentials.



(a) Type-0 (b) Type-1a (c) Type-1b

Figure 4: The three types of τ-collisions. The left side of each panel shows a graph representation of the collision (red nodes are real passwords, green nodes are generated through τ), while the right side of each panel shows the effect of the collision on the bucket. A "hole" in the bucket (depicted in red diagonal lines) may come either from Type-1a or Type-1b.

Overall, while the existence of a *single* collision seems harmless at first sight, multiple collisions reveal a network of "semantic" relations across different passwords. This information-rich structure, see Figure 14, can be used to learn a lot of privacy-sensitive information about the characters of an encrypted password. Our attacks show that it is possible for an attacker to perform the reconstruction step "*collision*"⇒"*semantically similar passwords*".

Empirical Analysis of τ-collision Next, we provide statistics on the appearance of τ-collision in a real password dataset. For the evaluation, we use the same dataset employed in MIGP paper [11], i.e., the 4iQ password leaks collection [26]. This is a dataset of 1.4 billion unique email-password pairs. As τ_n function, we consider Das-r and P2P, where n is either 10 or 100 (the most common parameters used in the original MIGP [11]). We then run the processing protocol on the above four configurations, i.e., $\texttt{Das-r}_{10}$, $\texttt{Das-r}_{100}$, $\texttt{P2P}_{10}$, $\texttt{P2P}_{100}$, and count the number and type of collisions for each user in $\mathcal{D}$. By using Das-r as our τ function and $n{=}10$, we recorded the percentage of users with at least two passwords with τ-collision to be 21.0%. This percentage of users goes up to 23.2% for $n{=}100$. On the other hand, for P2P the percentage of users with at least two passwords with τ-collision slightly drops to 20.2% for $n{=}10$, but it reaches 24.6% at $n{=}100$. Interestingly, Type-0 collisions never occur in isolation in 4iQ; that is, whenever there is a Type-0 collision within the passwords of a user, there will also be a Type-1

4. Breach Extraction Attack: Filtering Guesses Based On the Cryptographic Leakage

In this section, we present a "warm-up" leakage attack with the goal of extracting the *non-publicly available* portion of the dataset, i.e., a breach extraction attack. This first attack uses the exact number of τ-collisions to **impose constraints and filter out** password guesses that do not fit the observed leakage; that is, if the attacker observes X Type-0 collisions and Y Type-1 collisions, then he goes down a list of potential password guesses and discards all guesses that do not give X Type-0 and Y Type-1. Despite the simplified setting (where the target user is the only one hashing to this bucket), this attack highlights the advantage gained by the attacker through τ-collisions—an advantage overlooked in the design of MIGP [11].

On the Simplified Setting. This attack considers a simplified setting where (i) multiple credentials of the *target user* u_{trgt} have been compromised, (ii) the attacker has access to all compromised passwords of the user except the target password w_{trgt}, and (iii) the target user u_{trgt} is the only user of $\mathcal{D}$ in the bucket. The attack in this subsection relies on a simplified setting as conditions (ii) and (iii) may not always be applicable. However, it is worth mentioning that due to MIGP's parameterization, $\mathcal{D}$ can assign only a single user to a bucket, thus fulfilling condition (iii). Condition (ii) is met in the following scenario: when the target user u_{trgt} is initially present in a publicly disclosed dataset within MIGP, and subsequently, a new unseen password w_{trgt} from a non-public dataset is added to $\mathcal{D}$. While not universally applicable, this scenario is realistic, wherein the attacker is aware of all but one of u_{trgt}'s compromised credentials. We believe that privacy guarantees should hold regardless of the likelihood of the setting. Overall, this attack in the simplified setting illustrates the problematic nature of the leakage profile described in Equation (1).

Threat Model. The attacker is a user A of MIGP who aims to infer the credentials of another *target user* u_{trgt}. The set of credentials $\mathcal{W}_{\mathsf{trgt}}$ of the target user stored in MIGP is a combination of publicly available compromised passwords and a single compromised password that is *not publicly available*, i.e., $\mathcal{W}_{\mathsf{trgt}}{=}\{\bar{w}_1, \ldots, \bar{w}_m, w_{\mathsf{trgt}}\}$. Given the knowledge of the m public passwords, A's objective is to guess the unknown *target password* w_{trgt} with a small number of queries. We operate in a simplified model where the bucket that contains u_{trgt} credentials contains only $\mathcal{W}_{\mathsf{trgt}}$ (and their synthetic password counterparts).

Attack Overview. The basic premise of this attack is that the τ-collision can be utilized by the adversary to impose constraints on how w_{trgt} relates to the publicly known passwords and their synthetic generations via τ. By doing so, the attack *filters out* all guesses that do not match the pattern of τ-collisions and reduce the search space for w_{trgt}. The attack follows a two-step offline strategy:

(1) Setup phase: The attacker A starts with a list of guesses $\mathcal{G} = [g_1, \ldots]$ to use as candidates for the target password w_{trgt} (i.e., the hypothesis space). A uses the target's username u_{trgt} to compute the bucket index and issues a query to obtain the bucket $\mathcal{Z}_j$ associated to u_{trgt} from the server, following Figure 2. Then, A queries the server with u_{trgt}'s public passwords $[\bar{w}_1, \ldots, \bar{w}_m]$ learning the mapping between the PRF evaluation and its plaintext counterpart;

that is: $\{t_1 \leftrightarrow \mathsf{F_k}(u_{\mathsf{trgt}}, \bar{w}_1), \ldots, t_m \leftrightarrow \mathsf{F_k}(u_{\mathsf{trgt}}, \bar{w}_m)\}$.

Algorithm 1: FILTERINGATTACK

```
Data: Guess list G, target's username u_trgt, target's public
      passwords {w̄_1, ..., w̄_m}.
Result: An updated guess list G.
// Setup Phase
1  Compute label j = Ĥ(u_trgt) and request Z_j from Server;
2  Query the Server on pairs (u_trgt, w̄_1), ..., (u_trgt, w̄_m) to
     retrieve t_1 = F_k(u_trgt, w̄_1), ..., t_m = F_k(u_trgt, w̄_m);
// Offline Phase
3  Compute plaintext bucket based on public passwords, i.e.,
     multiset PPass ← {w̄_1, ..., w̄_m} ∪ {τ_n(w̄_1), ..., τ_n(w̄_m)};
// Type-0 Public2Target (P2T) Filtering
4  Find the pairs FlipPr← {PRFout, PRFout ⊕ 1} of Type-1
     collisions in Z_j;
5  If a pair from FlipPr is not in
     {t_1, t_1 ⊕ 1}, ..., {t_m, t_m ⊕ 1}, there is a Type-0 collision
     from a public password to w_trgt;
6  Filter out all guesses g from G such that g ∉ ⋃_{i=1}^m τ_n(w̄_i);
// Type-0 Target2Public (T2P) Filtering
7  If there is a public password w̄_i that does not appear as a
     Type-0 collision in PPass but its PRF evaluation t_i appears
     as a Type-0 collision in FlipPr, then we have a Type-0
     collision from w_trgt to a public password;
8  Filter out all guesses g from G such that w̄_i ∉ τ_n(g);
// Type-1 Filtering
9  for every g in G do
10     Define D_g = {(u_trgt, w̄_1), ..., (u_trgt, w̄_m)} ∪ {(u_trgt, g)};
11     Run locally the protocol from Figure 1 on D_g to get bucket
         Z'_j that captures the collisions of g with public passwords;
12     If |Z'_j| ≠ |Z_j|, the number of Type-1 collisions that w_trgt
         is involved in does not match, filter out g from G;
13 end
14 return the updated password list G;
```

(2) Offline phase: A, accessing the bucket $\mathcal{Z}_j$, enumerates all the collisions involving w_{trgt}. Based on the type of collision, the attacker A filters out guesses from the input list $\mathcal{G}$. The first filtering step (Lines 4−6 in Algorithm 1) concerns the case where there is a Type-0 collision from a publicly known password of the user to the (unknown) target password w_{trgt}. In this case, the bucket $\mathcal{Z}_j$ will contain a PRF evaluation that appears once in its regular form and once with its last bit flipped while also being absent from $\{t_1, \ldots, t_m\}$. The attacker can filter out all guesses from $\mathcal{G}$ that do not satisfy the above detectable pattern. The second filtering step (Lines 7−8) concerns the case where there is a Type-0 collision from the (unknown) target password w_{trgt} to a publicly known password and, additionally, this public password does not have another Type-0 collision with the set $\{\bar{w}_1, \ldots, \bar{w}_m\}$. In this case, the attacker can verify that one of the PRF evaluations $\{t_1, \ldots, t_m\}$ also appears in $\mathcal{Z}_j$ with its last bit flipped, while the corresponding underlying plaintext password does not participate in a Type-1 collision with $\{\bar{w}_1, \ldots, \bar{w}_m\}$. The attacker can filter out all guesses that do not satisfy the above detectable pattern. Finally, as we detailed in Subsection 3.1, there is no way for an attacker to differentiate between Type-1a and Type-1b. Luckily, both of these collisions result in a "hole" in the bucket, and since the *aggregate* number of absent passwords is observable, the attacker can filter out all guesses that do not satisfy this number (see Lines 9−13 in Algorithm 1).

The Filtering Attack algorithm outputs an *updated* list of guesses that meet *all observed collisions simultaneously*. The attacker can now start an online attack using this updated list to significantly increase the effectiveness of his guesses based on the observed leakage from τ-collisions.

Our description in this Section focuses on the algorithmic intuition of this warm-up breach extraction attack. We present its evaluation in Appendix D.

In conclusion, this warm-up attack exemplifies how an attacker can exploit τ-collisions to glean information about stored passwords. Building on this foundation, we broaden our investigation, relaxing our threat model and introducing practical attacks against protocol implementations.

5. Additional Leakage from Cloudflare's `MIGP` Implementation Design Choices

In the following, we conduct a security analysis of the *implementation* of the `MIGP` protocol by *Cloudflare* [22], which we refer to as $\texttt{MIGP}_{\texttt{CF}}$. According to [11], the implementation $\texttt{MIGP}_{\texttt{CF}}$ is currently deployed and openly available and offers a (1) public-facing API similar to HIBP, and (2) a new breach alerting feature within *Cloudflare*'s web application firewall (WAF) product. This service is an opt-in feature in WAF, which detects login requests to *Cloudflare* customer websites. We note that there is also a `Python` code [27] with the same leakage as $\texttt{MIGP}_{\texttt{CF}}$ while introducing additional vulnerabilities (see Appendix C.1).

