Privacy-Preserving Password Cracking: How a Third Party Can Crack Our Password Hash Without Learning the Hash Value or the Cleartext

Norbert Tihanyi, Tamas Bisztray, Bertalan Borsos, and Sebastien Raveau

***Abstract*—Using the computational resources of an untrusted third party to crack a password hash can pose a high number of privacy and security risks. The act of revealing the hash digest could in itself negatively impact both the data subject who created the password, and the data controller who stores the hash digest. This paper solves this currently open problem by presenting a Privacy-Preserving Password Cracking protocol (3PC), that prevents the third party cracking server from learning any useful information about the hash digest, or the recovered cleartext. This is achieved by a tailored anonymity set of decoy hashes, based on the concept of predicate encryption, where we extend the definition of a predicate function, to evaluate the output of a one way hash function. The protocol allows the client to maintain plausible deniability where the real choice of hash digest cannot be proved, even by the client itself. The probabilistic information the server obtains during the cracking process can be calculated and minimized to a desired level. While in theory cracking a larger set of hashes would decrease computational speed, the 3PC protocol provides constant-time lookup on an arbitrary list size, bounded by the input/output operation per second (IOPS) capabilities of the third party server, thereby allowing the protocol to scale efficiently. We demonstrate these claims both theoretically and in practice, with a real-life use case implemented on an FPGA architecture.**

***Index Terms*—Password security, hash cracking, k-anonymity, privacy enhancing technology, data privacy;**

I. Introduction

PASSWORDS are the most widely used mechanism for knowledge based user authentication [1]. Although tech firms such as Apple, Google and Microsoft are pushing for a *passwordless future* [2], the transition will not impact every domain of identity management as alternatives often fail to provide a set of benefits already present in passwords [3]. Consequently, passwords will remain a part of our every day life, especially in areas where it is not feasible to employ technologies required for passwordless authentication [3], [4]. Therefore, related security and also *password management* practices should be kept up to date [5].

Tamas Bisztray received funding from the Research Council of Norway (forskningsradet) under Grant Agreement No. 303585 (CyberHunt project), the EU Connecting Europe Facility (CEF) programme under Grant Agreement No. INEA/CEF/ICT/A2020/2373266 (JCOP project) and the Horizon Europe programme under Grant Agreement No. 101070586 (PHOENi2X project). The views and opinions expressed herein are those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Norbert Tihanyi was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through the TKP2021-NVA Funding Scheme under Project TKP2021-NVA-29.

Storing passwords in cleartext poses a high security risk as attackers upon compromising the system could learn not only the passwords, but the *password choosing patterns* of individuals [6]. A possible mitigation is to only store the hash of the password [7]. Recovering the cleartext from a hash digest is very resource intensive and is often infeasible with sufficiently long and complex passwords. Unfortunately, *password choosing habits* of individuals often lack such characteristics [8]. There are well-known software tools such as *hashcat* or *John the Ripper* that can be used to perform password cracking attacks efficiently. Advancements in modern *GPU*s and *FPGA*s have rendered previously popular hashing algorithms obsolete. The same is true for *password policies* regarding recommended length, character set and complexity [9], [10], [11], [12].

Testing password security can be conducted for *legitimate purposes*. The main motivation behind this paper originates from a penetration testing project that took place in 2020. During a penetration testing engagement the *Red Team* was able to retrieve an *NTLM* hash of an important service account. It was known, that all service account passwords are randomly generated 9 character long strings containing uppercase and lowercase letters plus numbers. Because of the sensitive nature of the project, the *Red Team* could not share the exact value of the *NTLM* hash. According to the signed contract, the team was not allowed to reveal any cleartext passwords from the engagement to third parties. There are many small organizations and freelancers conducting *penetration testing* and *red teaming* activities, who would rely on such services, but are prevented from doing so for privacy considerations. If the *Red Team* transfers a password hash, the cracking server learns the exact value of the hash digest and if the cracking is successful the corresponding *cleartext password* as well.

As the example shows, an entity might have a legitimate reason to crack a password hash, but could lack the computational capacity to perform the cracking process within a reasonable time. One solution would be to use a cloud service called *"password cracking as a service"* (*PCaaS*), to utilize the resources of a third party. This can be concerning both from a security, and a privacy perspective. As an example, hashes can be used in *pass the hash* attacks [13] without having to crack the password itself. Moreover, if the cleartext is recovered the third party might learn privacy sensitive information as individuals often embed *personally identifying information* (*PII*) when constructing their passwords [14], [15], [16].

We would like to find answers to the following questions:

(1) *can we use the resources of an untrusted third party for the cracking process without revealing the exact value of the target hash digest* and (2) *if the hash is cracked, can we prevent information disclosure on the recovered cleartext*. In theory, this could be prevented by employing *homomorphic encryption* [17], [18], where the third party performs operations on encrypted data. The domain of *homomorphic encryption* has been the subject to a lot of attention and there were many achievements that brought us closer to practical applications in the last decade [19]. Unfortunately, as of today no such algorithm has the efficiency that could allow us to take advantage of this technology for the realization of our goals [20], [18]. To the best of our knowledge prior to this publication privacy-preserving password cracking protocols have not been considered and documented in scientific literature. Such a protocol not only satisfies the client's security and privacy needs upon using *PCaaS*, but can also assist in compliance with *data protection and privacy regulation*, where the data controller can document justifiable and explainable privacy protection measures upon using this *privacy enhancing technology* (*PET*). The main contributions of this paper can be summarized as:

1) We introduce the idea of a *Privacy-Preserving Password Cracking* (3PC) protocol
2) We extend the concept of *predicate functions* to evaluate decoy hash digests that make up our anonymity set, thereby resolving data transfer and performance issues
3) We show that the *probabilistic information* the *cracking server* learns is not practically useful, and the protocol is resistant against attacks and *foul play*
4) The protocol provides *plausible deniability*, where the client can claim to have aimed for a different target
5) Demonstrations of the implemented protocol, both with *"toy examples"*, and realistic use cases, showing the scalability and efficiency of the protocol
6) The protocol ensures the client, using *proof of work*, that the server exhausted the agreed search space

The paper will be structured as follows: Section II overviews related literature, while Section III discusses the application of predicate functions, and introduces the 3PC protocol with two toy examples. Section IV presents the security and privacy aspects of 3PC, followed by Section V where we showcase a real-life example, implemented on a FPGA architecture. Section VI concludes our results.

II. Related literature

The idea to hide valid cryptographic keys and hashes among fake ones appeared in literature 20 years ago. Arcot [21] systems used a list of junk RSA private keys to protect the original private key. An attacker who tries to crack the key container will recover many plausible private keys, but will not be able to tell which one is the original until he tries each to access resources via an authentication server. In 2010 Bojinov et al. [22] introduced the *Kamouflage system*, a theft-resistant password manager which generates sets of *decoy passwords*.

Juels and Ristenpart introduced the *honey encryption* scheme [23], designed to produce a ciphertext which, when decrypted with incorrect keys, produces plausible-looking but bogus plaintexts called *honey messages*. This makes it impossible for an attacker to tell when decryption has been successful. In 2013 Jules and Rivest proposed a simple method [24] for improving the security of hashed passwords. The system in addition to a real password with each user's account associates some additional *honeywords* (false passwords). The attempted use of a honeyword triggers an alarm.

Compromised Credential Checking (C3) services, such as HaveIBeenPwned and Google Password Checkup can reveal if user credentials appear in known data breaches [25], [26]. In this setup, clients can provide an N-bit prefix of the hashed password. The server then returns all recorded breached passwords that match this prefix. The client then conducts a local final check to confirm if there is a match. However, in this scenario sharing a small hash prefix of user passwords can significantly increase the effectiveness of remote guessing attacks [27]. The anonymity set it provides will can only come from passwords of the same prefix, which might not be an adequately large set. On the other hand if the prefix is too small, it might return a lot of results, which in this setting would be more difficulty to download for the user. To counter this issue Li et al. developed and tested two new protocols that offer stronger password protection and are practical for deployment [27] . This approach of only sharing a part of the password hash is useful, which we upgraded for the 3PC protocol to allow the creation of a more fine tuned anonymity set.

For our protocol, understanding what distribution passwords follow will be also important. As Hou in [28] underlined, several research papers incorrectly assume that passwords follow a *uniform distribution*. In large real password data sets certain popular passwords are used by multiple users [29]. In [30] Blocki et al. presents how a *frequency list* can be created based on this observation, essentially ranking passwords from most frequent to least frequent. Inspired by Malone [31], Wang et al. showed that such passwords follow a *CDF-Zipf* distribution. These results apply to large user generated data sets where recurrence of passwords can be observed. If we want to analyze a large corpus of machine generated passwords results may be different. Such sets are usually created as a dictionary for password cracking. If the generation algorithm simply outputs random strings, the resulting data set by definition follows a uniform distribution. However, password generation based on real user passwords is shown to perform better. The most popular techniques are: *Rule-based dictionary* attacks, *probabilistic context free grammars* (*PCFG*s), *Markov models*, and machine learning techniques [32], [33]. The likelihood of each password can be calculated if we assign a probability to each production rule. However, determining the probability distribution over such data sets is not a straightforward task and is outside the scope of this research. It is different from assessing word-list quality [34], [35], or individual password strength [36], [37], [38], [39], as both are a separate line of research. As Aggarwal et al. shows [32], *PCFG parse trees* are usually *ambiguous* which means, there can be multiple ways to produce a single word. Furthermore, if a capable adversary aims to determine the

probability distribution of such data sets but doesn't know the original creation mechanism, they can get a completely different ranking order if for example a *Markov model* is used to rank a data set created by *PCFG*. Not to mention that each model is fully dependent on the original training data. Note, that Bonneua warns against the use of traditional metrics such as *Shannon entropy* and *guessing entropy* for evaluating large password data sets [29].

For hiding the original hash digest among decoy hashes we will utilize a concept similar to predicate encryption. In predicate encryption [40], a ciphertext is associated with descriptive attribute values υ in addition to a plaintext p, and a secret key is associated with a predicate f. Decryption returns plaintext p if and only if $f(\upsilon) = 1$. In our case there will be no secret key associated with the predicate, as hash functions cannot be reversed in the same way as an encryption function but as the concept is similar we use this terminology.

III. The 3PC protocol

First, we review some important cryptographic primitives and their properties we rely on, which is followed by the main steps of the 3PC protocol. Next, we introduce how a predicate function can evaluate the output of a hash function. Finally, we examine how the protocol works in action by showcasing two *"toy examples"* serving as proof of concept.

