Useful Password Hashing: How to Waste Computing Cycles with Style

Markus Dürmuth
Horst-Görtz-Institute for IT Security
Ruhr-University Bochum
Bochum, Germany
markus.duermuth@rub.de

ABSTRACT

Password-based authentication is widely used today, despite problems with security and usability. To control the negative effects of some of these problems, best practice mandates that servers do not store passwords in clear, but password hashes are used. Password hashes slow down the password verification and thus the rate of password guessing in the event of a server compromise. A slower password hash is more secure, as the attacker needs more resources to test password guesses, but at the same time it slows down password verification for the legitimate server. This puts a practical limit on the hardness of the password hash and thus the security of password storage.

We propose a conceptually new method to construct password hashes called "useful" password hashes (UPHs), that do not simply waste computing cycles as other constructions do (e.g., iterating MD5 for several thousand times), but use those cycles to solve other computational problems at the same time, while still being a secure password hash. This way, we are convinced that server operators are willing to use slower password hashes, thus increasing the overall security of password-based authentication.

We give three constructions, based on problems from the field of cryptography: brute-forcing block ciphers, solving discrete logarithms, and factoring integers. These constructions demonstrate that UPHs can be constructed from problems of practical interest, and we are convinced that these constructions can be adapted to a variety of other problems as well.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Security and Protection — *Authentication*

General Terms

Security

NSPW'13, September 9–12, 2013, Banff, AB, Canada.

ACM 978-1-4503-2582-0/13/09 ...$15.00.
http://dx.doi.org/10.1145/2535813.2535817 .

Keywords

Authentication, password hashing

1. INTRODUCTION

Passwords are still the most widely used form of user authentication on the Internet. This is unlikely to change in the foreseeable future, because alternative technologies such as security-tokens and biometric identification have a number of drawbacks that prevent their wide-spread use outside of specific realms: Security tokens, for example, need to be managed, which is a complicated task for Internet-wide services with millions of users, they can be lost, and they require some standardized interface to connect them to every possible computing device (including desktop computers, mobile phones, tablet PCs, and others). Biometric identification systems require extra hardware readers, false-rejects cause user annoyance, and many biometrics are no secrets (e.g., we leave fingerprints on many surfaces we touch). Passwords, on the other hand, are highly portable, easy to understand by users, and relatively easy to manage for the administrators. Still, there are a number of problems, most notably the trade-off between strong password vs. passwords that humans can conveniently remember. Various studies and recommendations have been published presenting the imminent threat of insufficiently strong passwords (see, e.g., [3, 21, 34]).

Best practice mandates that passwords are not stored in clear in computer systems, and typically only the hash of a password is stored. Assuming that these hash functions are one-way, i.e., it is intractable to compute a matching input to a given output, the most efficient attack against such password hashes are *guessing attacks*, where the attacker systematically "guesses" a large number of likely password candidates, applies the password hash, and verifies if the result matches. Studies indicate that a substantial number of passwords can be guessed with moderately fast hardware [35]. Therefore, using plain MD5, SHA-1, or even newer hash functions such as SHA-256 or SHA-3, is not considered safe.

To slow down guessing attacks, so-called *password hashes* were proposed. These are hash functions specifically designed for storing passwords, and they are constructed in a way that makes them slow to evaluate in order to lower the number of guesses an attacker can verify given a fixed amount of resources. Commonly used password hashes are iterated hash functions (e.g., 500 iterations of MD5, or several thousand iterations of SHA-1), the *password-based key derivation function PBKDF2* (PKCS #5 v2.0 and RFC 2898 [14, 10]), bcrypt [28], which is based on the Blowfish

key setup phase, and scrypt [25], which is designed to use large amounts of memory trying to slow down computations specifically on GPUs.

While using these functions slows down an attacker as desired, it naturally also slows down the verification by the legitimate server by approximately the same factor (depending on the computer architecture used by the legitimate server and the attacker, respectively). This is problematic because the more secure the password hash is, the more it slows down the legitimate server, and thus server operators might hesitate to use password hashes that offer high security. The reason here is that, from the point of view of the server operator, the computing cycles spent for the password hash are actually "wasted computing cycles". We believe that, if there was a way to spend these cycles in a way that is "useful" for the server operator, then he will be more willing to use slow password hashes.