The running instance of $\texttt{MIGP}_{\texttt{CF}}$ uses `Das-r` as a τ function with parameter n=8. Our interaction with the *Cloudflare* team confirmed our initial analysis that the public-facing API database of $\texttt{MIGP}_{\texttt{CF}}$ has been populated only with the credentials from the `4iQ` [26] collection. $\texttt{MIGP}_{\texttt{CF}}$ implements a different preprocessing step than the one formalized in Figure 1 (and, so in [11]), which results in additional leakage compared to what was reported in the previous section. The most significant difference is that $\texttt{MIGP}_{\texttt{CF}}$ [22] defines each bucket as a `List` Abstract Data Type (ADT) while the original protocol [11] instructed that each bucket is a `Set` ADT.

We briefly summarize the impact of these implementation choices in the following:

(1) Preserved PRF Duplicates. Duplicate PRF evaluations remain in a bucket modeled as a `List`, rather than being eliminated by the `Set`. The attacker can now infer Type-1 collisions *with certainty*, no guesswork needed. The attacker can not only identify *which* PRF outputs had a collision but also count *how many times* each collision occurred. Additionally, the attacker does not need to have any prior knowledge of the number of passwords in the bucket (a simplifying assumption we made in the warmup attack). In $\texttt{MIGP}_{\texttt{CF}}$, the number of passwords for bucket $\mathcal{Z}_i$ can be computed as $d_i = \frac{|\mathcal{Z}_i|}{(n+1)+1}$. The denominator captures the fact that the list stores the original password plus the n passwords generated by τ. As we discuss next, the extra $+1$ term is due to storing the PRF evaluation of the username.

(2) PRF Evaluation of the Username. For every real password in $\mathcal{D}$, $\texttt{MIGP}_{\texttt{CF}}$ appends an additional entry in

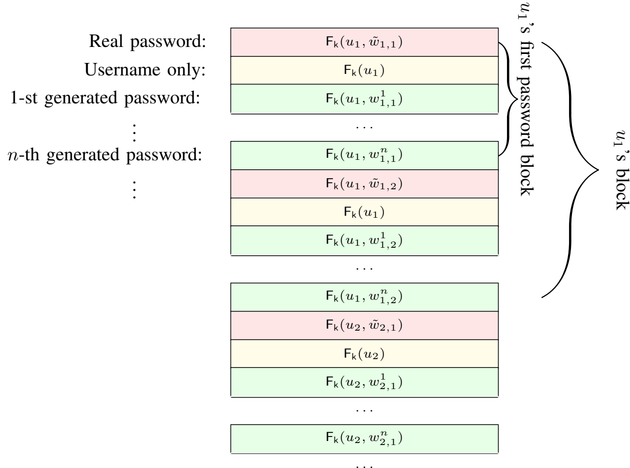


Figure 5: Structure of a bucket in $\texttt{MIGP}_{\texttt{CF}}$ implementation.

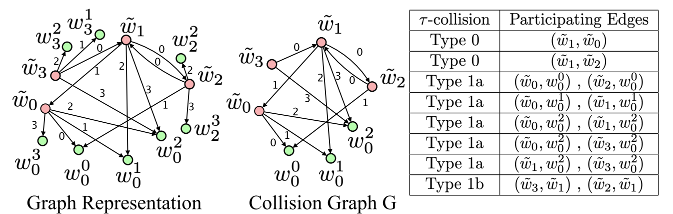
the bucket that corresponds to the PRF evaluation of the username alone, denoted as $\mathsf{F}_\mathsf{k}(u_i)$ (see the yellow entries in Figure 5). This entry allows a user to verify if her email address is included in $\mathcal{D}$ *without needing to input their password*. Note that $\mathsf{F}_\mathsf{k}(u_i)$ is replicated for every password of user u_i. Ultimately, the feature unintentionally exposes both the number of distinct users in the bucket and the exact number of passwords per user at no query cost for the attacker. If each bucket had been implemented as a set, this feature (that is *not part of the original protocol* [11]) would not have resulted in additional leakage.

(3) Preserved Ordering. When processing the j-th password of u_i, the protocol appends the PRF evaluation of $w_{u_i,j}$ followed by the n PRF evaluations on the corresponding similar passwords $[\mathsf{F}_\mathsf{k}(u_i, w^1_{u_i,j}), \ldots, \mathsf{F}_\mathsf{k}(u_i, w^n_{u_i,j})]$. We emphasize that $\texttt{MIGP}_{\texttt{CF}}$ follows strictly the aforementioned order of appending actions.[5] Since each bucket is implemented as a list, the order in which elements are added is maintained. Thus, for every element in the bucket, an adversary can use the position of this entry to determine whether it corresponds to a real password evaluation or a similar password generated through τ. More critically, in the case of a similar password, the attacker can determine the rank in which τ produced the corresponding password. This bucket structure is depicted in Figure 5. Hereafter, with the term "*password block*", we refer to the consecutive $n = 10$ PRF evaluations generated for a single real password $\tilde{w}$.

While this additional leakage is seemingly harmless on its own, it significantly aids the inferences from τ-collisions.

Collision Graphs. We abstract the implementation leakage using a special graph-based representation that we refer to as a collision graph. Figure 4 shows a graph-based representation where there is an edge for every pair of original-synthetic passwords. On a high level, a collision graph is the above graph but with only the edges that participate in a τ-collision. More formally, a **collision graph** $G=(V,E)$ of a user u_i is a direct graph with labeled edges, where each node represents the PRF evaluation of a password of u_i from bucket $\hat{\mathsf{H}}(u_i)$. When it comes to edges, we have three cases: (i) If there is a Type-0 collision between $\tilde{w}_i$ and $\tilde{w}_j$, then edge $(\tilde{w}_i, \tilde{w}_j)$ is in E, (ii) If there is a Type-1a collision from $\tilde{w}_i$ and $\tilde{w}_j$ to synthetic password w, then edges $(\tilde{w}_i, w)$ and $(\tilde{w}_j, w)$ are in E, and (iii) If there is a Type-1b collision from $\tilde{w}_i$ and $\tilde{w}_j$ to real password $\tilde{w}_k$, then edges $(\tilde{w}_i, \tilde{w}_k)$ and $(\tilde{w}_j, \tilde{w}_k)$ are in E. Every edge comes with a label in $[1, n]$ reporting the rank of the application of τ_n that generated w'. See Figure 6 for an illustrative example. Hereafter, to denote such direct edges, we also use the notation $w \xrightarrow{r} w'$. Given a bucket $\mathcal{Z}$, the attacker can build one collision graph per user in $\mathcal{Z}$. This operation does not require any external information or query to the server. Once a collision graph is built, its structure is used to infer information about the underlying passwords.

Responsible Experimental Setup. Our attacks require some degree of interaction with the $\texttt{MIGP}_{\texttt{CF}}$ production server via API calls. To minimize the risk of any adverse effects on the server and avoid exposing any non-public credentials (if any) we conducted the attacks on a simulated environment. Specifically, we used an instance of [22] with *identical parameters* to those of the production server. However, we stress that the introduced attacks would have been equally effective on the $\texttt{MIGP}_{\texttt{CF}}$ production server without any required modifications.



τ-collision	Participating Edges
Type 0	$(\tilde{w}_1, \tilde{w}_0)$
Type 0	$(\tilde{w}_1, \tilde{w}_2)$
Type 1a	$(\tilde{w}_0, w^0_0)$, $(\tilde{w}_2, w^0_0)$
Type 1a	$(\tilde{w}_0, w^1_0)$, $(\tilde{w}_1, w^1_0)$
Type 1a	$(\tilde{w}_0, w^2_0)$, $(\tilde{w}_1, w^2_0)$
Type 1a	$(\tilde{w}_0, w^2_0)$, $(\tilde{w}_3, w^2_0)$
Type 1a	$(\tilde{w}_1, w^2_0)$, $(\tilde{w}_3, w^2_0)$
Type 1b	$(\tilde{w}_3, \tilde{w}_1)$, $(\tilde{w}_2, \tilde{w}_1)$

Figure 6: Left: A graph-based representation of the relation across all passwords. Middle: The collision graph where each edge is part of a τ-collision. Right: All τ-collisions and their participating edges.

6. Breach Extraction via Collision Graphs: One Known Password and τ Inversion

In this section, we introduce a refined version of the warm-up attack described in Section 4. This new breach extraction attack exploits the additional leakage provided by the $\texttt{MIGP}$ implementation (see Section 5).

A More Realistic Setting. Shifting towards a more realistic setting, we relax the conditions described in the warmup attack. Unlike the conditions in Section 4, where the target user u_{trgt} must be the only one hashing in the bucket, we now allow for an arbitrary number of users in the bucket. Furthermore, while the warmup attack assumed knowledge of all but one of the u_{trgt}'s passwords, in this new setting, the attacker knows *only one* password of u_{trgt}.

Threat Model. Much like Section 4, the attacker is a user A who aims to infer the credentials of another target

5. If u_i has multiple passwords, this process is repeated for every other u_i's passwords, before moving to the next user u_{i+1}.

user u_{trgt}. The target user's credential set $\mathcal{W}_{\mathsf{trgt}}$ in $\mathcal{D}$ consists of one publicly available compromised password and an undisclosed number of not publicly known compromised passwords. Given the knowledge of a single password, A's objective is to guess the rest of the passwords of u_{trgt} with as few queries to the server as possible.

Attack Objective. For this attack, we switch to the graph representation of the passwords of u_{trgt}, i.e., the collision graph. Let G be the the collision graph of u_{trgt} (e.g., Figure 7). A node is considered *visited* when the attacker successfully reconstructs its plaintext password. The objective of the attack is to visit all nodes in G.

Thus, attacker A has to handle two cases: (i) given a known password w, traverse edge $w \xrightarrow{r} w'$ in its natural direction to derive the unknown password w', and (ii) given a known password w, traverse edge $w \xleftarrow{r} w'$ in its *opposite* direction to derive the unknown password w'. In the first case, the attacker can trivially derive the unknown password w' by applying τ on the known password w and selecting the output with rank r, i.e., $w'=\tau_n(w)_r$. The edges of (i) are referred to as "*natural edges*". Traversing natural edges does not require any queries to the server since τ is applied locally. In the second case, the attacker must traverse an edge in the opposite direction, requiring the *inversion* of the application of τ of rank r denoted as $w'=\tau_n^{-1}(w, r)$. However, since τ is non-invertible in the general case, the attacker needs to use an approximation of the inverse function τ_n^{-1} in this scenario. Our approximation is a generative model that produces candidate pre-images offline, which can be validated by querying the server.