A hash function is a computationally efficient deterministic function mapping from an arbitrary size input (*message space*) into a fixed size output of length l (*digest space*). A hash function is called a *cryptographic hash function* if the usual security properties are satisfied: *pre-image resistance* (or one-way property), *second pre-image resistance*, and *collision resistance*. We also require that a hash function exhibits the *avalanche effect* and satisfies the *random oracle model* where the output is uniformly distributed. It is important to note that theoretically there are infinitely many collisions for a hash function, but these should be difficult to find, meaning, there is no explicit or efficient algorithm that can output a collision.

A. Protocol Design and Notations

For the rest of the paper the following notation will be used:

- t: The *target hash* which the client wants to crack
- Σ: the finite set of hexadecimal symbols {0-F}, where the finite sequence of symbols of length l over the alphabet Σ is denoted by Σ^l
- Θ^*: For any alphabet Θ, the set of all strings over Θ
- h: A cryptographically secure *hash function* where Θ^* serves as the input space for h, i.e., $h : \Theta^* \to \Sigma^l$
- $\mathcal{X}_\upsilon$: Set of *decoy hashes*, serving as an anonymity set for the target hash, where $t \in \mathcal{X}_\upsilon$
- N_υ: The desired number of *decoy hashes* the client wants as the size of $\mathcal{X}_\upsilon$
- υ: A vector that describes every hash in $\mathcal{X}_\upsilon$, and $\upsilon \in \Sigma^{2l}$
- $\mathcal{DS}$: The *cracking data set*, selected by the client and used by the cracking server to recover hashes from $\mathcal{X}_\upsilon$
- $\mathcal{CS}$: *Candidate password set*, a set of password-hash pairs the server managed to crack from $\mathcal{X}_\upsilon$

A high level pseudo code of the 3PC protocol can be seen in Algorithm 1. Our two parties are the client and the server, as indicated below in Figure 1. The client wants to crack the *target hash*, and recover the cleartext password. First, the client checks using REQ-H, if the server side architecture can efficiently crack the desired hash function. The server responds with ACK-H, specifying the hash cracking speed in hash-rate/secundum (H/Sec). Based on this information, using DEF-DSR the client selects a *data set* $\mathcal{DS}$, and specifies the desired number of *candidate passwords* which is denoted by r. Meaning, the *candidate set* $\mathcal{CS}$ should contain approximately r elements. Next, the client creates a vector υ that defines the set of decoy hashes, such that $|\mathcal{X}_\upsilon| \approx N_\upsilon$ and $t \in \mathcal{X}_\upsilon$ is satisfied (CLC-NV, GEN-V). We note, that creating a vector υ which defines exactly N_υ elements is linked to the theory of y-smooth numbers. As a consequence N_υ can only be approximated when υ is created. The *cracking data set* which can be a selection of dictionary words with mangling rules, brute force rules, etc., and the vector υ is transferred to the server (SND-P). The server starts hashing every password in $\mathcal{DS}$ (CRK-XV). If a resulting hash digest is in the set defined by the vector υ (this is checked by the predicate function, not by a direct comparison), this hash digest along with the corresponding *cleartext password* is added to the list of *candidate passwords*. After exhausting the search space the server sends $\mathcal{CS}$ back to the client (SND-CS). To evaluate if the cracking was successful, the client simply checks the *candidate list* to see if $(s, t) \in \mathcal{CS}$ for some $s \in \mathcal{DS}$ (CHK-CS).

Essentially, the server tries to crack every password in the *decoy set*, never knowing which is the *target hash*. We want to underline, that the server at no point has any information on whether t is cracked or remains uncracked, which is of utmost importance. This is due to the protocol design, where it is close to impossible to crack all hashes in the *decoy set*.

Algorithm 1 3PC protocol Client-Server exchange

```
procedure 3PC(t, r, DS)
    REQ-H(h)                          ▷ Request hash information
    i ← ACK-H(h)                      ▷ Acknowledge hash info request
    DS, r ← DEF-DSR(i)                ▷ Define data set and r
    N_υ ← CLC-NV(r, |DS|)             ▷ Calculate N_υ
    υ ← GEN-V(t, N_υ)                 ▷ Generate decoy hashes
    SND-P(DS, υ)                      ▷ Send parameters to server
    CS ← CRK-XV(υ, DS)                ▷ Cracking decoy set
    SND-CS(CS)                        ▷ Send candidate passwords
    CHK-CS(CS, t)                     ▷ Check target hash in candidate set
end procedure
```

The decoy hash set will serve as an anonymity set, providing *k-anonymity* for the *target hash digest*, and the server can only have a $1/|\mathcal{X}_\upsilon|$ chance to guess t when observing $\mathcal{X}_\upsilon$ alone, which in practice can easily be around 10^{-70}. The decoy hashes at the same time as protecting t, can ensure the privacy protection of the *pre-image* of t, regardless of whether it is cracked or not. This is a really important claim. As the server recovers many plausible cleartext passwords these will serve as an anonymity set for the *pre-image* of t. Using the *security*

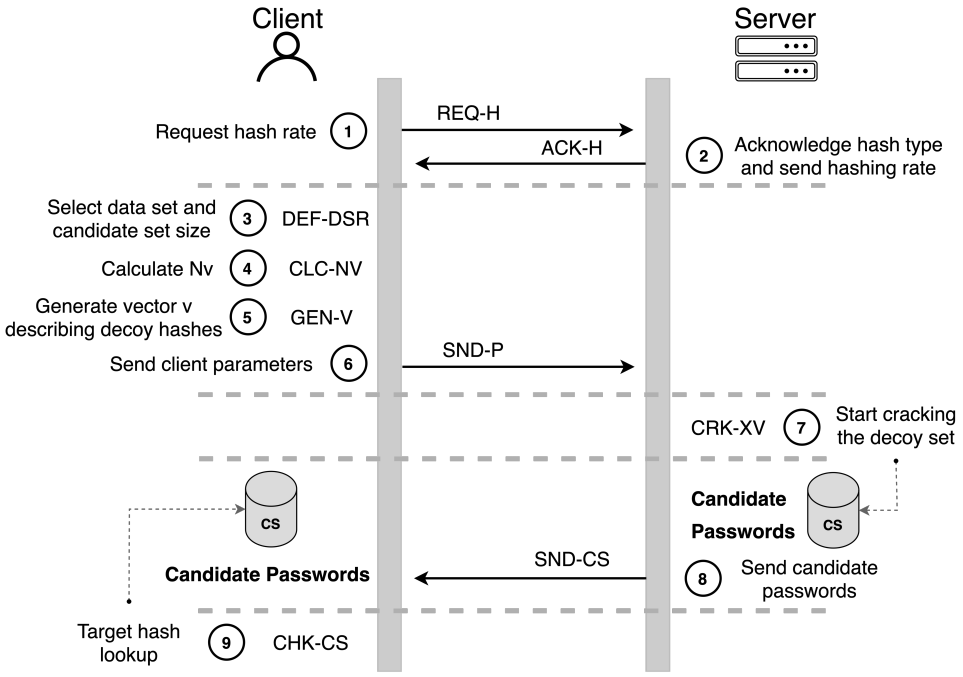

Fig. 1. Client-Server interaction. A simplified version on how information exchange is made between the client and the third party server

parameters $\mathcal{DS}, N_v$ and r, one can calculate and adjust the probabilistic knowledge of the server about both the *target hash*, and the *cleartext password* that created the hash. This knowledge can be tailored based on the calculated privacy needs of the client, or the resources of the third party.

Privacy demands could require ever larger sets of decoy hashes, where transferring $|\mathcal{X}_v|$ unique hashes would be a serious bottleneck. In addition to the privacy challenges, this problem is also solved by the 3PC protocol, as the decoy set is transferred in a compact form described by the vector v. This, at the same time allows the *predicate function* to remove limitations on the amount of hashes we can efficiently handle. The client or the server side never needs to write the entire $\mathcal{X}_v$ set to disk, which could be a bottleneck, as in realistic scenarios it can contain 10^{70} unique hash digests or more. This in itself is an important contribution, allowing the CRK-XV process to finish regardless of the number of decoy hashes in $\mathcal{X}_v$ as we will show in Lemma III.1.

B. Establishing the Security Parameters

In the following, we examine how the client can predictably control the number of hashes that are cracked from the *decoy set* $\mathcal{X}_v$. Since the *pre-images* of the decoy hashes are fully random, this "cracking success rate" will be determined only by the size of the *cracking data set* $|\mathcal{DS}|$ and $|\mathcal{X}_v|$. What passwords are present in the *data set* will have no influence on this calculation. To understand this, we need to look at how password hashes are distributed over the *co-domain* of h. When elements of $\mathcal{DS}$ are hashed, the output is spread randomly over Σ^l as shown in Figure 2. This means that by picking a *hash digest* randomly from the output space of h, the probability of selecting one that has a *pre-image* from $\mathcal{DS}$ is simply $|\mathcal{DS}|/|\Sigma^l|$. As there is no correlation between the *cleartexts* and the *hashes*, by observing only the output space, selecting any of the hash digests carries a $|\mathcal{DS}|/|\Sigma^l|$ chance to have a *pre-image* in $\mathcal{DS}$. By picking k hash digests, the expected number of them having a *pre-image* in $\mathcal{DS}$ is k times $|\mathcal{DS}|/|\Sigma^l|$.

This expected number is the same whether k hashes are picked from $\mathcal{X}_v$, or by selecting k randomly from Σ^l. Thus we can choose from $\mathcal{X}_v$, the set described by v, where all hashes by definition satisfy the predicate function, which is introduced in Section III-C. Now, we can calculate the expected number of *candidate passwords* (r) we get from a given $\mathcal{X}_v$ when cracking with a *data set* with size $|\mathcal{DS}|$:

$$N_v \frac{|\mathcal{DS}|}{|\Sigma^l|} \approx r \tag{1}$$

This formula is used by the client in CLC-NV and GEN-V, to determine N_v, and then to check if vector v can ensure approximately r candidate passwords in $\mathcal{CS}$. If we want to have more *decoy hashes* in $\mathcal{X}_v$ which upon getting cracked will reveal a *pre-image* in $\mathcal{DS}$, we simply need to increase $|\mathcal{X}_v|$.