In this work we will explore constructions that have dual use: First, they are *secure password hashes* suitable to store passwords securely, and second, using them as password hashes will let the legitimate server to solve some other problem of interest as a "side product". We are, to the best of our knowledge, the first to consider such functions. We provide three specific constructions that solve three different problems from the field of cryptography, but we believe the idea is by no means restricted to those problems. On the contrary, the constructions show different strategies that we expect to be applicable to other problems and similar constructions as well.

1.1 Related work

It was observed early that users tend to choose weak passwords [20], and several subsequent studies have found similar results [3, 36, 37, 15, 8, 19, 4]. Several tools are freely available to perform attacks against hashed passwords. Probably best known is *John the Ripper* [9], which has fast implementations for various password hashing schemes (mostly for CPUs) as well as sophisticated *mangling rules* that efficiently produce password candidates from carefully chosen dictionaries (containing English words, names, and more strings that are typically used for passwords) that are transformed by mangling rules. Another well-known tool is HashCat [1], which is particularly interesting because it offers one of the fastest and most complete collections of implementations of password hashes on GPUs. More password guessers have been proposed in the literature. One example is based on probabilistic grammars [34], which can be seen as a way to automatically extract mangling rules from sets of real-world passwords. Another approach uses Markov models, which have proven suitable for a number of tasks related to human-readable text. Narayanan et al. [21] proposed a method which guessed passwords in some random order and was mostly suited for the use in rainbow tables (this method is implemented in John the Ripper as an extension to the above attack). A more recent variant was proposed by Castelluccia et al. [6] and can produce password guesses in decreasing order of likelihood.

To prevent guessing attacks one can either use a stronger password hash or force the user to choose stronger passwords. The latter is usually tried by using password rules, which is largely ineffective and creates major usability issues, and no satisfying solution has been found yet. While traditionally simple hash functions (most notably MD5 and SHA-1) are used, these are a poor choice nowadays: Password crackers on modern GPUs can verify up to billions of guesses per second [1] when such simple hashes are used, which cracks even relatively strong passwords in reasonable time. Iterated hash functions, such as 500 iterations of MD5 were a reasonable choice before the widespread availability of cheap GPUs, but their use is discouraged nowadays. PBKDF2 [14, 10] is significantly stronger, but still can be evaluated fast enough to mount a substantial guessing attack [10]. The password hash bcrypt [28] is probably the best choice nowadays, but it is not yet clear if fast implementations on FPGAs might exist, which seems possible as the memory footprint of the algorithm is relatively small. To specifically address this issue scrypt [25] was proposed, which uses large amounts of memory. However, it lacks wide acceptance, and the large memory requirements make it even hard to implement on the legitimate server. Recently, a competition was announced [24] to find a suitable replacement. Note that our proposal does not target this competition, but rather seeks fundamentally new ideas in the field of password hashes.

Several papers aim to measure the *strength of passwords*. So-called pro-active password checkers where used to exclude weak passwords [31, 16, 3, 23, 5], however, most proactive password checkers use relatively simple rule-sets to determine password strength, which have been shown to be a rather bad indicator of real-world password strength [33, 17, 7]. More recently, Schechter et al. [30] classified password strength by counting the number of times a certain password is present in the password database, and Markov models have been shown to be a very good predictor of password strength and can be implemented in a secure way [7].

A related idea in the context of proof-of-work systems, termed *bread-pudding protocols*, was proposed by Jakobsson and Juels [13], but their design cannot use arbitrary problems as underlying problem, and they give examples for problems very similar to the proof-of-work only. Becker et at. [2] mention using proofs-of-work to compute something useful as interesting research direction. The concept is vaguely related to the Chinese Lottery [18, 29] which utilizes spare cycles of existing hardware.

1.2 Outline

Section 2 briefly reviews relevant aspects of password hashing, before Section 3 introduces the concept of *useful password hashes* and discusses basic requirements. In Sections 4–6 we give three constructions for UPHs based on three different problems, brute-forcing block ciphers, computing discrete logarithms, and factoring integers, which demonstrate the usefulness of the basic idea and outline strategies in implementing UPHs. In Section 7 we discuss these constructions and give some hints towards practical applications. We conclude with some final remarks in Section 8.

2. PASSWORD HASHING

Storing passwords in clear is bad practice, because in case the password database leaks all passwords are leaked as well. This is problematic, because not only gives this the attacker access to the server, but he is accessing the server in a way virtually indistinguishable from a legitimate user. Furthermore, if the user used the same password on other accounts

(so-called *password re-use* is frequently found [11, 12]) the attacker gains access to those accounts as well.