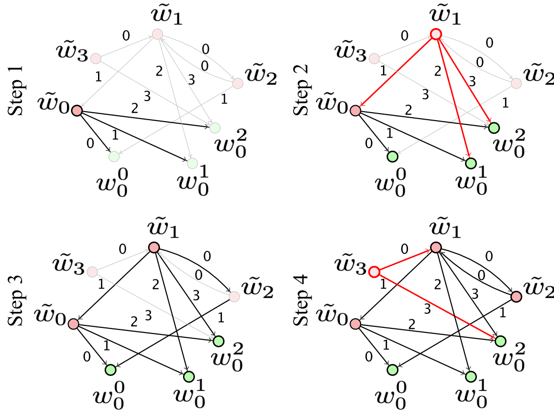


Figure 7: Traversal of a collision graph. Transparent nodes are unknown passwords, while non-transparent are known/reconstructed passwords. Red-highlighted edges are inputs to the τ_n^{-1} model for guessing the red-circled target.

Approximate τ_n^{-1} via Deep Learning. The above discussion on edge traversals focuses on a single edge, but, in reality, an unknown original password w' may be neighboring with *multiple known/reconstructed* passwords due to τ-collisions. This redundancy benefits the attacker, allowing them to synthesize several (known-password, rank) pairs that are adjacent to target password w' and thus derive the plaintext value of w'. An illustrative example is presented in Step 2 of Figure 7, where the unknown node $\tilde{w}_1$ has three red edges to known neighboring passwords: $\tilde{w}_0$, w_0^1, and w_0^2. Formally, the proposed inversion model is a function that takes as an input a collection of (known-password, rank) pairs of nodes that neighbor to target w' and outputs an ordering of guesses for the unknown password w', i.e., $\tau_n^{-1} : \{(\mathcal{W}, [1, n])\}^* \rightarrow [\mathcal{W}]$, where $\mathcal{W}$ is the password space and $[\mathcal{W}]$ is an ordering on the latter. The ordering gives a smaller rank to passwords with a higher probability of being the pre-image of the input pairs. The attack might be further improved by incorporating auxiliary information on the target; for instance, by conditioning the model generation on the user's email address [28].

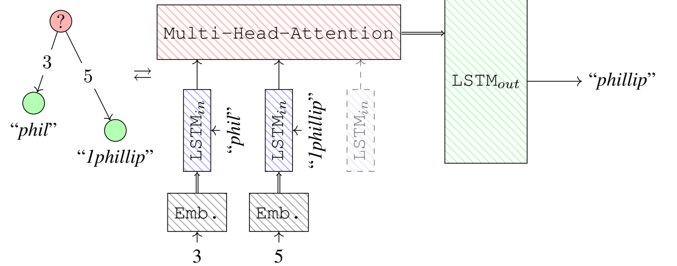


Figure 8: The deep learning pipeline for approximating τ^{-1}. The origin of each $\Rightarrow$ is used as the initial state in an LSTM.

Deep Learning Architecture. We approximate τ_n^{-1} using a deep neural network. A representation of the architecture is depicted in Figure 8. For a known password $(w_{\text{known}}, r_{\text{known}})$, the model first maps the scalar value of the rank r_{known} to a dense vector using an embedding matrix (depicted as a black-striped box in Figure 8). This vector initializes the state of a Long Short-Term Memory network [29] $\texttt{LSTM}_{in}$ (blue-striped box), which processes the characters of w_{known} one-by-one and outputs a vector after the last character. This procedure is applied in parallel for every known password-rank pair neighboring the target password, generating a collection of vectors. A Multi-Head Attention mechanism [30] combines these vectors through a learned parametric transformation to produce dense vectors as an output. These vectors initialize the state of the final 2-layers $\texttt{LSTM}_{out}$ (green-striped box). $\texttt{LSTM}_{out}$ generates a guess for the unknown target password based on the inputs. In essence, $\texttt{LSTM}_{out}$ can be regarded as a conditional variant of the model proposed by Melicher et al. [31].

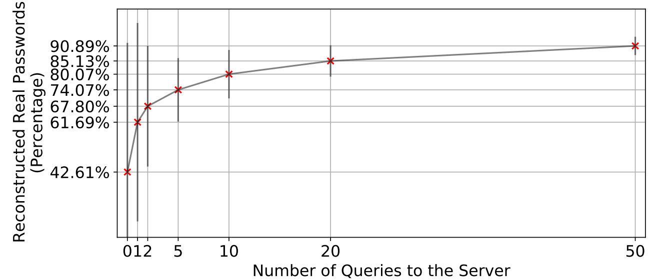
Deep Learning Training/Inference. Following the setup of [11], we split the breached collection `4iQ` [26] in two equal-sized sets: D_0 and D_1, ensuring that no user has credentials in both sets simultaneously. We use D_0 to populate the $\texttt{MIGP}_{\texttt{CF}}$ server, reserving D_1 as training data for the model of our attack. We train the model in a supervised manner. That is, for each user in D_1, we build a collision graph by the leakage from $\texttt{MIGP}_{\texttt{CF}}$. Subsequently, for each node w' in the collision graph with at least one outgoing edge, we generate a collection of one-hop neighbors, denoted as $(w_1, r_1), \ldots$. This collection of input $(w_1, r_1), \ldots$, along with the desired output w', constitutes a single labeled input

data point for the training process. The entire model is then end-to-end jointly trained as a single network. Once trained, we obtain an autoregressive and conditional model that assigns probabilities to passwords in the password space. During inference, the model generates guesses by exploring the password space in decreasing probability order using Beam search or ancestral sampling.

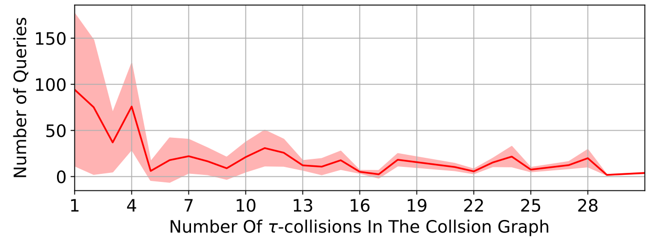
On Prioritizing Node Visits. The last piece of the puzzle is to decide how to prioritize the use of τ_n^{-1} when the traversal is presented with more than one potential input target (i.e., unknown passwords with outgoing edges to known passwords). Depending on the strategy for prioritizing the input targets, we may have a different number of queries from the attacker to the server. We present a greedy strategy for reducing the number of queries to the server.

We illustrate the traversal algorithm based on Figure 7. Given a collision graph G and an initial known password $\tilde{w}_0$, attacker A starts by visiting all the nodes reachable from $\tilde{w}_0$ by traversing the black-colored natural edges in Step 1; thus, A derives w_0^0, w_0^1, and w_0^2 by locally applying τ and without interacting with the server. In general, A can use any graph traversal algorithm on natural edges e.g., BFS rooted in $\tilde{w}_0$. If the graph has not been fully visited after the traversal algorithm, it means the attacker has to traverse non-natural edges incident to a known password. When faced with multiple candidate nodes, the attacker employs a *score function* to determine the next target node. Specifically, A applies τ_n^{-1} to all candidate nodes and picks the one with the highest confidence, defined as the probability assigned by the model to the first ranked element of the password space. For example, in Step 2 of Figure 7, the attacker chooses $\tilde{w}_1$ over $\tilde{w}_2$ and $\tilde{w}_3$ partly because $\tilde{w}_1$ has three known neighbors and, thus, a higher confidence on the first guess. Once the next target node is determined, the attacker uses τ_n^{-1} to generate guesses and validates them against the server through queries. Upon a successful match for $\tilde{w}_2$, the attacker updates the set of known passwords and continues the traversal of as many natural edges as possible e.g., edge $(\tilde{w}_1, \tilde{w}_2)$ in Step 3. The attack ends when all the nodes in the collision graph have been visited. This protocol is formalized in Algorithm 2. We note here that when initiating the attack, the attacker must execute a single *setup query* to the server using w to identify which PRF evaluation in the bucket corresponds to the w entry.

Evaluation: Attack Effectiveness. For each user in D_0 with τ-collisions, we construct a collision graph using the $\texttt{MIGP}_{\texttt{CF}}$ implementation leakage. Randomly selecting a real password as the known initial password, we execute the breach extraction attack algorithm until all unknown passwords are reconstructed. The average number of queries needed by the attacker to reconstruct an (unknown) real password is shown in Figure 9a. Remarkably, around 42% of the real passwords within a collision graph can be reconstructed without requiring any queries to the server, i.e., via natural edge traversals which are implemented with a local application of the τ function. Approximately 61% of the unknown passwords in the collision graphs can be guessed with at most one query. This percentage increases to 74.2%



(a) The effectiveness of the breach extraction attack measured as the number of queries per password (X-axis) that are needed to reconstruct the plaintext password from the leakage (Y-axis).



(b) Analysis on how the number of τ-collisions in a collision graph (X-axis) affects the breach extraction within a graph (Y-axis).

Figure 9: Evaluation of the breach extraction attack using the leakage from $\texttt{MIGP}_{\texttt{CF}}$ and a single known password.

after only five queries and to 90.4% after 50 queries. The high number of queries is due to rules of `Das-r` that remove characters from the original password. In such cases, the model of τ_n^{-1} needs to generate the deleted characters with very little context (see rules $2-4$ in Table 2)

Evaluation: The Impact of Collisions. In the next experiment, we study how the number of τ-collisions in a collision graph affects the effectiveness of the breach extraction within this graph. Intuitively, graphs with more τ-collisions offer either (i) more inputs for the model τ_n^{-1} and, consequently, more confident guesses for τ_n^{-1}, or (ii) more candidate unknown passwords when τ_n^{-1} has to choose its next target, or (iii) both of the above. This relationship is depicted in Figure 9b, where the number of queries required to reconstruct a password is plotted against the number of collisions. Interestingly, the number of queries to the server drops from 100 to less than 10 after only five τ-collisions. Users with more τ-collisions revealed through the leakage of $\texttt{MIGP}_{\texttt{CF}}$ are found to be more vulnerable to our attack.