Visually, we can see this from in Figure 2, that as we expand the $\mathcal{X}_v$ set, it will "gobble up" more of the dots representing the hashed $\mathcal{DS}$ words. The *target hash* is only cracked if the *cracking data set* used contains a *pre-image* for t. Increasing $\mathcal{X}_v$ will achieve two things: it produces more candidate passwords for a given *data set* selected, and it decreases the probability of a correct guess on the *target hash digest*. Due to the random distribution of the hash function the security parameter r is just an expected value. Although we can reliably estimate it, we will not know the exact number of candidates that will be returned in the *candidate set* until the end of the cracking process. Also, the *server* will never

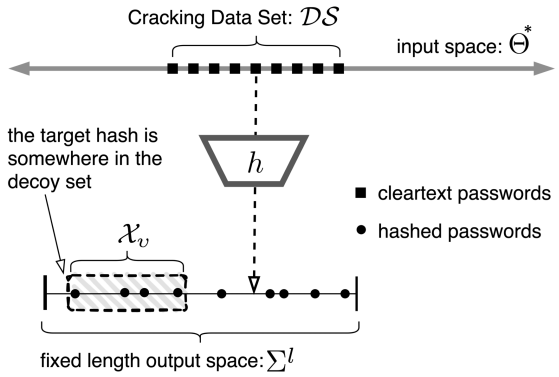

Fig. 2. Hashing the cracking data set and selecting an anonymity set $\mathcal{X}_\upsilon$ for a target hash t

know if a certain *data set* has found the *target hash*, or it is still uncracked.

C. Mathematical foundations of 3PC

An important contribution of this paper which we next introduce, is the extension of the concept of predicate encryption to one way hash functions. Following that, we present the underlying mathematical problem that needs to be solved during the step GEN-V.

Definition III.1 (**Predicate Function**). *The set of all strings of length l over the alphabet Σ is denoted by Σ^l, where our alphabet will be the set of hexadecimal characters. For a vector $\upsilon = (\upsilon_1, \ldots, \upsilon_{2l}) \in \Sigma^{2l}$ we define a predicate function P_υ over Σ^l as follows: Given an input vector $x = (x_1, \ldots, x_l) \in \Sigma^l$, $P_\upsilon(x) = 1$ if $(\upsilon_{2i-1} \leq x_i \leq \upsilon_{2i})$ for all $1 \leq i \leq l$, $P_\upsilon(x) = 0$ otherwise.*

For example if $x = (3, C) \in \Sigma^2$ and $\upsilon = (2, 5, C, D) \in \Sigma^4$, we have $P_\upsilon(x) = 1$ since $2 \leq x_1 \leq 5$ and $C \leq x_2 \leq D$ are satisfied for our vector υ. For a given $\upsilon \in \Sigma^{2l}$, let $\mathcal{X}_\upsilon$ be the set of all x vectors which satisfy $P_\upsilon(x) = 1$. The number of elements (cardinality) of $\mathcal{X}_\upsilon$ is denoted by $|\mathcal{X}_\upsilon|$. The size of $\mathcal{X}_\upsilon$ can be calculated from the definition of υ by the following formula:

$$|\mathcal{X}_\upsilon| = \prod_{i=1}^{l} max\{\upsilon_{2i} - \upsilon_{2i-1} + 1, 0\} \tag{2}$$

Clearly, we have a set of $|\mathcal{X}_\upsilon|$ different vectors making up $\mathcal{X}_\upsilon$, i.e.,

$$\mathcal{X}_\upsilon = \begin{bmatrix} x^1 = (x_1^1, \ldots, x_l^1) \\ x^2 = (x_1^2, \ldots, x_l^2) \\ \vdots \\ x^{|\mathcal{X}_\upsilon|} = (x_1^{|\mathcal{X}_\upsilon|}, \ldots, x_l^{|\mathcal{X}_\upsilon|}) \end{bmatrix}$$

Note, that $|\mathcal{X}_\upsilon| = 1$ iff $\{\upsilon_{2i} = \upsilon_{2i-1} : \forall 1 \leq i \leq l\}$ and $|\mathcal{X}_\upsilon| = 0$ iff $\exists i$ such that $\upsilon_{2i-1} > \upsilon_{2i}$. Similarly, if for all $1 \leq i \leq l$, υ_{2i} is the greatest element of Σ and υ_{2i-1} is the least element of Σ, then $|\mathcal{X}_\upsilon| = |\Sigma^l|$.

Lemma III.1. *The calculation of $P_\upsilon(x)$ is independent of the cardinality of $\mathcal{X}_\upsilon$.*

Proof. From condition $\{\upsilon_{2i-1} \leq x_i \leq \upsilon_{2i} : 1 \leq i \leq l\}$, $P_\upsilon(x)$ can be calculated in constant time which in the worst case is $2 \cdot l$ comparison. □

We say an integer N is y-smooth if N has no prime divisors greater than y.

Lemma III.2. *$|\mathcal{X}_\upsilon|$ is always a 13-smooth number.*

Proof. For any $\upsilon = (\upsilon_1, \ldots, \upsilon_{2l}) \in \Sigma^{2l}$ vector it is trivial that $0 \leq \upsilon_i \leq |\Sigma|$ for all $1 \leq i \leq l$. From equation (2), the prime factors of $|\mathcal{X}_\upsilon|$ are always less than $|\Sigma| = 16$, from which the lemma follows. □

Corollary III.1. *The prime factorization of every $|\mathcal{X}_\upsilon|$ number is of the form $2^A \times 3^B \times 5^C \times 7^D \times 11^E \times 13^F$, where $A, \ldots, F \in \mathbb{N}$.*

During step CLC-NV, an appropriate N_υ is calculated from the size of the *data set* $|\mathcal{DS}|$, and the desired number of *candidate passwords* (r) using equation 1,

$$N_\upsilon = \frac{r|\Sigma^l|}{|\mathcal{DS}|} \tag{3}$$

In the ideal scenario, the client can create a vector υ that defines the set of decoy hashes $\mathcal{X}_\upsilon$ with exactly N_υ elements, i.e., $|\mathcal{X}_\upsilon| = ||N_\upsilon||$ where $||\cdot||$ denotes the nearest integer. However, N_υ is not necessarily 13-smooth, so we need to find a 13-smooth number close to the original N_υ. We can take the logarithm of both sides of the equation $|\mathcal{X}_\upsilon| = ||N_\upsilon||$, i.e.,

$$\log(2^A \times 3^B \times 5^C \times 7^D \times 11^E \times 13^F) = \log(N_\upsilon)$$

from which we get

$$\begin{gathered} A\log(2) + B\log(3) + C\log(5) + \\ D\log(7) + E\log(11) + F\log(13) \\ -\log(N_\upsilon) = 0 \end{gathered} \tag{4}$$

This is a 7-variable *integer relation problem*, and can be viewed as a special subset sum optimization problem. Using the Lenstra, Lenstra and Lovász (LLL) basis reduction algorithm [41], or the $PSLQ$ algorithm, one can find the fitting integers. We note, that not all 13-smooth numbers are suitable for our needs, therefore we need to run LLL for different inputs with varying beta reduction parameter. For example, although $N_\upsilon = 5^{l+1}$ is 13-smooth, it is not possible to divide the factors into l slots, where $\{\upsilon_{2i} - \upsilon_{2i-1} \leq 16 : \forall 1 \leq i \leq l\}$ stays true. Modern LLL implementations can solve integer relation problems with more than 500 variables. In our case we can always get an appropriate result in polynomial time (in a few seconds on an average computer).

As previously stated, our objective is to crack the *target hash* and hide it in a *decoy set* represented by υ. As $P_\upsilon(t) = 1$ must be satisfied we construct υ from the *target hash*. Depending on how we adjust the degree of freedom in vector υ, it can allow $|\mathcal{X}_\upsilon| - 1$ decoy hashes to satisfy the function P_υ. These vectors will serve as decoy hashes.

Now we possess the minimum knowledge to understand the protocol. First, we present 3PC through two toy examples, before moving on with the security analysis.

D. Toy Examples

Toy Example 1 - Dictionary attack: The *Rock-You* database contains $14\,344\,391$ unique passwords which will be our $\mathcal{DS}$ for this example [6]. The target hash is $t = (C, 6, B, F, A, B, A, 2) \in \Sigma^8$, where t is the *CRC-32* output of our unknown password. CRC-32 is not a cryptographically secure hash function however, it's presentable output size is more suitable for a toy example. After requesting hashing information (REQ-H) and getting acknowledgement (ACK-H) from the server, the client is looking to create a vector v that defines an $\mathcal{X}_v$ set, such that after the cracking process approximately 20 candidate passwords are returned in $\mathcal{CS}$. Knowing the size of the dictionary $|\mathcal{DS}| = 14\,344\,391$, and the security parameter $r = 20$, we can calculate N_v. This means that after $14\,344\,391$ hash calculations the *client* expects approximately 20 candidates from the *Rock-You* database to fall into $\mathcal{X}_v$. To satisfy this requirement, step CLC-NV uses formula 1 to calculate N_v (the expected size of $\mathcal{X}_v$),

$$N_v = \frac{r|\Sigma^l|}{|\mathcal{DS}|} = \frac{20 \cdot 16^8}{14\,344\,391} \tag{5}$$

The result is $N_v \approx 5988.36$. Having calculated N_v, we need to construct a vector v where $P_v(t) = 1$ and $|\mathcal{X}_v|$ is around this size. Note, that 5988 is not 13-smooth as $5988 = 2^2 \cdot 3 \cdot 499$. Using the GEN-V algorithm a suitable v can be selected: $(C, F, 2, 6, A, B, D, F, 9, F, B, B, A, A, 0, 6) \in \Sigma^{16}$. This defines an $\mathcal{X}_v$ decoy set with 5880 elements, which satisfies Lemma III.2, Corollary III.1, and is close to N_v. Moreover, the degrees of freedom can be divided into l slots, while satisfying $v_{2i} - v_{2i-1} \leq 16, \forall\, 1 \leq i \leq 8$. As such, it will produce nearly the same number of expected candidates:

$$|\mathcal{X}_v| \frac{|\mathcal{DS}|}{|\Sigma^l|} = 5880 \frac{14\,344\,391}{16^8} \approx 19.63 \tag{6}$$

After receiving SND-P the server starts hashing every element in the dictionary, and it looks for matches in the set $\mathcal{X}_v$. Here lies one of the major benefits of the protocol: the server never has to make $|\mathcal{X}_v|$ comparisons for each password. After calculating the hash, the server simply checks if $P_v(h(s)) = 1$, $\forall s \in \mathcal{DS}$. In this toy example we expect around 20 passwords from $\mathcal{DS}$ to satisfy P_v. The *hash-cleartext* pairs where $P_v(h(s)) = 1$ can be seen in Table I.