7. Extracting Password Templates: A Neural Network Approach With No Known Passwords

In this section, we provide the most general breach extraction attack that uses both the leakage from the cryptographic design as well as the leakage from Cloudflare's implementation to extract "*templates*" of the unknown server's passwords by solely **exploiting the topology of their collision graph** from the bucket, i.e., the attacker doesn't know any password of the target user. In this context, the term

Table 1: Set of properties/classes inferred by the Graph Neural Network.

Regex:	**(1)** “*\d+$*”	**(2)** “*0.+*”	**(2)** “*1.+*”	**(4)** “*.+a$*”	**(5)** “*[a-zA-Z]+$*”	**(6)** “*[A-Z].+$*”	**(7)** “*.+0$*”	**(8)** “*[^\d]+\d$*”	**(9)** “*[^\d]+\d\d$*”
Description:	It consists solely of digits.	It starts with “*0*”	It starts with “*1*”	It ends with “*a*”	It consists solely of alphabetic characters	It starts with a capital letter	It ends with “*0*”	It ends with a digit	It ends with two digits

“*template*” refers to a generalized description of the underlying string, such as specifying that the password begins with the character “*0*” or consists entirely of numeric characters. After successfully mounting the attack, the adversary gains knowledge of password templates, allowing them to narrow down the search space by considering only guesses that align with the extracted templates or directly using template-based password generation techniques [32], [33]. Similar to previous attacks, an online phase follows, where each template-agreeing guess is issued to the server for verification by comparing its OPRF evaluation to the corresponding entry in the bucket. Interestingly, once the adversary correctly guesses a single password, the adversarial setting can be reduced to the one of Section 6. Consequently, the adversary can also use the model for τ_n^{-1}.

Threat Model. Much like in previous Sections, the attacker is a user A who aims to infer the credentials of another target user u_{trgt}. The target user's credential set $\mathcal{W}_{\mathsf{trgt}}$ in $\mathcal{D}$ consists of an undisclosed number of not publicly known compromised passwords. Given the knowledge of no passwords of u_{trgt}, A's objective is to guess the passwords of u_{trgt} with as few queries to the server as possible.

Graph Neural Networks (GNNs). At the core of our attack is the use of Graph Neural Networks (GNNs) [34]. On a high level, a GNN is a neural model that operates over graphs. GNNs can be viewed as learning techniques that capture the topological information inherent in graph-like data. This is accomplished through a message-passing algorithm, which iteratively propagates information among neighboring nodes and edges, thereby updating their internal representations. Once an appropriate representation has been attained, the GNN's output is fed into a conventional machine learning architecture, such as a fully-connected network, to execute the classification task. The main characteristic and pivotal property of a GNN is that it is input permutation-invariant. That is, if the model is applied to two isomorphic graphs, it will produce the same output irrespective of the original representation of the graphs.

Attack Objective. In this attack, we train a GNN to perform a node classification task. Taking as input a whole collision graph, the GNN's objective is to infer a password template for each real password[6] in the graph (e.g., the red nodes in Figure 6). In our setting, we define a password template as a composition of the 9 structural properties listed in Table 1. We treat each property as a binary variable, with a value of 1 indicating that the GNN determines the collision graph's structure implies the corresponding template, and 0 indicates otherwise. For instance, the password “***Password10***” would have template $[0,0,0,0,0,1,1,0,1]$ with respect to the templates of Table 1, where the value of the i-th bit signals the presence or absence of the i-th property. Even when a bit is set to 0, it provides information about the underlying password by excluding all passwords with that specific property from the search space. It must be noted that some properties are not independent of each other; for instance, the presence of property 1 (only digits) automatically excludes properties 3, 4, and 5 from being present. The properties in Table 1 are derived manually, considering the transformations induced by the τ function used in $\mathtt{MIGP_{CF}}$, with $n{=}8$ (the value used by the $\mathtt{MIGP_{CF}}$ production server). If we were to increase the value of n, more properties would emerge, providing the attacker with greater discriminative power and allowing them to infer more complex templates from the collision graphs.

Modeling. Given a collision graph, the GNN initializes the states of the nodes and edges as follows. Nodes receive one of three possible feature vectors, depending on whether they represent (1) a real password, (2) a generated password, or (3) both simultaneously (e.g., in the case of a Type-0 and Type-1b collision). Similarly, edges are initialized with one of eight possible values that correspond to their label, representing the rank r of the application of the τ function that resulted in the edge. Both node and edge feature vectors are initialized randomly at the outset and subsequently optimized during the training process. Once the graph is initialized, we conduct 6 rounds of message passing, during which both node and edge representations are updated. As an update function, we employ a fully connected-based architecture alternated with batch-normalization and a simple average pooling mechanism. Finally, following the message passing iterations, the nodes' representations serve as input for a fully-connected network. This network generates the final prediction, producing disjointed probability values for each class listed in Table 1.

Training. The model is trained following the same general procedure used for the first attack in Section 6. Given the plaintext passwords in $\mathcal{D}_1$, we extract collision graphs from the buckets generated by the $\mathtt{MIGP_{CF}}$ setup phase. Then, for each collision graph G, we collect the set of labels by deriving the templates (i.e., classes) for each real password in G. After having generated the training set, we fit the model in a supervised way. That is, given an input collision graph, we train the GNN to assign the right set of classes to each real password.

Evaluation. During our evaluation, we trained the GNN on $\mathcal{D}_1$ and tested it by running the setup phase of $\mathtt{MIGP_{CF}}$ on $\mathcal{D}_0$ to derive collision graphs for users with at least one τ-collision. We then applied the GNN to each graph and tested its accuracy in inferring the properties from Table 1.

The model predicts a property with an average accuracy of 90.6%. Figure 10 shows the model's individual class accuracies. The lowest accuracy class (79%) is for passwords

6. Once recovered the real password, similar passwords can be simply generated by applying τ on it.

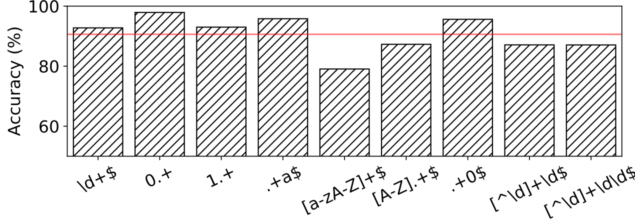


Figure 10: GNN's accuracy per individual property. The Red lines denotes average accuracy.

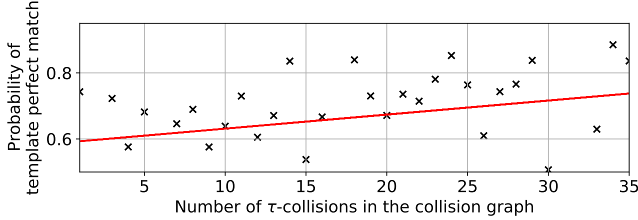


Figure 11: Analysis on how the number of τ-collisions in a collision graph (X-axis) affects the probability of the GNN predicting the **exact** password template (Y-axis).

solely of alphabetic characters.

In 61% of instances, the model perfectly matches the password template, accurately guessing all binary properties simultaneously. The template with the highest accuracy in GNN's predictions is $[0, 0, 0, 0, 1, 0, 0, 0, 0]$, with 82.2% accuracy. The all-zeros template $[0, 0, 0, 0, 0, 0, 0, 0, 0]$, follows closely at 79.5%. The most complex template the model can perfectly match is $[0, 0, 0, 1, 1, 1, 0, 0, 0]$, accounting for only 0.32% of all passwords in `4iQ` [26].

Most model errors are false negatives, resulting from missing edges in collision graphs.[7] The likelihood of encountering the required τ-collision increases with the number of τ-collisions in the collision graph. This correlation is shown in Figure 11.

In summary, this attack further highlights that τ-collisions lead to information leakage about the passwords in `MIGP`. We will introduce a novel `MIGP` protocol that offers ideal security without practical cost next.

8. `MIGP` 2.0: A New Leakage Profile

In this Section, instead of proposing a protocol and then defining its leakage profile, we first establish a desired leakage profile that is not susceptible to the newly found attacks and then propose a protocol that achieves it.

The Achilles' heel of the leakage described in Equations (1) and (2) is that it *reveals τ-collisions* within each bucket. We have identified two reasons that cause τ-collisions. First, the $\mathsf{result}_i = \mathsf{match} + \mathsf{similar}$ of Equation (2) is responsible for Type-0 collisions (see Figure 4 (a)) since the same password appears both as $\mathsf{F_k}(u, \tilde{w}_i)$ and $\mathsf{F_k}(u, \tilde{w}_i) \oplus 1$ in the bucket. Second, Equation (1) reveals the size of the set, i.e., argument $|\mathcal{Z}_{\hat{\mathsf{H}}(u_i)+1}|$ in $\mathsf{L_{client}}$, which reveals the repetition of synthetic passwords through the "holes" left in the bucket. Type-1a and Type1-b collisions are observable due to the aforementioned "holes". We fix both these issues in our new leakage profile for `MIGP 2.0`.

A Better Leakage Profile for `MIGP`. We observe that the result $\mathsf{match} + \mathsf{similar}$ is not only problematic for security reasons but also redundant to the querier. The crucial information that `MIGP` must convey is that the queried password "appears as is" in the breached credential dataset; the additional fact that it also appears as a similar password does not escalate the severity of the breach. Therefore, in the case of a $\mathsf{match} + \mathsf{similar}$, our newly proposed leakage profile returns just $\mathsf{result}_i = \mathsf{match}$. The new options are:

$$\mathsf{result}_i = \begin{cases} \mathsf{match} & \text{, if } (u_i, \tilde{w}_i) \in \mathcal{D} \\ \mathsf{similar} & , \exists (u_i, \tilde{w}_j) \in \mathcal{D} : \tilde{w}_i \in \tau_n(\tilde{w}_j) \\ & \quad \text{and } (u_i, \tilde{w}_i) \notin \mathcal{D} \\ \mathsf{none} & \text{, otherwise} \end{cases} \tag{5}$$

The updated set of result_i options is crucial to eliminate the presence of Type-0 collisions. Toward fixing the remaining τ-collisions, we observe that the size of the bucket $|\mathcal{Z}_{\hat{\mathsf{H}}(u_i)+1}|$ *varies depending on the number of repetitions* of synthetic passwords. To fix this, we propose a new leakage profile for the client $\mathsf{L_{client}}$ where the information disclosed is *independent* of the number of repetitions. We achieve this by replacing the number of entries of the queried bucket ($|\mathcal{Z}_j|$ of Equation (1)) with t_i which is the number of original breached credentials $(u, \tilde{w}) \in \mathcal{D}$ that are mapped in the queried bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u_i)+1}$ (where u_i is the username of the i-th client query). The new leakage profile is:

$$\begin{aligned} &\mathsf{L_{server}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i\in[q]}) \text{ as defined in Equation (1)} \\ &\mathsf{L_{client}}(\mathcal{D}, \{(u_i, \tilde{w}_i)\}_{i\in[q]}) = (\mathsf{result}_i, t_i)_{i\in[q]} \end{aligned} \tag{6}$$

where $t_i = |\{(u, \tilde{w}) | (u, \tilde{w}) \in \mathcal{D} \text{ and } \hat{\mathsf{H}}(u_i) = \hat{\mathsf{H}}(u)\}|$ and result_i as defined in Equation (5). Our new leakage profile (Equations (5) and (6)) does not disclose any τ-collisions and, in fact, is not affected by which similar passwords are generated or by what is their relationship to the real passwords. This implies that the **security of a protocol that achieves our new leakage profile is not affected by the presence of similar passwords.** This is because the leakage profile for $n = 0$ is identical to the one for $n > 0$, i.e., similar passwords do not increase the leakage.