TABLE I
TOY EXAMPLE 1: GENERATED CANDIDATE HASHES

#	password	CRC-32	#	password	CRC-32
1	tangan	C5AEFBA5	11	0849831211	D4BFDBA6
2	hornbyneho	C3AEFBA0	12	lumpibuniz	E3ADEBA4
3	28707adnen	C4AE9BA4	13	sweep21	E3BEEBA6
4	lisa1842	C4BECBA2	14	577672	E3BEFBA5
5	**0BChrist**	**C6BFABA2**	15	050462654	E5BDBBA1
6	sapphire24	C6BFDBA2	16	horses33	F2BEBBA0
7	Kissarmy1!	D2ADFBA6	17	zuzuloka	F3AECBA1
8	whateva89	D2BE9BA3	18	ms.jackson2008	F3BEDBA5
9	keno333_	D3BDABA3	19	a2gfamilymaster	F5BDDBA5
10	bighottie	D4AEABA2	20	alana123456789	F6ADABA2

From the returned candidate set the client can see that the *target hash* is cracked and the password is `"0BChrist"`. The server was expecting around 20 candidates from the dictionary, but has no idea if the cracking was successful. It is also important to ask the question if the passwords are equally likely in this case. This is thoroughly examined in SectionIV. For the sake of this example we selected a password from *Rock-You* to begin with, hence it was present in the candidate set.

Toy Example 2 - Brute force: A more realistic case is to use a cryptographically secure hash function. In this scenario the *client* has the extra knowledge, that the hash digest hides an *8 digit PIN code*. The client must consider, that leaking such information could significantly improve the guessing ability of the third party as we will discuss in Section IV.

Let $t = (B, 2, 3, B, \cdots, A, 7, 9, 3) \in \Sigma^{64}$ represent the *SHA-256* output of the following PIN code: `"43256891"`, where the *cleartext* is not known by the *client*. There are 10^8 different 8 digit number codes which will be our $\mathcal{DS}$. The client determines $r = 10$ to be the expected number of *candidate passwords* (which would be once again, an insufficient amount in a realistic scenario). To find a vector v that satisfies this the client first performs step CLC-NV:

$$N_v = \frac{r|\Sigma^l|}{|\mathcal{DS}|} = \frac{10 \cdot 16^{64}}{10^8}$$

The number of *decoy hashes* should be $N_v \approx 1.158 * 10^{70}$. By using the GEN-V algorithm we can generate a suitable $v = (v_1, \ldots, v_{128}) \in \Sigma^{128}$ vector, where $P_v(t) = 1$ is satisfied and $|\mathcal{X}_v|$ is near N_v. A suitable vector can be $v =$ `[7c27385c3f3f3f3f3f0c3f3f3f3f3f0c3f0c3f3f0c3f3f3f3f3f0c0c3f0c0c3f3f3f3f0c3f0c0c0c0c3f0c0c3f3f0c0c0c3f0c3f0c0c3f0c0c3f0c0c3f0c0c3f0c0c3f]` $\in \Sigma^{128}$.

After the server calculates the SHA-256 hashes for all the 10^8 different number codes, 9 different PINs are cracked from the set $\mathcal{X}_v$. By observing the candidates the *cracking server* still cannot know which is the password or if it is cracked at all. The cracked *candidate passwords* can be seen in Table II:

TABLE II
TOY EXAMPLE 2: GENERATED CANDIDATE HASHES

#	password	SHA-256
1	15851680	85869d73ebe4c562cbde168898669053...
2	18662804	a58bbf75d9cad8fc764cb3f364823a3b...
3	28251765	b26c78a4916d348565d986d4a6926034...
4	36823110	b27ccbfa99fc96dc38a445accd40d148...
5	37012370	945ab8ad984bfcb38b4f36cc36b73cae...
6	**43256891**	b23be566408ad8d2f1ac0d84330c3127...
7	56995169	7366edc5bc43387536ba6f47ad2ac834...
8	60409880	b689a9c7a5d539c8abfc197ae87a705e...
9	98509815	b3859a5f5ccfef995bd723c35598d137...

Selecting such a small candidate set size and exhausting the search space for a specific input can raise several concerns from a privacy point of view which we will discuss in connection with this example in Section IV. One could argue that for this toy example it would not be necessary to use a *PCaaS* third party, as *SHA-256* is a hash function is fast to compute, therefore, anyone could hash $|\mathcal{DS}| = 10^8$ passwords on their laptop using *John the Ripper*. This is however not the case if we change the hash function to *bcrypt*. If combined with proper key stretching techniques (e.g: 2^{16} iterations)

bcrypt would make it infeasible to brute force even such small key-spaces on desktop computers.

IV. Security and Privacy analysis of 3PC

This section considers the knowledge the third party cracking server possesses, or information an attacker could gain upon acquiring the vector v, considering different side channel attacks. Finally, we examine how a malicious attacker could try to tamper with the results and how such situations are evaded by the protocol design.

A. Probabilistic knowledge without assumptions

At first, we examine the best guessing strategy for the server without any assumption on the probability distribution of passwords, as they can vary from uniform (in case of machine generated) to variations of Zipf distribution (for human generated). Since by definition $t \in \mathcal{X}_v$, even without starting the cracking process a correct guess on the *target hash* can be made with $1/|\mathcal{X}_v|$ probability. *What can the server at this point know about the potential cleartext?* By design, the cracking process can only be successful if the selected *data set* contains a cleartext s^* such that $h(s^*) = t$. Let $\mathcal{A}$ be the event the server guesses s^* correctly, and $\mathcal{B}$ that $s^* \in \mathcal{DS}$. We can calculate the conditional probability of guessing s^* right:

$$\mathcal{P}(\mathcal{A}) = \mathcal{P}(\mathcal{A}|\mathcal{B}) \cdot \mathcal{P}(\mathcal{B}) \tag{7}$$

Note, that $s^* \in \mathcal{DS}$ is not an assumption but a condition. If this condition is satisfied the server is faced with the following situation. Without spending any resources, the best guess would be $1/|\mathcal{DS}|$. Note, that at this stage there is no point in examining the probability distribution over $\mathcal{DS}$. The predicate function selects every password in the *data set* uniformly with exactly $|\mathcal{DS}|/|\mathcal{X}_v|$ probability, and only a random subset of $s \in \mathcal{DS}$ satisfies $P_v(h(s)) = 1$, in other words:

$$\mathcal{P}\Big(P_v(h(s)) = 1\Big) = \frac{|\mathcal{DS}|}{|\mathcal{X}_v|}, \forall s \in \mathcal{DS}. \tag{8}$$

The server can greatly increase it's guessing ability by sorting out all passwords where $P_v(h(s)) = 0$ as shown in Figure 3.

Lemma IV.1. *If condition $\mathcal{B}$ is fulfilled and $|\mathcal{X}_v| < |\Sigma^l|$, the server must hash $\forall s \in \mathcal{DS}$ to maximize $\mathcal{P}(\mathcal{A}|\mathcal{B})$ over the given data set.*

Proof. If $|\mathcal{X}_v| < |\Sigma^l|$, then only a random subset of $s \in \mathcal{DS}$ will satisfy $P_v(h(s)) = 1$, others must be sorted out. During the hashing process the server's guess can be calculated as:

$$\mathcal{P}(\mathcal{A}|\mathcal{B}) = \frac{1}{|\mathcal{DS}| - \sum_{i=1}^{|\mathcal{DS}|} |P_v(h(s_i)) - 1|} \tag{9}$$

This can reach it's maximum when the predicate function has been calculated for $\forall s_i \in \mathcal{DS}$, from which the lemma follows. □

We mention again, that considering the probability distribution over the *candidate set* should only be attempted after sorting has been performed. From equation 9, and Figure 3 it

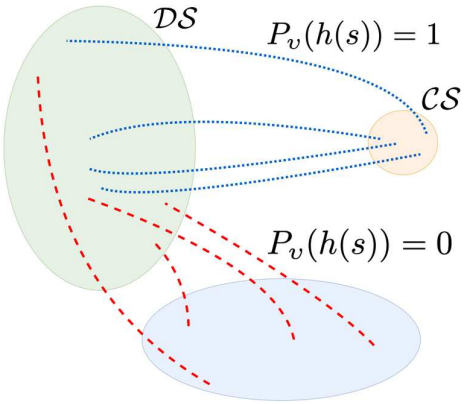

Fig. 3. Sorting the data set with the predicate function

can be seen, that if for example the hashing process is almost complete, and only one element is left in $\mathcal{DS}$ that is not sorted, performing that last hashing operation can further decrease the size of the set we need to guess from. Now it is obvious, that as the hashing process starts and after there are for example five candidates in $\mathcal{CS}$, the server cannot simply stop and say that now it has a $1/5$ chance to guess the cleartext.

The next vital question to examine is what can the server know about condition $\mathcal{P}(\mathcal{B})$. In the traditional cracking scenario, it was always obvious when a certain *data set* managed to crack a password, it was when a cleartext was found for the only hash digest. In our case there are always approximately r cleartexts found. One way to be certain that $\mathcal{P}(\mathcal{B}) \approx 1$, is to increase the size of the *data set* to $|\Sigma^l|$ unique elements. Then most passwords in $\mathcal{X}_v$ will be cracked regardless of how N_v was chosen. As a consequence $\mathcal{P}(\mathcal{A}) \approx 1/\mathcal{CS} \approx 1/N_v$.

Since it is completely impossible to hash $|\Sigma^l|$ elements with real hash functions, the server can never narrow down it's real guess to the *candidate set* without making assumptions or learning implicit information. For example if the server thinks that the *target hash* hides a human generated password, and the *data set* contains a dictionary of popular words and mangling rules, it might assign a higher probability for condition $\mathcal{B}$. If the password however, is generated by a password manager then it is highly unlikely that such a *data set* will recover the *target hash*. If the server aims to increase $\mathcal{P}(\mathcal{B})$, even if $|\Sigma^l|$ cannot be reached, a mathematically viable approach is to increase the size of the *cracking data set*. Even though this works perfectly in theory, it has little practical usefulness for the server: After step SND-P, the number of decoy hashes $|\mathcal{X}_v|$ is fixed. In addition to the dictionary we requested ($\mathcal{DS}_1$), the server will perform further attacks with $d-1$ more *data sets*.

$$\sum_{i=1}^{d} |\mathcal{DS}_i| \frac{|\mathcal{X}_v|}{|\Sigma^l|} \approx \sum_{i=1}^{d} r_i \tag{10}$$

As expected for every *data set*, the server will get r *candidate passwords*. Although this increases $\mathcal{P}(\mathcal{B})$, $\mathcal{P}(\mathcal{A}|\mathcal{B})$ decreases as the *candidate set* grows. Let $k = |\mathcal{DS}_1 \cup \mathcal{DS}_2 \cdots \cup \mathcal{DS}_d|$ where $\mathcal{DS}_i \cap \mathcal{DS}_j = \varnothing$ where $i \neq j$, $i, j \leq d \in \mathbb{N}^+$ then,

$$\lim_{k \to |\Sigma^l|} \sum_{i=1}^{d} r_i \to |\mathcal{X}_v| \tag{11}$$

As a consequence of 11, without being able to make any assumptions, the server can not make a guess on the pre-image better than $1/|\mathcal{X}_\upsilon|$.

Using the same vector υ as in Toy Example 1, we can simulate how an attack looks like when further *data sets* are used to crack elements in $\mathcal{X}_\upsilon$. The server now selects a different core dictionary containing 605834 French passwords[1]. To this, the server adds mangling rules appending a digit, and a special character at the end of every password using the following command: `john --format=crc32 --wordlist=dictionnaire_fr --mask=?w?d?s vectorv.txt`. Note, that in this case the vectorv.txt file contains all 5880 hashes, as *John the Ripper* doesn't support configuring a predicate function, but with $|\mathcal{X}_\upsilon|$ being this small writing it's content to a file is not a concern. As we later see in the real life use case, writing all the *decoy hashes* to a hard drive would be completely impossible. In total we expected $r = (5880 \cdot 605834 \cdot 10 \cdot 32)/16^8 \approx 265$ *candidate passwords*. Our cracking process returned 277 candidates which is close to r. A few examples from the new candidate set can be seen in Table III.

TABLE III
FURTHER CRACKING $\mathcal{X}_\upsilon$ FROM EXAMPLE 1.

#	password	#	password
1	Abaigar9^	190	ospedaletto3<
33	bréchaumont7"	204	Proostdij6]
46	Chassé1+ 3+	207	québecoise4
170	Montagne-Cherie7}	271	wadonville8(
182	NorthCrawley9&	277	éléphantiasis1[

Predicting, the password choosing behaviour of a single individual who we do not know anything about is not a straightforward task, especially if this behaviour can include using a password manager that generates random strings of unknown length. This leaves a malicious attacker with an almost impossible task when it is trying to determine $\mathcal{P}(\mathcal{B})$, where the only viable option is to try and learn further information to maximize $\mathcal{P}(\mathcal{B})$, and analyse the *candidate set* in hopes to improve $\mathcal{P}(\mathcal{A}|\mathcal{B})$.

B. Side-channel information on the search space

In the second toy example, if the server would somehow learn that we were after an 8 digit PIN code, it would be able to simply guess from the *candidate set*. This is due to that in addition to $\mathcal{P}(\mathcal{B}) = 1$, an exhaustive search has been performed on 8 digit PINs therefore, Lemma IV.1 has been satisfied. Thus guessing the password becomes $\mathcal{P}(\mathcal{A}) = 1/|\mathcal{CS}|$.

A possible mitigation strategy that works even if the server learns this information on $\mathcal{P}(\mathcal{B})$, is to increase the number of decoy hashes in $\mathcal{X}_\upsilon$ substantially. Thus, $|\mathcal{CS}|$ will grow as well, and even if the server guesses from the *candidate list* it will contain too many entries. At this point we would like to emphasize again, that the 3PC protocol does not communicate such information to the server. If the client is concerned that the server could learn useful information by observing the *data set*, the client can also change the *key-space*, such that it defines a bigger search space. Such a strategy for Toy Example 2 could be that the client asks all combinations of lower-case letters and digits ut to 8, where $|\mathcal{DS}| = 46^8$, or ask for an up to 9 character brute force with digits instead of exactly 8. Since this also raises the number of hashing operations, the decision on whether such a strategy is necessary, or feasible, depends on the security and privacy needs of the client.

An important differentiator between the client and the server side, is that the client in some cases can possess a lot of implicit information. The client must take extra care not to leak this info through a badly constructed *data set*, as it can impact the server's guess on the key space. If the password is hashed by *bcrypt*, only a small *data set* is selected that contains the core dictionary of words specific to the target. This can be name, street address, places lived, pet names, hobbies, favourite sports teams, etc., which is supplied with a set of mangling rules. Such information on its own can carry significant privacy risks both to the *data subject*, who's *PII* is leaked, and to the client side who would reveal implicit information to the server. For the above reasons, revealing privacy sensitive information through a badly constructed data set, that would explicitly identify the entity who the *target hash* belongs to, must be avoided. In contrast to the PIN code cracking example, where a increased *data set* is beneficial, with the second example it is more than necessary to increase the *data set*. For such data to remain unlinkable in this context, the *client* must consider relevant *privacy protection principles*, to ensure that the *data set* serves as a sufficiently sized anonymity set of similar data [42], [43], [44].

C. Server Side Guessing Based On Password Ranking

In connection with Toy Example 2 we would like to examine if the server can achieve a $\mathcal{P}(\mathcal{A}|\mathcal{B}) > 1/|\mathcal{CS}|$ guess by observing the candidate set. As we employed a brute-force attack, where most of the *data set* consists of random strings this will provide *perfect k-anonymity* if the *target hash* also hides a random generated string. What if the *target hash* hides a human generated password, would this *candidate set* still offer sufficient privacy protection? The security of human chosen PINs is a well researched topic [45], where 4 and 6 digit PINs were shown to follow a Zipf distribution [46]. Human chosen PINs are likely to follow non-random patterns such as "12345678", "11111111", a date structure like "19930810" or leet talk [47], such as "13371337" which reads "leetleet". If $\mathcal{X}_\upsilon$ is large these categories of *pre-images* will also be represented in $\mathcal{CS}$, as the predicate function selects a random subset of $\mathcal{DS}$. However, this will not provide perfect k-anonymity as the majority of passwords in the candidate set will be a random looking string. If an attacker tries to rank these PIN codes in the *candidate list*, the *target hash* can be in one of these human categories, or among the random strings where it is up to the assumptions of the server to try and determine which group is the one to pick from. In this example if the client wants to ensure k-anonymity even for human generated PINs, the best approach is to start with a *cracking data set*, containing number codes only adhering

[1]https://github.com/clem9669/wordlists/blob/master/dictionnaire_fr

to that format. As this is a smaller subset of all 8 digit number codes, N_υ can be chosen to be larger in order to provide a sufficiently sized anonymity set, where all *candidate passwords* look human generated. Thus if the *target hash* hides such a password it is now hidden in a set of similar data.

Referring back to Toy Example 1 where we employed a dictionary of human generated passwords as our *cracking data set*, one could also argue that the returned candidates are not equally likely, as password choosing habits of individuals usually does not follow a uniform distribution over all *cracking data sets* [15]. If the *data set* $\mathcal{DS}$ is a set of random equally likely strings, this would be true however, this cannot be generalised for all *cracking data sets*. As discussed in connection with Toy Example 1, the *Rock-You* database was shown by Wang et al. to follow a Zipf-like distribution [48]. As a consequence, employing such a cracking dictionary can result in some passwords in the *candidate set* appearing more probable from the server's point of view.

As the *candidate set* can only contain passwords from the given $\mathcal{DS}$ that was used for the cracking process, the ranking of passwords in the original dictionary (which the server possesses), can be applied to the *candidate set* to establish an order. Since our $\mathcal{DS}$ in 3PC will only contain every password once, frequency ranking for example is only possible if knowledge about other frequency ranked databases are taken into account [2]. Note, that Wang et al. used frequency ranking as a stepping stone from which the distribution was concluded [48]. As we discussed in Section II, there are many possible strategies to rank passwords, where each can produce a different result based on previous knowledge, training data, and assumptions it is built upon. Without implicit knowledge on the use case, ranking could completely impair the server's guessing ability, where even choosing at random could yield better results.

Ranking the passwords in $\mathcal{DS}$, and trying to guess the *target hash* based on the likelihood of the returned cleartext passwords in $\mathcal{CS}$, will not improve the guessing ability of the server. As we assume that the output of a hash function is uniformly distributed, the decoy passwords in $\mathcal{CS}$ are equally likely to be selected from $\mathcal{DS}$, regardless of what rank or likelihood is assigned to them. As such, the *candidate passwords* are essentially a randomized subset of the cracking set $\mathcal{DS}$. Which randomized subset is picked, depends on how we selected our υ. To put it differently, the vector υ will predetermine which hashed *pre-images* can satisfy the predicate function from $\mathcal{DS}$. However, this can never be known in advance without performing the hashing operation for every cleartext in $\mathcal{DS}$, as no correlation is assumed between cleartext passwords and their hashes. Thus, the occurrence of passwords from $\mathcal{DS}$ in the *candidate set*, is not based on how likely those passwords were, or what rank they had. If they all appear in the *data set* $\mathcal{DS}$ once, they are equally likely to show up in the *candidate set* $\mathcal{CS}$. To demonstrate this in practice, we used the *target hash* `0BChrist : C6BFABA2` from Toy Example 1, and created two additional vectors (υ' and υ"), which contain the same *target hash*. Each vector υ defines a different $\mathcal{X}_\upsilon$ decoy set. These υ vectors over the *Rock-You* data set will produce three different *candidate sets* depicted in Figure 4.

[2]https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-withcount.txt.tar.gz

TABLE IV
FREQUENCY RANKING

Frequency table		
υ	υ'	υ"
tangan : 14	alphan : 4	madagascar : 312
horses33 : 11	jenascia : 2	sapphire24 : 4
sapphire24 : 4	ozitos : 1	miyacute : 2

We employed a frequency analysis and a NIST password complexity [12] check on all three *candidate sets*. The results of the frequency analysis based on the original *Rock-You* database can be seen in Table IV. As for the NIST guideline, there are only two passwords fulfilling the complexity requirements, which are `"Kissarmy1!"` and `"MyP@ssw0rd!"`, whereas υ" didn't contain a suitable candidate. Both of these passwords had a rank one frequency, so for each vector, these two ranking strategies produced completely contradictory results. Furthermore, the ranking strategies failed to pinpoint the *target hash* as a likely candidate, for all three vectors.