Our `MIGP` 2.0 Protocol. Our new `MIGP` protocol addresses the τ-collisions by using a new preprocessing phase $\Pi^{\tau_n, \ell}_{\mathsf{new\text{-}init}}$ that produces ℓ buckets differently than [11]. The absence of τ-collisions in our new design eliminates the offline attacks described in 4 and allows us to prove the security with respect to the leakage functions of Equation (6).

New Pre-Processing Phase $\Pi^{\tau_n, \ell}_{\mathsf{new\text{-}init}}$. The formal description of our new pre-processing phase $\Pi^{\tau_n, \ell}_{\mathsf{new\text{-}init}}$ is depicted in Figure 12. This procedure is divided into two parts: A first part in which the original pairs $(u, \tilde{w})$ of the breached dataset $\mathcal{D}$ are encoded into buckets $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$

7. While a password may possess a particular property, the τ-collision(s) required to manifest that specific property in the collision graph might be absent, rendering the model unable to infer its existence.

Pre-preprocessing phase $\Pi^{\tau_n,\ell}_{\text{new-init}}$

Parameters: 2HashDH OPRF F, password generator τ_n, number of buckets ℓ, and hash function $\hat{\mathsf{H}} : \{0,1\}^* \to \{0,1\}^{\log \ell}$.

Server

Input: Dataset $\mathcal{D} = \{(u_1, \tilde{w}_1), \ldots, (u_{|\mathcal{D}|}, \tilde{w}_{|\mathcal{D}|})\}$.

Output: OPRF key k and ℓ buckets $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$.

```
 1 : Sample k ←$ Z_q
 2 : Initialize the lists Z_1 = ⊥, ..., Z_ℓ = ⊥
 ................ Part one: Encoding of original credentials ................
 3 : For every (u, w̃) ∈ D :
 4 :     j = Ĥ(u)
 5 :     Add F_k(u, w̃) to Z_{j+1}
 ................ Part two: Encoding of similar credentials ................
 6 : Initialize the sets L_1 = ⊥, ..., L_ℓ = ⊥
 7 : For every (u, w̃) ∈ D and for every w ∈ τ_n(w̃) :
 8 :     j = Ĥ(u)
 9 :     If (u, w) ∈ D or (u, w) ∈ L_{j+1} :
10 :         Sample v ←$ {0,1}^λ and add F_k(u, v) to Z_{j+1}
11 :     Else :
12 :         Add F_k(u, w) ⊕ 1 to Z_{j+1} and (u, w) to L_{j+1}
13 : For every i ∈ [ℓ] : Shuffle the entries of Z_i
14 : Return k, Z_1, ..., Z_ℓ
```

Figure 12: Our new pre-processing phase of MIGP produces buckets that do not disclose relations between original and similar credentials. The first part of the protocol encodes original credentials. The second part of the protocol encodes similar credentials while hiding collisions.

much like the original MIGP protocol, and a second part in which the server carefully adds similar passwords (generated through τ_n) to their corresponding buckets by taking into account repetitions. In particular, during the second part of the pre-processing, the server compares the about-to-be-inserted synthetic password against the original credentials and against the similar credentials already encoded in the buckets. If $\tau_n(\tilde{w})$ produces a collision with either of the two, then the server adds a *dummy entry* in the corresponding bucket by adding $\mathsf{F}_\mathsf{k}(u, v)$ where v is random λ-bits long string. Intuitively, this dummy entry has the role of simulating the addition of a non-colliding password. We represent the buckets as lists that we randomly shuffle at the end of the initialization.[8]

Part one of Figure 12 is straightforward, and it is identical to the pre-processing of the original passwords of the original MIGP protocol. The server samples a random PRF key $\mathsf{k} \leftarrow\$ \mathbb{Z}_q$ and, for every original pair $(u, \tilde{w}) \in \mathcal{D}$, it adds $\mathsf{F}_\mathsf{k}(u, \tilde{w})$ to the $j+1$-th bucket $\mathcal{Z}_{j+1}$ where $j = \hat{\mathsf{H}}(u)$. At the end of this first part, the encoding (i.e., PRF evaluation) of all the original pairs is distributed across the ℓ buckets, where the distribution depends on the hash function $\hat{\mathsf{H}}(\cdot)$. In part two of Figure 12, the server adds to the buckets the similar passwords generated through τ_n. To avoid τ-collisions, a similar password w is encoded and added to a bucket $\mathcal{Z}_j$ if and only if the bucket $\mathcal{Z}_j$ does not already contain the encoding of (u, w). Pair (u, w) can be already encoded into bucket $\mathcal{Z}_j$ if either (i) (u, w) is in $\mathcal{D}$, or (ii) there exists an (already pre-processed) original pair $(u, \tilde{w}) \in \mathcal{D}$ whose similar password generation produces w and, in turn, passwords $\tau_n(\tilde{w})$ have been already encoded into the $\mathcal{Z}_j$. Hence, the server needs to keep track of similar passwords that have already been encoded in each bucket (via $\mathcal{L}$ sets). Finally, the server shuffles the entries of every bucket $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$ to uniformly distribute the dummy entries with the original/similar ones. The output of the pre-processing protocol is the OPRF key k and the buckets $\mathcal{Z}_1, \ldots, \mathcal{Z}_\ell$, which will be stored by the server.

Query phase Π^ℓ_{query}. Clients can query the server by using the exact same query protocol Π^ℓ_{query} of the original MIGP (Figure 2). Our way of pre-processing the dataset $\mathcal{D}$ achieves the functionality described in Equation (5). This is because a similar credential (u, w) (where $w \in \tau_n(\tilde{w})$ for some $(u, \tilde{w}) \in \mathcal{D}$) is encoded as a random dummy entry in a bucket $\mathcal{Z}_j$ only if there is $(u', \tilde{w}')$ that has been already encoded in $\mathcal{Z}_j$ and $(u', \tilde{w}') = (u, w)$. Now, if $(u', \tilde{w}') \in \mathcal{D}$, the corresponding entry for $(u', \tilde{w}')$ is encoded as $\mathsf{F}_\mathsf{k}(u', w')$ (i.e., no bit flipped). On the other hand (if $(u', \tilde{w}') \notin \mathcal{D}$), the credentials $(u', \tilde{w}')$ is encoded as $\mathsf{F}_\mathsf{k}(u', \tilde{w}') \oplus 1$ (i.e., bit flipped since $\tilde{w}'$ must be generated using τ_n).

Security. Because of our new pre-processing phase, our new MIGP protocol is $(q, \mathsf{L}_{\text{client}}, \mathsf{L}_{\text{server}})$-secure (Definition 1) for every $q \in \mathbb{N}$, where $\mathsf{L}_{\text{client}}$ and $\mathsf{L}_{\text{server}}$ are the new leakage functions we defined in Equation (6). We refer the reader to Section C for the formal result.

We highlight that, to prove the security of our protocol, it is fundamental to simulate the queried buckets only using t_i. This is possible thanks to our pre-processing phase that always produces buckets (composed of pseudorandom entries) of size $t_i \cdot n$ (i.e., multiple of n) when $n > 0$,[9] independently from the dataset $\mathcal{D}$ and similar passwords generator τ_n used by the server. The old protocol [11] produces buckets of different sizes depending on the number of collisions, which may not be a multiple of n. Hence, the simulator (of the security proof) needs to know the exact number of entries in each bucket (i.e., $|\mathcal{Z}_j|$ is part of the leakage). However, as we have demonstrated, $|\mathcal{Z}_j|$ reveals sensitive information about the original passwords when collisions are present.

Overhead of MIGP 2.0. MIGP 2.0 adds only negligible overhead on top of the old version [11]. Checking if a password is already present in the bucket adds minimal cost. [11] performs an addition of an element using the API of the set-insertion function, and whether the element is actually inserted or not is controlled/decided by the implementation of the insertion-function. One can verify whether a given insertion resulted in an addition by checking if the size of the set increased by one.

8. With an ideal set implementation, the shuffle operation becomes superfluous. However, we intentionally include it to emphasize its critical importance and proactively mitigate potential implementation errors.

9. If no similar passwords are generated (i.e., $n = 0$), bucket $\mathcal{Z}_i$ will be of size t_i.

Second, regarding shuffling, this can be done incrementally while building the buckets. For example, if the bucket is implemented as an array, we can insert each new element in the leftmost empty cell x, and randomly choose any occupied cell y to swap with x, guaranteeing a permuted array in the end.

9. Conclusion

The findings of this work confirm that designing cryptographic protocols with controlled disclosure is challenging, as it requires a thorough understanding of the implications of the proposed leakage profile. Interestingly, the vulnerabilities that we discovered are not due to errors in cryptographic proofs or algorithms, e.g., the private set intersection and the OPRF work as they are supposed to. The issue is that design choices may or may not be problematic based on the *application's context*. In the application of similarity-based C3, a set-based formulation inadvertently reveals the absence of duplicate entries, which in turn reveals hidden connections between the passwords that were encoded. Overall, these findings have the potential to help future controlled-disclosure designs to balance their scalability with *well-understood leakage implications*. This sets the stage for a new wave of cryptosystem designs that are not only more robust but also cognizant of the intricate dance between data privacy and system efficiency.