Here we would like to underline a crucial idea: the client must never generate more than one υ vector using GEN-V, as the intersection of the corresponding $\mathcal{X}_\upsilon$ sets can narrow the search for the *target hash*. Our goal here was to demonstrate that different υ vectors produce different candidate sets over a specific $\mathcal{DS}$. As in general, it is not possible to accurately predict the behaviour of hypothetical individuals, who we do not know anything about, the server cannot make assumptions on what ranking strategy to use. Hence, in this case we rely on empirical evidence and highlight that applying different ranking strategies could easily mislead the server, regardless of which random subset of $\mathcal{DS}$ was selected by υ.

D. Implicit information disclosure

Imagine a scenario, where the *NATO Communications and Information Agency* (*NCI Agency*) wanted to recover the cleartext for the `"58727AD23361CA0323D4B3C22A6AFE78"` *NTLM hash* using a specialized *PCaaS* company. After the cracking process the server side observes the *candidate set*, and finds the cleartext password `Bices2014`, among the other password candidates. The *Battlefield, Information, Collection and Exploitation Systems* (*BICES*) serves as the primary intelligence-sharing network between and among all 28 *NATO* member nations, seven associated partner nations and the *NATO organization*. Upon learning that the customer is NCI Agency, a potential attacker could extrapolate the following information:

- This password could belong to a classified *NATO system* called *BICES*;
- The use of special characters is not enforced by the password policy;
- The password was possibly not changed in the last 8 years, in which case no password change policy is enforced.

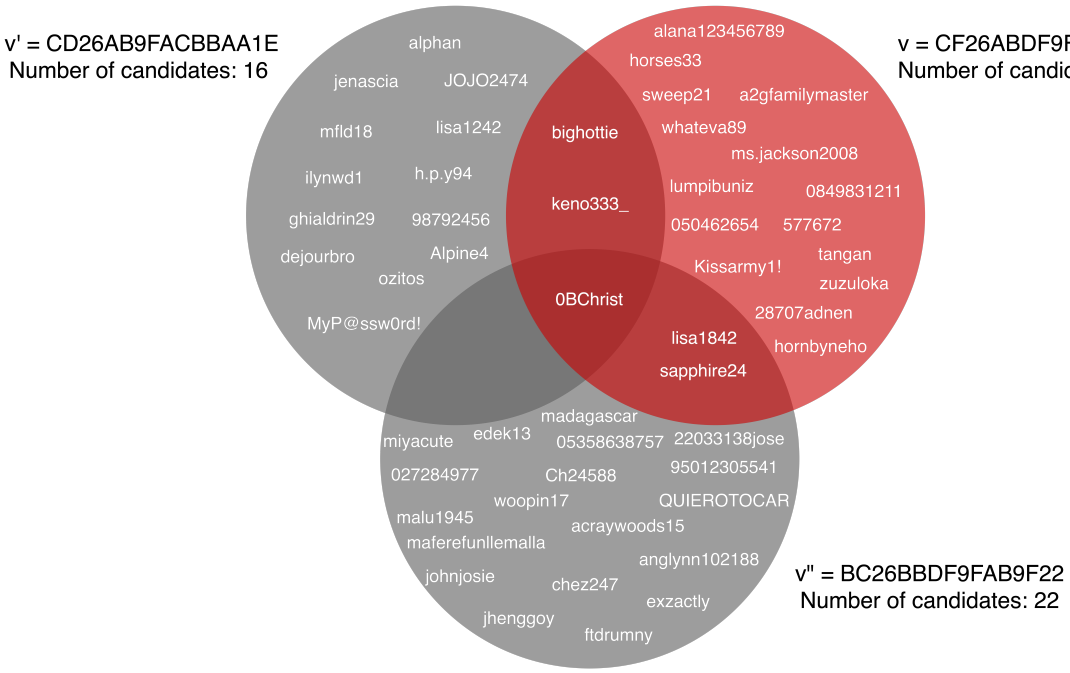

Fig. 4. An example on how a different vector υ defines a different random subset of $\mathcal{DS}$

Learning the client's identity is a form of information disclosure that can serve as a starting point for attackers. *NCI Agency* follows best practices that the hash of a password is confidential and should not leave the security boundary of the organization [13]. As a standalone password without the direct connection to NATO, `Bices2014` does not convey any useful information to the server. The word Bices can be associated with various concepts. For example in the Haitian *Creole* language Bices means bicycles and is spelled exactly the same. Similarly, Bices is the abbreviation for a mining seminar organized annually in China. By design, the 3PC protocol does not allow such information to be exchanged between the server and the client. However, the client must take precaution not to leak this via other channels. This is an example for information disclosure outside of the 3PC protocol.

Although this looks like a high risk scenario on the first glance, everything depends on how the *data set* was chosen. Let the selected core dictionary contain common words, and ones related to NCI Agency, such as; NATO, BICES, North, Atlantic, Treaty, Organization, NCI, Agency, Jens, Soltenberg, etc. In addition, this can be complemented with a set of mangling rules such as; capitalizing the first letter of every key-word, appending one to three special characters, and optionally replacing relevant letters in the core words to form leet talk like N4t0. A *data set* constructed with such rules, will not only have a lot of relevant passwords, but it can be argued that the probability distribution is close to uniform, as these passwords look equally likely from the server's perspective. The security parameter r can be adjusted such that the *candidate set* can contain millions of passwords in this format, like NC1N4T0.2022, Bic3$pass123 etc.

What we deem acceptable from a privacy perspective greatly depends on the use case. If a company tests corporate passwords of employees in important roles, getting the password cracked by common dictionaries is a good scenario. To discover and change a weak password, is better then the inevitable data breach. Note, that the password is only cracked by such a *cracking data set*, if the cleartext happens to be in this format. The more difficult question comes when the password is not cracked. What if the third party continues cracking with different dictionaries, brute force attacks etc. In this case, the server runs into the problem of growing *candidate sets* as we previously discussed in relation to equation 10.

E. Foul Play

When the client side starts step CHK-CS of the protocol to see if a *pre-image* for the target hash was found, as a part of this step it must investigate whether the server indeed exhausted the agreed search space and performed $|\mathcal{DS}|$ hashing operations. To confirm this, the client relies on *proof of work*, which is a concept where a *prover* demonstrates to a *verifier* that a certain amount of a computational effort has been expended in a specified interval of time [49]. In the case of 3PC the client knows that approximately r candidates must be returned based on the formula, $|\mathcal{X}_\upsilon| \frac{|\mathcal{DS}|}{|\Sigma^l|} \approx r$, where $P_\upsilon(x) = 1$ must be satisfied for all hash digests.

The server cannot simply fill the candidate set with randomly chosen r hash digests as they need to fulfill the *predicate function* for the given vector υ. If the server selects fitting hash digests with fake *cleartext passwords*, the client can simply select a random subset in $\mathcal{CS}$, and hash it to verify if it is indeed the correct one. Obviously, r is only an expected value but deviating from it significantly can suggest that the server is not truthful in expending the appropriate resources.

F. Plausible deniability

If any entity upon acquiring the vector v or the *candidate set*, would claim that the client was trying to break a password hash t' that belongs to them, they can not incriminate the client. Even if it is true that $t' \in \mathcal{X}_v$, the client can always rely on plausible deniability and say that this happened by chance. Indeed, when a specific vector v is calculated for a given *target hash*, it is true that any $x \in \mathcal{X}_v$ could have been the seed given to GEN-V, that produced vector v. This means that the probability that an arbitrary hash digest from Σ^l falls into $\mathcal{X}_v$ is $\frac{|\mathcal{X}_v|}{|\Sigma^l|}$, which is a non-negligible probability if the parameters were chosen properly. The client, before transferring v, can check if the chosen N_v provides satisfactory plausible deniability based on the scenario.

G. Zero-Knowledge variation

If the *cracking data set* is small, it enables the 3PC protocol to be used in a *zero-knowledge* setting. As we noted, the upper limit for the size of $\mathcal{X}_v$, is how many *candidate passwords* can be transferred and stored from a given *data set*. If the client only needs a small *cracking data set*, where it would be possible to store and transfer $|\mathcal{DS}|$ hashes and the corresponding passwords, then N_v can be selected such that: $N_v = |\mathcal{X}_v| = |\Sigma^l|$ i.e., $\{v_{2i} = F \wedge v_{2i-1} = 0 : \forall 1 \leq i \leq l\}$. Now, there is no need to calculate the predicate function as $P_v(h(s)) = 1, \forall s \in \Theta^*$ as the decoy set is essentially the output space of h. It is then trivial, that not sending the *target hash* at all, or sending it in vector v with the above described *zero-knowledge* setting, is equivalent.

V. Real life experiments

In this experiment we are looking to empirically verify the following question: Is it really feasible to crack billions of hashes using the theory introduced in the 3PC protocol? In the following section, we show the protocol running on an FPGA architecture, where the second part of this section will aim to analyse and interpret the results and related privacy implications.

A. Implementation on the RIVYERA FPGA cluster

The Cost-Optimized Parallel Code Breaker (COPACOBANA) FPGA hardware was introduced in the annual Conference on Cryptographic Hardware and Embedded Systems (CHES) in 2006 [50]. The RIVYERA FPGA hardware architecture [51] is the direct successor of COPACOBANA, equipped with Spartan–6 Family FPGA modules which can be seen in Figure 5. The demonstration of the 3PC protocol is conducted on a RIVYERA S6-LX150 cluster, containing 256 Xilinx Spartan–6 LX150 FPGA modules.

RIVYERA is using the *se_decrypt* 3.00.08 cryptanalysis framework, developed by SciEngine[3], which is a password cracking tool designed to maximize the efficiency of the FPGA modules. This tool allows configuring the predicate function for a given vector v, without any design changes or modification in the software. As discussed in the introduction, the main

[3]https://www.sciengines.com/it-security-solutions/cryptanalysis-tools/

Fig. 5. RIVYERA S6-LX150 server [51]

motivation behind this paper originates from a penetration testing engagement where the Red Team was able to retrieve an *NTLM* hash for an important service account. It was known that all service account passwords are randomly generated 9 character long strings containing uppercase and lowercase letters plus numbers. According to the signed contract revealing any cleartext passwords to third parties was prohibited. However, the client made an exception and approved the use of a PCaaS, if the Red-Team can assure that the third party server does not have a guess with better than 2^{-29} probability. The following example presents a realistic scenario that solves this problem using 3PC.

Let the data set $\mathcal{DS}$ be the set of all 9 character long strings containing uppercase and lowercase letters plus numbers hence, $|\mathcal{DS}| = 62^9$. According to the specified security requirements, the server must not have a better guess than $1/|\mathcal{CS}| < 2^{-29}$. As a reminder, the 3PC protocol provides a stronger security, as the server can not make a $1/|\mathcal{CS}|$ guess on the *candidate set* without making several assumptions. The client wanted assurance that even in the worst case scenario if $s^* \in \mathcal{DS}$ is known (in other words $\mathcal{P}(\mathcal{B}) = 1$), the maximum guessing probability of the server is $1/|\mathcal{CS}|$. In this example the NTLM *target hash* is $t =$`8AC54208A85C340AE9B8B0CDB236F14C`.

Compared to the GEN-V step in the 3PC protocol, where it was possible to set a degree of freedom on each hexadecimal character of the target hash, the `se_decrypt` tool is more limited. The built-in `--hit-mask` parameter can be used as a "restricted" GEN-V function to create v. It only allows setting the degree of freedom by bytes (by two hex characters). To be more precise, the n-th bit of the hit mask (from the right) corresponds to the n-th byte of the target (also from the right). If the n-th bit of the hit mask is a "1", the n-th byte of a resulting *candidate password hash* must completely match the n-th byte of the *target hash*, otherwise it does not need to match. The `se_decrypt` expects this vector v, and the `--hit-mask` both in hexadecimal representation. As a consequence, we can only select even powers of sixteen as the size of N_v. To transform the vector v which is suitable for the `se_decrypt` tool, one needs to solve the following RIVYERA specific inequality:

$$2^{29} < \frac{|\mathcal{DS}| \cdot 16^{2x}}{|\Sigma^l|} \tag{12}$$

The first x which satisfies 12 is $x = 13$, for which we have

$$\frac{62^9 \cdot 16^{26}}{16^{32}} \approx 806\,873\,234$$

Thus we expect approximately $r \approx 806$ million candidates after hashing all possible strings in $\mathcal{DS}$. A suitable vector is $v =$`[88AACC550F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F44CC]` $\in \Sigma^{64}$. One can easily generate the RIVYERA specific mask on the server side namely `8AC50000000000000000000000000004C`. This shows that from the 32 hexadecimal character long target hash we need to hide 26 characters (16^{26}) in any position. Using the *se_decrypt* tool, one can use the following parameters to start cracking the data set: `se_decrypt -a nthash --hash 8AC50000000000000000000000000004C --hit-mask C001 --lower-case --upper-case --numbers --min-len 9 --max-len 9 --no-stop ---logfile candidate.txt`
Please note, that the binary representation of the `C001` parameter value is `1100000000000001`, meaning we disregard 13 bytes from the middle (26 hexadecimal characters) from the original hash. The `8AC50000000000000000000000000004C` hash together with the `--hit-mask C001` produces the same decoy hashes as the original vector v, therefore these are equivalent representations of the $\mathcal{X}_v$ *decoy set*.

In our control test, brute forcing the whole 62^9 key space for one NTLM hash takes approximately 19.6 hours on RIVYERA with 192 billion hash/sec. Here comes the real benefit of the 3PC protocol: After applying the `--hit-mask C001` parameter on our modified vector v defining $|X_v| = 16^{26}$ hashes, the RIVYERA cluster finished the cracking process after 19.6 hours, under the same time as it took to find a single hash, and found 806834341 *candidate passwords*. Writing all candidates to disk took less than 50 seconds which is negligible compared to the cracking time. The output of the cracking process can be seen in Figure 6. The last step is to send the resulting file to the client, which is around 20.0 GB. The client then simply checks whether $\exists(s,t) \in \mathcal{CS}$, which is true in our case. The cleartext corresponding to the *target hash* t is `"bKFQ4Q8C0"`.

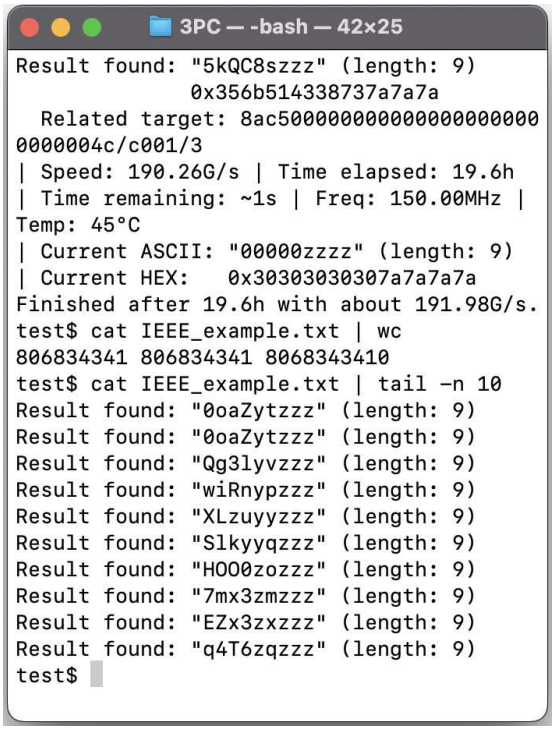

Fig. 6. Bash terminal output on the cracking process

B. Experimental result analysis

A very clear advantage of 3PC is that using a predicate function allows the efficient handling of huge anonymity sets, that would otherwise be infeasible. In our experiment presented on RIVYERA, the size of the *decoy set* was selected to be $|\mathcal{X}_v| = 16^{26} = 202824096036516704239472512860 16$. Applying the P_v function to the hashed *data set* produced an anonymity set of $r \approx 806$ million decoy passwords. In this example $\mathcal{CS}$ provides *perfect k-anonimity* on the *target hash*, as the password is hidden in a set of similar data.

Despite the *decoy set* being large, the 3PC protocol utilizes the predicate function to allow a constant time check for verifying $h(s) \in \mathcal{X}_v$, (based on lemma III.1) for $\forall s \in \mathcal{DS}$. As a result we can increase $|\mathcal{X}_v|$ without any penalties on computational speed. The only upper limit is the size of the *candidate set* which the server needs to write to disk. In NTLM, one hash value is exactly 128 bit. From this we can easily calculate how much hard drive space it would take if one would attempt to write the content of the *decoy set* to the hard drive: $(16^{26} \cdot 128)/8/1024^5 \approx 2.88 \cdot 10^{17}$, which is 3 million billion Petabytes. This would clearly be impossible to handle with today's technology. However 800 million cleartext-hash pairs in the *candidate set* only takes up 20 GB, which can be easily transferred to the client. To the best of our knowledge this example was the first documented case when perfect *k-anonimity* was achieved during a password cracking process using a third party service.

C. Password analysis and Parameter security

What if the client was not sure if the *target hash* hides a machine generated code? Would it be safe to use a brute force attack, assuming that the *candidate set* provides a sufficient anonymity set for a password like `JohnnY007`? To answer this question we have analyzed the 800 million *candidate passwords* which are a random subset of the *data set*, selected by the P_v function. We have used the `cracklib-check` plugin, which groups passwords into six categories. From the 800 million candidates approximately 0.07% of the passwords were flagged as weak. The exact distribution for each category can be seen in TABLE V.

As an example we list below some of the dictionary based words that the tool found.

TABLE V
STRENGTH DISTRIBUTION OF RANDOMLY GENERATED PASSWORDS

Category	Number of passwords	Percentage
The password is random (OK)	806263493	99.93%
The password is too simple	532747	0.066%
Based on a dictionary word	11763	0.00146%
Reversed dictionary word	11895	0.00147%
Not enough different characters	6111	0.00076
Looks like an Insurance number	8332	0.001%

- `WnbBATman` : This password can be interpreted as the famous superhero of the Warner Brothers. "WnB Batman".
- `MRloNDon6`: Starts with a Title, "MR", followed by a dictionary word, and ends with a number. "Mr London 6" can also be mistaken as human generated.
- `PaYpaLQSC`: What about a password used by employees from the Paypal Quality Service Center?
- `goOgLEFOG`: This can be interpreted as the Google Cloud Platform API solution "Google Fog".

Although there are a good number of passwords that can be easily classified as human generated, this set would not provide perfect k-anonymity for all human generated passwords. The occurrence of shorter dictionary words is higher, but passwords that use all 9 characters in a plausible sequence are less common. This is to be expected, as the protocol is designed to provide an anonymity set for data that makes up the *cracking data set*. Therefore, we provide the following guidelines for practical applications.

If the client possesses no implicit information on the *pre-image* of the *target hash*, the best approach is to start with a hybrid attack, based on a core dictionary and a rule-set. A smaller *data set* with a larger N_v, for which the client can repeatedly switch between the *data sets* of similar size, without changing $\mathcal{X}_v$ has an added benefit. This will make it impossible for the server to conduct attacks with significantly bigger *data sets* such as brute force attacks, as the resulting *candidate set* for the same N_v could reach sizes of hundreds of Petabytes, making it impossible to store or evaluate the results. If these cracking sessions are proven to be unfruitful, the client could choose to pivot to brute force attacks. If this entails a significantly larger *data set*, the client can shrink $\mathcal{X}_v$, but never create a new vector v through GEN-V.

VI. CONCLUSION

This paper introduces privacy-preserving password cracking, showing how the computational resources of an untrusted third party can be used to crack a password hash. The anonymity sets that hide the target hash, would make it impossible in traditional cracking scenarios to process or store this number of hashes and the corresponding cleartext results. We circumvented this by extending the theory of predicate functions to operate on the output of hash functions. On top of this, we demonstrated that increasing the number of decoy hashes bears no impact on the hash rate, making the 3PC protocol very efficient. The implementation of the protocol was shown through two Toy Examples, and one real life implementation running on the RIVYERA FGPA cluster. Through these, we were able to verify our original goals we set out to examine. The server only learns probabilistic information both on the *target hash* and the *cleartext password*. The protocol is resistant against *foul play*, where the server gains no tangible advantage towards learning the target hash, or it's *pre-image* by not following the steps of the protocol. The protocol ensures *plausible deniability*, where the client can claim to have aimed for a different target. Through a proof of work scheme, the client can have a statistical argument if they suspect that the requested search space has not been exhausted. The empirical tests and the theoretical analysis suggests that the 3PC protocol is suitable for practical use both from a security, a privacy, and an efficiency perspective.