Acknowledgments

The second author was supported by the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM). The third author was supported in part by grants from the Commonwealth Cyber Initiative (CCI) and the Commonwealth Commercialization Fund (CCF), as well as corporate gifts from Accenture, Lockheed Martin Integrated Systems, and Protocol Labs. The last author was supported in part by the NSF award CNS-2154732 and the Meta Security Research Award.

References

[1] V. Enterprise, "2020 Databreach Investigation Report," https://enterprise.verizon.com/resources/reports/2020/2020-data-breach-investigations-report.pdf, 2020, accessed: November 30, 2023.

[2] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *NDSS*, vol. 14, 2014, pp. 23–26.

[3] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 295–310.

[4] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: An algorithmic framework and empirical analysis," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 176–186.

[5] M. Islam, M. S. Bohuk, P. Chung, T. Ristenpart, and R. Chatterjee, "Araña: Discovering and Characterizing Password Guessing Attacks in Practice," in *32nd USENIX Security Symposium (USENIX Security 23)*, (To appear).

[6] "Have i been pwned?: Check if your email or phone is in a data breach," https://haveibeenpwned.com/, accessed: November 30, 2023.

[7] "Google Blog: Protecting your data, no matter where you go on the web," https://blog.google/technology/safety-security/google-password-checkup-cross-account-protection/, accessed: November 30, 2023.

[8] S. Kannepalli, K. Laine, and R. C. Moreno, "Password Monitor: Safeguarding Passwords in Microsoft Edge," https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, 2021, accessed: November 30, 2023.

[9] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart, "Beyond credential stuffing: Password similarity models using neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 417–434.

[10] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1242–1254.

[11] B. Pal, M. Islam, M. S. Bohuk, N. Sullivan, L. Valenta, T. Whalen, C. Wood, T. Ristenpart, and R. Chatterjee, "Might I Get Pwned: A Second Generation Compromised Credential Checking Service," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022, pp. 1831–1848.

[12] T. Hunt, "Seized Genesis Market Data is Now Searchable in Have I Been Pwned, Courtesy of the FBI and "Operation Cookie Monster"," https://www.troyhunt.com/seized-genesis-market-data-is-now-searchable-in-have-i-been-pwned-courtesy-of-the-fbi-and-operation-cookie-monster/, 2023, accessed: November 30, 2023.

[13] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," in *Proc. of the 13th ACM CCS*, 2006, pp. 79–88.

[14] M. Chase and S. Kamara, "Structured Encryption and Controlled Disclosure," in *Proc. of the 16th IACR ASIACRYPT*, 2010.

[15] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks," in *Proc. of the 42nd IEEE S&P*, 2021.

[16] ——, "Data Recovery on Encrypted Databases With k-Nearest Neighbor Query Leakage," in *Proc. of the 40th IEEE S&P*, 2019.

[17] ——, "The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution," in *Proc. of the 41th IEEE S&P*, 2020.

[18] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-Abuse Attacks Against Searchable Encryption," in *Proc. of the 22nd ACM CCS*, 2015, pp. 668–679.

[19] F. Falzon, E. A. Markatou, Akshima, D. Cash, A. Rivkin, J. Stern, and R. Tamassia, "Full Database Reconstruction in Two Dimensions," in *Proc. of the 27th ACM CCS*, 2020.

[20] E. A. Markatou, F. Falzon, R. Tamassia, and W. Schor, "Reconstructing with Less: Leakage Abuse Attacks in Two Dimensions," in *Proc. of the 28th ACM CCS*, 2021.

[21] E. M. Kornaropoulos, N. Moyer, C. Papamanthou, and A. Psomas, "Leakage Inversion: Towards Quantifying Privacy in Searchable Encryption," in *Proc. of the 29th ACM CCS*, 2022, pp. 1829–1842.

[22] CloudFlare, "MIGP_go," https://github.com/cloudflare/migp-go, 2022, accessed: November 30, 2023.

[23] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-optimal Password-protected Secret Sharing and T-PAKE in the Password-only Model," in *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 2014, pp. 233–253.

[24] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, “Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online),” in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 276–291.

[25] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, “Protecting Accounts from Credential Stuffing with Password Breach Alerting,” in *USENIX Security Symposium*, 2019, pp. 1556–1571.

[26] “Over 1.4 Billion Clear Text Login Credentials Found in Single Database on Dark Web,” https://infowatch.com/analytics/leaks_monitoring/97798.

[27] M. Islam, “MIGP_python,” https://github.com/islamazhar/MIGP_python, 2022, accessed: November 30, 2023.

[28] D. Pasquini, G. Ateniese, and C. Troncoso, “Universal Neural-Cracking-Machines: Self-Configurable Password Models from Auxiliary Data,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024 (To appear).

[29] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[31] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, “Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks,” in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Aug. 2016, pp. 175–191.

[32] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti, “Improving password guessing via representation learning,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1382–1399.

[33] M. Xu, J. Yu, Z. Xinyi, C. Wang, S. Zhang, H. Wu, and W. Han, “Improving Real-world Password Guessing Attacks via Bi-directional Transformers,” in *32nd USENIX Security Symposium (USENIX Security 23)*, (To appear).

[34] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The Graph Neural Network Model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

Appendix A. 2HashDH OPRF

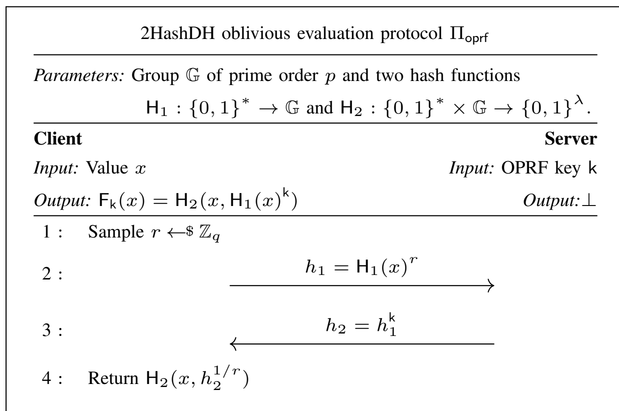


Figure 13: Oblivious evaluation protocol for the 2HashDH OPRF $\mathsf{F}_\mathsf{k}(x) = \mathsf{H}_2(x, \mathsf{H}_1(x)^\mathsf{k})$.

The original work of Jarecki et al. [23], presented such a construction for the case of verifiable OPRF. The protocol we described above is essentially identical to that of [23] except that we ignore verifiability property. This latter costruction (without verifiability) is formally studied in [24].

The security of OPRF is defined as usual for multi-party computation protocols, i.e., the view of an adversary (client or server) can be simulated by having access to the input of the adversarial party and the ideal OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$ (see [24, Figure 1]) which, on input x and k, it returns $\mathsf{F}_\mathsf{k}(x) = y$ to the client (and nothing to the server). When an OPRF protocol is secure (as described above) is said to realize the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$. [24] have demonstrated that the protocol of Figure 13 realizes the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$ under the (n, q)-one-more DH assumption (we refer the reader to [24] for the formal definition of the assumption). Moreover, [24] proves the security of the protocol in the universal composability (UC) setting which allows protocol composition in a secure fashion. Below, we recall the result of [24].

Theorem 1 ([24, Theorem 1]). *Let H_1 and H_2 be two hash functions modeled as random oracles. Also, let q and q_1 be the number of OPRF evaluations and the number of random oracle queries for H_1, respectively. For $n = q+q_1$, if (n, q)-one-more DH assumption ([24, Section 3]) holds for $\mathbb{G}$, then the protocol 2HashDH (Figure 13) realizes the functionality $\mathcal{F}_{\mathsf{oprf}}$ in universal composability setting and random oracle model.*

Appendix B. Client-side Similar Passwords Generation

The query phase protocol we described in Section 2 is a simplification of [11] since we do not consider the possibility of generating similar passwords on client-side. For completeness, we describe how client-side generation is implemented in the original `MIGP` protocol. Let τ_n^{client} be a similar password generator held by the client (this generator can either identical or different to the one used on the server-side). The objective is to allow the client to generate similar passwords $(\widetilde{w}_1, \ldots, \widetilde{w}_m) = \tau_m^{\mathsf{client}}(w)$ and see if the latter match any of the (either original or similar) passwords stored by the server. This is accomplished by allowing the client to compute $(\widetilde{w}_1, \ldots, \widetilde{w}_m) = \tau_m^{\mathsf{client}}$ and invoke m times the ideal OPRF functionality $\mathcal{F}_{\mathsf{oprf}}((u, \widetilde{w}_1), \mathsf{k}), \ldots, \mathcal{F}_{\mathsf{oprf}}((u, \widetilde{w}_m), \mathsf{k})$ where $\mathcal{F}_{\mathsf{oprf}}((u, \widetilde{w}_i), \mathsf{k})$ denotes the invokation of $\mathcal{F}_{\mathsf{oprf}}$ on client's input $(u, \widetilde{w}_i)$ and server's input k. As a result, the client obtains $y_1 = \mathsf{F}_\mathsf{k}(u, \widetilde{w}_1), \ldots, y_m = \mathsf{F}_\mathsf{k}(u, \widetilde{w}_m)$. This is sufficient to check membership as we described in Figure 2. In particular, the client returns $\mathsf{similar}$ if there exists $i \in [m]$ such that $y_i \in \mathcal{Z}_{j+1}$ or $y_i \oplus 1 \in \mathcal{Z}_{j+1}$, and none otherwise. Observe that for the case of similar passwords $\tau_m^{\mathsf{client}}(w)$ generated on client-side, the protocol only returns $\mathsf{similar}$ (and not match) since the check is evaluated over the similar passwords $(\widetilde{w}_1, \ldots, \widetilde{w}_m) = \tau_m^{\mathsf{client}}(w)$. For more details, we refer the reader to [11].

Appendix C.
Leakage-based Security Definition of MIGP

Definition 1 (($q, \mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}}$)-security of MIGP)**.** Fix the public parameters ℓ and τ_n of the MIGP protocol. Let $\mathsf{L}_{\mathsf{client}}$ be a function that, on input the server's dataset $\mathcal{D}$ and the client's credential (u, w), it defines the leakage revealed to the client. Similarly, let $\mathsf{L}_{\mathsf{server}}$ be a function that, on input the server's dataset $\mathcal{D}$ and the client's credentials (u, w), it defines the leakage revealed to the server. The MIGP protocol is $(q, \mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}})$-secure if there exists a pair of PPT simulators $(\mathsf{S}_{\mathsf{client}}, \mathsf{S}_{\mathsf{server}})$ such that, for every $\mathcal{D} = \{(u_1, w_1), \dots, (u_{|\mathcal{D}|}, w_{|\mathcal{D}|})\}$ of original breached credentials and for every q credential queries $(u'_1, w'_1), \dots, (u'_q, w'_q)$, the following two conditions hold:

- **Client-side leakage:** The distributions $\mathbf{View}^{\mathsf{client}}_{\{(u'_i, w'_i)\}_{i\in[q]},\mathcal{D}} = (\mathbf{View}^{\mathsf{client}}_{u'_1,w'_1,\mathcal{D}}, \dots, \mathbf{View}^{\mathsf{client}}_{u'_q,w'_q,\mathcal{D}})$ and $\mathsf{S}_{\mathsf{client}}(\{(u'_i, w'_i)\}_{i\in[q]}, \mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u'_i, w'_i)\}_{i\in[q]}))$ are computationally indistinguishable where $\mathbf{View}^{\mathsf{client}}_{u'_i,w'_i,\mathcal{D}}$ denotes the view of a client (i.e., internal states and messages exchanged with the honest server) during the i-th execution of the protocol $\Pi^{\ell}_{\mathsf{query}}$ on client's input (u'_i, w'_i) and server's inputs $\mathcal{Z}_1, \dots, \mathcal{Z}_\ell, \mathsf{k}$ (generated through $\Pi^{\tau_n,\ell}_{\mathsf{init}}(\mathcal{D})$).
- **Server-side leakage:** The distributions $\mathbf{View}^{\mathsf{server}}_{\{(u'_i, w'_i)\}_{i\in[q]},\mathcal{D}} = (\mathbf{View}^{\mathsf{server}}_{u'_1,w'_1,\mathcal{D}}, \dots, \mathbf{View}^{\mathsf{server}}_{u'_q,w'_q,\mathcal{D}})$ and $\mathsf{S}_{\mathsf{server}}(\mathcal{D}, \mathsf{L}_{\mathsf{server}}(\mathcal{D}, \{(u'_i, w'_i)\}_{i\in[q]}))$ are computationally indistinguishable where $\mathbf{View}^{\mathsf{server}}_{u'_i,w'_i,\mathcal{D}}$ denotes the view of a server (i.e., internal states and messages exchanged with the honest client) during the i-th execution of the protocol $\Pi^{\ell}_{\mathsf{query}}$ on client's input (u'_i, w'_i) and server's inputs $\mathcal{Z}_1, \dots, \mathcal{Z}_\ell, \mathsf{k}$ (generated through $\Pi^{\tau_n,\ell}_{\mathsf{init}}(\mathcal{D})$).

C.1. Leakage in the Python implementation

The implementation of the τ function in [27] is problematic. Specifically, it can generate as output the input password, in which case it creates self-loops according to the graph-theoretic formulation. Another issue is that a single output set of τ function may contain the same similar password multiple times. Interestingly, the above points imply that this faulty implementation creates collisions even if there is a single breached credential (collisions are possible with itself). For example, when utilizing Das-r, a Type-0 collision that is also a self-loop reveals that the real password does not contain any characters that can be made uppercase (i.e., numbers, special characters, and uppercase). Conversely, the absence of such a collision informs us that the password includes lowercase characters.

Rank	Description	Example (input: "*passw0rd*")
1	Capitalize the string	"*PASSW0RD*"
2	Delete last character	"*passw0r*"
3	Delete last 2 character	"*passw0*"
4	Delete last 3 character	"*passw*"
5	Append "*0*"	"*passw0rd0*"
6	Prepend "*1*"	"*1passw0rd*"
7	Append "*a*"	"*passw0rda*"
8	Prepend "*0*"	"*0passw0rd*"
9	Delete second character	"*pssw0rd*"
10	Prepend "*a*"	"*apassw0rd*"

Table 2: Top-10 Das-r rules.

Appendix D.
Evaluation of the Filtering Attack

Next, we empirically demonstrate that τ-collisions reduce the entropy of the passwords in MIGP.

Data: We conduct the experiment on a randomly sampled subset (10%) of the 4iQ dataset [26]. Henceforth, we refer to this subset as $\mathcal{S}$. For each user in $\mathcal{S}$ with $e + 1$ credentials, we randomly select a password as the target one (i.e., w_{trgt}), while using the remaining e entries as the public passwords $[\bar{w}_1, \dots \bar{w}_e]$. Then, we utilize MIGP's setup protocol of Figure 1 to initialize the buckets. We test two different configurations of τ_n: Das-r$_{10}$ and Das-r$_{100}$. We focus on Das-r as it is the similarity function used in MIGP implementations.

Baseline attack strategy: We compare the success of the guessing strategy from Section 4 with a baseline attack. This captures an attack on the ideal functionality of MIGP with a leak function, where τ-collisions are not present. Hence, the attacker can only perform the third step (online phase) of the attack from Section 4. This means the attacker traverses the unfiltered search space until the target password w_{trgt} is guessed.

Search space: In our setup, we generate the search space as: $\mathcal{G} = \bigcup_{i=1}^{e} \tau_q(\bar{w}_i)$. In other words, we apply the similarity function to all known public passwords for the user and merge them into a single list that is then sorted based on password probability. This ensures the auxiliary information provided by the public password on the target one is exploited also in the baseline attack, providing a more fair comparison. To compute $\mathcal{G} = \bigcup_{i=1}^{e} \tau_q(\bar{w}_i)$, we use $q{=}323$, i.e., all the available rules for Das-r are applied. Additionally, we append all unique passwords in $\mathcal{S}$, sorted by decreasing frequency, to $\mathcal{G}$. Intuitively, this represents the best guessing ordering when auxiliary information on the target password is exhausted. Moreover, this ensures that the target w_{trgt} is always in the search space $\mathcal{G}$, simplifying the results presentation and interpretation.

Results: We run the baseline attack and collisions-based one on the same data and with the same initial conditions. In

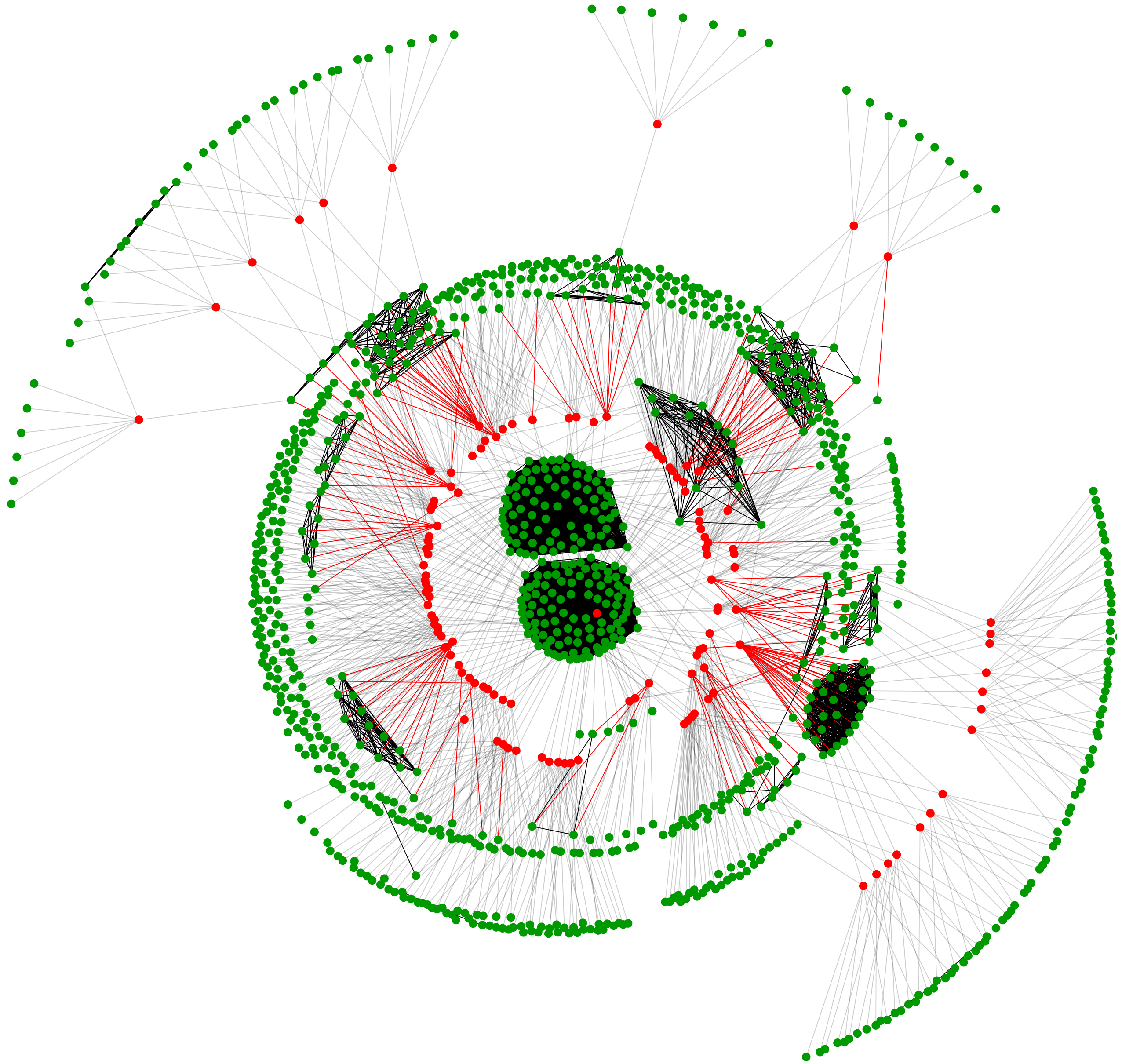


Figure 14: Example of (large) collision graph extracted from the $\texttt{MIGP}_{\texttt{CF}}$ production server. Red nodes depict real passwords, whereas green nodes are passwords generated through τ. Black edges represent type-1-induced collisions, whereas red ones are induced by type-0 collisions. Gray ones depict standard non-collision edges generated by the application of the τ function on real passwords during the $\texttt{MIGP}_{\texttt{CF}}$ setup phase. For the sake of clarity, we chose not to report the rank on edges.

the evaluation, we only report results for users that present τ-collisions. For users with no τ-collision, the two attacks are equivalent. The average number of guesses required by the attacks is listed in Table 3 (column (a)) for the different setups. For the τ-collisions-based attack, the average setup cost is reported in parentheses. Column (b) reports the reduction factor in the number of queries compared to the baseline. For the considered setups, the attack strategy that exploits τ-collisions results in a reduction of guesses by three orders of magnitude. Column (c) reports the relative size of the search space after pruning via τ-collisions; that is, after the offline phase, the search space is pruned to be $2 \cdot 10^{-5}\%$ of the original size.