ACKNOWLEDGMENT

The authors would like to thank Dr. Axel Y. Poschmann for the constructive brainstorming sessions, and Dr. Lothar Fritsch for his insightful comments.

REFERENCES

[1] L. O'Gorman, "Comparing passwords, tokens, and biometrics for user authentication," *Proceedings of the IEEE*, vol. 91, no. 12, pp. 2021–2040, Dec. 2003. [Online]. Available: http://ieeexplore.ieee.org/document/1246384/

[2] S. Srinivas, "One step closer to a passwordless future," May 2022. [Online]. Available: https://blog.google/technology/safety-security/one-step-closer-to-a-passwordless-future/

[3] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 553–567, iSSN: 2375-1207.

[4] K. Siddique, Z. Akhtar, and Y. Kim, "Biometrics vs passwords: a modern version of the tortoise and the hare," *Computer Fraud & Security*, vol. 2017, no. 1, pp. 13–17, Jan. 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1361372317300076

[5] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y.-Y. Choong, K. K. Greene, and M. F. Theofanos, "Digital identity guidelines: authentication and lifecycle management," National Institute of Standards and Technology, Tech. Rep., Jun. 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf

[6] N. Tihanyi, A. Kovacs, G. Vargha, and A. Lenart, "Unrevealed Patterns in Password Databases Part One: Analyses of Cleartext Passwords," in *Technology and Practice of Passwords*, ser. Lecture Notes in Computer Science, S. F. Mjolsnes, Ed. Cham: Springer International Publishing, 2015, pp. 89–101.

[7] P. Kamal, "Security of Password Hashing in Cloud," *Journal of Information Security*, vol. 10, no. 2, pp. 45–68, Feb. 2019, number: 2 Publisher: Scientific Research Publishing. [Online]. Available: http://www.scirp.org/Journal/Paperabs.aspx?paperid=90861

[8] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "Passwords and the evolution of imperfect authentication," *Communications of the ACM*, vol. 58, no. 7, pp. 78–87, Jun. 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2699390

[9] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password Strength: An Empirical Analysis," in *2010 Proceedings IEEE INFOCOM*, Mar. 2010, pp. 1–9, iSSN: 0743-166X.

[10] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," in *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. Chicago, Illinois, USA: ACM Press, 2010, p. 162. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1866307.1866327

[11] S. M. Egelman, S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Of Passwords and People: Measuring the Effect of Password-Composition Policies," *NIST*, May 2011, last Modified: 2017-02-19T20:02-05:00 Publisher: Serge M. Egelman, Saranga Komanduri, Richard Shay, Patrick G. Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie F. Cranor. [Online]. Available: https://www.nist.gov/publications/passwords-and-people-measuring-effect-password-composition-policies

[12] Y.-Y. Choong, M. F. Theofanos, and H.-K. Liu, "United States Federal Employees' Password Management Behaviors – A Department of Commerce Case Study," *NIST*, Apr. 2014, last Modified: 2018-11-10T10:11-05:00 Publisher: Yee-Yin Choong, Mary F. Theofanos, Hung-Kung Liu. [Online]. Available: https://www.nist.gov/publications

[13] B. Ewaida, "Pass-the-hash attacks: Tools and Mitigation," 2010. [Online]. Available: https://www.sans.org/white-papers/33283/

[14] Y. Li, H. Wang, and K. Sun, "Personal Information in Passwords and Its Security Implications," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2320–2333, Oct. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7931642/

[15] D. Wang, P. Wang, D. He, and Y. Tian, "Birthday, name and bifacial-security: understanding passwords of Chinese web users," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, Aug. 2019, pp. 1537–1554.

[16] R. Veras, C. Collins, and J. Thorpe, "On the Semantic Patterns of Passwords and their Security Impact," in *Proceedings 2014 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/semantic-patterns-passwords-and-their-security-impact/

[17] R. L. Rivest, L. Adelmann, and M. L. Dertouzos, "On DataBanks And Privacy Homomorphisms," *Foundations of Secure Computation*, 1987. [Online]. Available: http://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf

[18] M. Alloghani, M. M. Alani, D. Al-Jumeily, T. Baker, J. Mustafina, A. Hussain, and A. J. Aljaaf, "A systematic review on the status and progress of homomorphic encryption technologies," *Journal of Information Security and Applications*, vol. 48, p. 102362, Oct. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212618306057

[19] P. Parmar, S. Padhar, S. Patel, N. Bhatt, and R. Jhaveri, "Survey of Various Homomorphic Encryption algorithms and Schemes," *International Journal of Computer Applications*, vol. 91, Mar. 2014.

[20] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A Survey on Homomorphic Encryption Schemes: Theory and Implementation," *ACM Computing Surveys*, vol. 51, no. 4, pp. 79:1–79:35, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3214303

[21] D. Hoover and B. Kausik, "Software smart cards via cryptographic camouflage," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*. Oakland, CA, USA: IEEE Comput. Soc, 1999, pp. 208–215. [Online]. Available: http://ieeexplore.ieee.org/document/766915/

[22] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-Resistant Password Management," in *Computer Security – ESORICS 2010*, ser. Lecture Notes in Computer Science, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer, 2010, pp. 286–302.

[23] A. Juels and T. Ristenpart, "Honey Encryption: Security Beyond the Brute-Force Bound," May 2014.

[24] A. Juels and R. L. Rivest, "Honeywords: making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 145–160. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516671

[25] Jennifer Pullman, Kurt Thomas, and Elie Bursztein, "Protect your accounts from data breaches with Password Checkup," 2019. [Online]. Available: https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html

[26] 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," 2019, pp. 1556–1571. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/thomas

[27] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for Checking Compromised Credentials," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1387–1403. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3354229

[28] Z. Hou and D. Wang, "New Observations on Zipf's Law in Passwords," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9777714/

[29] J. Bonneau, "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 538–552, iSSN: 2375-1207.

[30] J. Blocki, A. Datta, and J. Bonneau, "Differentially Private Password Frequency Lists," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/differentially-private-password-frequency-lists.pdf

[31] D. Malone and K. Maher, "Investigating the Distribution of Password Choices," *Computing Research Repository - CORR*, Apr. 2011.

[32] S. Aggarwal, S. Houshmand, and M. Weir, "New Technologies in Password Cracking Techniques," in *Cyber Security: Power and Technology*, ser. Intelligent Systems, Control and Automation: Science and Engineering, M. Lehto and P. Neittaanmäki, Eds. Cham: Springer International Publishing, 2018, pp. 179–198. [Online]. Available: https://doi.org/10.1007/978-3-319-75307-2_11

[33] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammars," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 391–405, iSSN: 2375-1207.

[34] A. Kanta, I. Coisel, and M. Scanlon, "PCWQ: A Framework for Evaluating Password Cracking Wordlist Quality," *The 12th EAI International Conference on Digital Forensics and Cyber Crime*, Dec. 2021, publisher: Springer. [Online]. Available: https://markscanlon.co/papers/PasswordCrackingWordlistQuality.php

[35] A. Kanta, S. Coray, I. Coisel, and M. Scanlon, "How viable is password cracking in digital forensic investigation? Analyzing the guessability of over 3.9 billion real-world accounts," *Digit. Investig.*, 2021.

[36] J. Galbally, I. Coisel, and I. Sanchez, "A New Multimodal Approach for Password Strength Estimation Part I: Theory and Algorithms," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 2829–2844, Dec. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7776908/

[37] D. Wang, D. He, H. Cheng, and P. Wang, "fuzzyPSM: A New Password Strength Meter Using Fuzzy Probabilistic Context-Free Grammars," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Toulouse, France: IEEE, Jun. 2016, pp. 595–606. [Online]. Available: http://ieeexplore.ieee.org/document/7579775/

[38] S. Oesch and S. Ruoti, "That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Thirteen Password Managers," Dec. 2019, arXiv:1908.03296 [cs]. [Online]. Available: http://arxiv.org/abs/1908.03296

[39] J. Galbally, I. Coisel, and I. Sanchez, "A New Multimodal Approach for Password Strength Estimation. Part II: Experimental Evaluation," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 2845–2860, Dec. 2017, conference Name: IEEE Transactions on Information Forensics and Security.

[40] S. Gorbunov, V. Vaikuntanathan, and H. Wee, "Predicate Encryption for Circuits from LWE," in *Advances in Cryptology – CRYPTO 2015*, R. Gennaro and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, vol. 9216, pp. 503–523, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-662-48000-7_25

[41] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, Dec. 1982. [Online]. Available: https://doi.org/10.1007/BF01457454

[42] L. Sweeney, "k-ANONYMITY: A MODEL FOR PROTECTING PRIVACY," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, Oct. 2002. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218488502001648

[43] R. Bayardo and R. Agrawal, "Data privacy through optimal k-anonymization," in *21st International Conference on Data Engineering (ICDE'05)*, Apr. 2005, pp. 217–228, iSSN: 2375-026X.

[44] A. Pfitzmann, T. Dresden, M. Hansen, and U. Kiel, "Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management – A Consolidated Proposal for Terminology," *Citeseer*, 2008. [Online]. Available: http://dud.inf.tu-dresden.de/Anon_Terminology.shtml

[45] P. Markert, D. V. Bailey, M. Golla, M. Dürmuth, and A. J. Aviv, "This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone

Unlock PINs," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 286–303, iSSN: 2375-1207.

[46] D. Wang, Q. Gu, X. Huang, and P. Wang, "Understanding Human-Chosen PINs: Characteristics, Distribution and Security," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Abu Dhabi United Arab Emirates: ACM, Apr. 2017, pp. 372–385. [Online]. Available: https://dl.acm.org/doi/10.1145/3052973.3053031

[47] W. Li and J. Zeng, "Leet Usage and Its Effect on Password Security," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2130–2143, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9316928/

[48] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's Law in Passwords," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 11, pp. 2776–2791, Nov. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7961213/

[49] N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "A Cross-Stack Approach Towards Defending Against Cryptojacking," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 126–129, Jul. 2020, conference Name: IEEE Computer Architecture Letters.

[50] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking Ciphers with COPACOBANA –A Cost-Optimized Parallel Code Breaker," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Berlin, Heidelberg: Springer, 2006, pp. 101–118.

[51] RIVYERA, "RIVYERA_s6_lx150_rev431_datasheet." [Online]. Available: https://www.sciengines.com/wp-content/uploads/RIVYERA_S6_LX150_REV431_DATASHEET_4HU.pdf