We emphasize that the ordering of guessing list $\mathcal{G}$ may not be optimal, and overall better guessing strategies can be exploited to reduce the expected number of guesses (for instance, by exploiting the match of similar passwords [11]). However, **we employ the same search space for both the baseline and the attack outlined in Section 4**. This ensures that any advantage gained by the baseline attack also benefits

Attack strategy	(a) Avg. Num. Guesses	(b) Avg. Gain over Baseline	(c) Avg. Search space reduction
τ-collision-based (Das-r$_{10}$)	57.54 (+1.68)	3754.94	$1.92 \cdot 10^{-5}$
τ-collision-based (Das-r$_{100}$)	74.46 (+1.68)	2864.84	$2.13 \cdot 10^{-5}$
Baseline (Das-r)	216077.41	/	/

Table 3: Average number of queries required to recover the target password using two approaches: one that exploits τ-collisions (referred to as τ-collision-based), and another approach that does not utilize τ-collisions (referred to as baseline). These comparisons are conducted based on the guessing strategy outlined in Appendix D.

the τ-collisions-based attack strategy, thus maintaining the gap between the two attacks. Indeed, regardless of the guessing strategy used, an attacker would always be able to exploit τ-collisions to exclude passwords from the search space and reduce the expected number of guesses.

Algorithm 2: greedy-visit

```
Data: Collision graph: G : (V, E), Set of known passwords: V̄
   // traverse all the natural edges.
1  V̄ = V̄ ∪ BFS(V̄);
2  if |V̄| == |V| then
       // Stop, if everything has been guessed.
3      return V̄;
4  end
   // Set of nodes in G reachable from V̄.
5  U = {v ∈ V̄/V  s.t. ∃_{u∈V̄} : (v -r-> u) ∈ E};
   // Pick best candidate to guess.
6  v* = argmax_{v∈U} score(v, τ_n^{-1}(c))
   // Get known child nodes for v*.
7  c = {(r, u) | u ∈ V̄ ∧ (v* -r-> u) ∈ E};
   // Guess v* using τ_n^{-1}.
8  for g ∈ τ_n^{-1}(c) do
9      o = query_server(g);
10     if o == match then
           // v* guessed! Guess the next one.
11         return greedy-visit(G, V̄ ∪ g);
12     end
13 end
```

Appendix E. Security of MIGP 2.0

Theorem 2. *Given the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$, for every $q \in \mathbb{N}$, for every $\ell \in \mathbb{N}$, and for every similar password generator τ_n, the MIGP protocol composed of the pre-processing phase $\Pi^{\tau_n,\ell}_{\mathsf{new\text{-}init}}$ (Figure 12) and query phase $\Pi^{\ell}_{\mathsf{query}}$ (Figure 2) is $(q, \mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}})$-leakage secure (Definition 1) where $\mathsf{L}_{\mathsf{client}}$ and $\mathsf{L}_{\mathsf{server}}$ are the ideal leakage functions defined in Equation* (6)

Proof. Fix ℓ, τ_n, and q. Also, let $\mathcal{D}$ and $\{(u'_i, \tilde{w}'_i)\}_{i\in[q]}$ be the server's breached credentials dataset and the q credentials queried by a honest client, respectively. We prove the client-side and server-side leakage (Definition 1) individually.

(Server-side leakage). To prove the server-side leakage, we need to build a simulator $\mathsf{S}_{\mathsf{server}}$ that, on input $\mathcal{D}$ and $\mathsf{L}_{\mathsf{server}}(\mathcal{D}, \{(u'_i, \tilde{w}'_i)\}_{i\in[q]})$, simulates the view of an adversarial server. It is easy to see that $\mathsf{S}_{\mathsf{server}}$ can be straightforwardly constructed given the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$ since $\mathsf{S}_{\mathsf{server}}$ has access to all the information necessary to honestly execute the MIGP protocol over the q client's queries $\{(u'_i, \tilde{w}'_i)\}_{i\in[q]}$. Indeed, the only messages (of the server's view $\mathbf{View}^{\mathsf{server}}_{\{(u,'_i\tilde{w}'_i)\}_{i\in[q]},\mathcal{D}}$) that depends on the client's inputs (not known by the simulator) are the bucket indexes $\{\hat{\mathsf{H}}(u'_i) + 1\}_{i\in[q]}$. Still, the latter are part of the leakage function $\mathsf{L}_{\mathsf{server}}(\mathcal{D}, \{(u'_i, \tilde{w}'_i)\}_{i\in[q]})$ (see Equation (6)). Note that we do not need to simulate the OPRF messages since the latter is modeled as an ideal functionality $\mathcal{F}_{\mathsf{oprf}}$, i.e., the simulator only needs send to $\mathcal{F}_{\mathsf{oprf}}$the sampled OPRF key k. Also, note that $\mathcal{F}_{\mathsf{oprf}}$ does not return any output to the server.

(Client-side leakage). Consider the following simulator $\mathsf{S}_{\mathsf{client}}$ that, given the ideal functionality $\mathcal{F}_{\mathsf{oprf}}$ and inputs $\{(u'_i, \tilde{w}'_i)\}_{i\in[q]}$ and $\mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u'_i, \tilde{w}'_i)\}_{i\in[q]})$ (see Equation (6)), simulates the client's view $\mathbf{View}^{\mathsf{client}}_{\{(u,'_i\tilde{w}'_i)\}_{i\in[q]},\mathcal{D}}$ as follows:

1) Initialize the buckets $\mathcal{Z}_{\hat{\mathsf{H}}(u'_1)+1} = \bot, \ldots, \mathcal{Z}_{\hat{\mathsf{H}}(u'_q)+1} = \bot$. Note that this are the only buckets queried by the adversarial client.
2) For every $i \in [q]$, the simulator sends $(u'_i, \tilde{w}'_i)$ to $\mathcal{F}_{\mathsf{oprf}}$ and receive the output y_i. If $\mathsf{result}_i = \mathsf{similar}$ set $y_i = y_i \oplus 1$ where result_i is the output (contained in the leakage function $\mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u'_i, \tilde{w}'_i)\}_{i\in[q]})$) of the MIGP protocol of the i-th client query $(u'_i, \tilde{w}'_i)$. If $\mathsf{result}_i \neq \mathsf{none}$, add y_i to the bucket $\mathcal{Z}_{j+1}$ where $j = \hat{\mathsf{H}}(u'_i)$.
3) For $i \in [q]$, let t_i (which is part of the leakage function $\mathsf{L}_{\mathsf{client}}(\mathcal{D}, \{(u'_i, \tilde{w}'_i)\}_{i\in[q]})$) be the number of original credentials mapped to the bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$, and n be the publicly known number of similar passwords generated by τ_n. For $i \in [q]$, set $s_i = t_i \cdot n$ if $n > 0$, and $s_i = t_i$ otherwise (i.e., $n = 0$).
4) For every $i \in [q]$, add randomly sampled strings (of the same length of the OPRF output) to the bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$ until it reaches the size s_i.[10] Finally, shuffle the entries of the bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$.
5) Simulate the client's view $\mathbf{View}^{\mathsf{client}}_{\{(u,'_i\tilde{w}'_i)\}_{i\in[q]},\mathcal{D}}$ by honestly following the protocol while using the elements computed above. Indeed, the information computed in Items 1, 2 and 4 are sufficient to simulate the view of the client. In particular, for every $i \in [q]$, (i) when the client invokes $\mathcal{F}_{\mathsf{oprf}}$ on $(u'_i, \tilde{w}'_i)$, the simulator returns y_i and, (ii) when the client asks for the bucket $j + 1 = \hat{\mathsf{H}}(u'_i) + 1$, the simulator returns the simulated bucket $\mathcal{Z}_{j+1}$.

What we need to show is that the entries of the queried buckets are simulated correctly.

10. Observe that the value t_i is not sufficient to simulate the buckets of the old MIGP protocol of Pal et al. [11]. This is because the number of entries s_i of the bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$ may change according to the number of collisions. Hence, to prove the security of [11], the number of entries of each queried bucket must be part of the leakage function. However, when collisions occur, this leaks sensitive information regarding the similar passwords.

First, for every $i \in [q]$, the queried bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$ has cardinality s_i which, in turn, is defined as either $s_i = t_i \cdot n$ (if $n > 0$) or $s_i = t_i$ (if $n = 0$). Hence, the cardinalities of the queried buckets are identical as that of an honest protocol execution.

Second, if a client's query $(u'_i, \tilde{w}'_i)$ yields either match or similar, then the matched entry contained in the corresponding bucket $\mathcal{Z}_{\hat{\mathsf{H}}(u'_i)+1}$ is computed as in the original protocol.

Lastly, each unmatched entry of each bucket is simulated by sampling it at random (instead of evaluating the PRF on the correct credential/dummy value). By using a hybrid argument, we can easily demonstrate that these simulated (random) entries are indistinguishable from the real PRF evaluations. This follows from the security of the PRF. The proof is standard so we omit it. □

By combining Theorem 2 and Theorem 1, we obtain the following corollary.

Corollary 1. *Let* H_1 *and* H_2 *be two hash functions modeled as random oracles. Also, let* q_1 *be the number of random oracle queries submitted to* H_1. *If* $(q + q_1, q)$*-one-more DH assumption [24, Section 3] holds for* $\mathbb{G}$*, then for every* $\ell \in \mathbb{N}$*, and for every similar password generator* τ_n*, the* `MIGP` *protocol composed of the pre-processing phase* $\Pi^{\tau_n,\ell}_{\mathsf{new\text{-}init}}$ *(Figure 12) and query phase* $\Pi^{\ell}_{\mathsf{query}}$ *(Figure 2) is* $(q, \mathsf{L}_{\mathsf{client}}, \mathsf{L}_{\mathsf{server}})$*-leakage secure (Definition 1) in the random oracle model where* $\mathsf{L}_{\mathsf{client}}$ *and* $\mathsf{L}_{\mathsf{server}}$ *are the ideal leakage functions defined in Equation* (6).