2.1 Classical password hashing

To store passwords more securely, a password *pwd* is usually not stored in plain, but the hash of the password is stored instead, and a random value called *salt* is included in the hash to prevent pre-computation attacks such as rainbow tables [22]. Basically, these schemes work as follows:

- *Store password:* When the server receives a new password *pwd* for the user *uid* he chooses a random string s (called *salt*), computes the hash

  $$h = H(pwd \parallel s),$$

  and stores the tuple (uid, h, s) in the password database.

- *Verify password:* To verify the correctness of a password pwd' for user *uid*, the server retrieves the tuple (uid, h, s) from the database. If

  $$h \stackrel{?}{=} H(pwd' \parallel s)$$

  then the verification succeeds and the provided password is correct, otherwise verification fails.

If an attacker has access to the password hash h and the salt s the attacker can test a large number of passwords $pwd_1, pwd_2, pwd_3, \ldots$ by computing the hashes $h_i = H(pwd_i \parallel s)$ and comparing those to the stored hash h (this is called an *offline guessing attack*). Such an attack typically has a high probability of success. This is usually countered by making the hash function H slower to evaluate.

2.2 Properties of password hashes

Next we discuss the properties that password hashes need to fulfill in order to be secure.

A password hash must be *pre-image resistant*, i.e., given the output h it must be infeasible to determine an input x such that $h = H(x)$. This is one of the classical properties of hash functions and the reason why hash functions were and still are the central ingredient for password hashes. Note that it is not necessary to find the "correct" password, *any* password that hashes to the correct value is sufficient, but usually the correct password will be easier to find if it is selected from a small subset of strings.

The function should be *slow to evaluate*. While this slows down the legitimate server as much as it slows down the attacker, the slow-down affects the attacker much more, because he needs to test many passwords (often millions or billions) to find the correct one. However, the weaker passwords are the slower should the password hash be, and there is a limit on what the server infrastructure is able to handle. This is the origin of our work, as we try to make it more likely that operators tolerate more expensive password hashing by giving password hashing additional use.

One often requires that the slow-down is particularly strong for computing architectures different from CPUs, as legitimate servers typically use CPUs, and attackers currently have best results using GPUs or FPGAs. We do not specifically consider this aspect because we believe that if UPHs are deployed in practice, then login servers will use the most appropriate hardware architecture for the problem at hand.

On determinism: For traditional password hashing as outlined above it is necessary that the hash function H is deterministic, as otherwise verifying a correct password is not possible (or at least it is less efficient). In the sequel we will use password hashes that are not deterministic, however, we can still correctly verify a password because we have a more complex verification function that makes sure that verification is handled correctly. (In some sense, adding salt to traditional hashes is exactly a way to make the password hash not deterministic, and we follow that path even further.)

On parallelism: Sometimes it is claimed that password hashes should be hard to parallelize to increase their security. However, this is incorrect, as the task of password guessing is incredibly parallelizable anyway, as individual password guesses can be verified independently of each other. In contrast, being able to parallelize a single password verification might even be desirable, as it can decrease the latency of password verification in cases when there are more computing cores than logins attempts at a point in time. The constructions we give in the following are easily parallelizable.

3. USEFUL PASSWORD HASHING

A useful password hash (UPH) is a password hash that fulfills two tasks at the same time: First and most importantly, it is a secure password hash, and second, it utilizes the computations required for securely storing passwords to compute some "useful" or "interesting" computational problem that we will call the *base problem* in the sequel.

3.1 General idea

The overall structure of a UPH is somewhat more general as that of normal password hashing:

- *Initialization:* A useful password hashing scheme aims at solving some specific problem. The specific instance of the problem is given as input to the initialization, and we can set up public parameters for subsequent computations. We run

  $$param \leftarrow init(input)$$

  to initialize some global parameters. There is a password database that keeps the password hashes, and in addition we initialize an empty database D that will typically collect partial results.

- *Store password:* Given a password *pwd* for user *uid*, one computes

  $$h \leftarrow create(pwd, param)$$

  and stores the tuple (uid, h) in a database.

  (When casting the "normal" password hash in this framework, *create* would choose a random salt and store the salt as well as the hash in the output tuple.)

- *Verify password:* To verify the correctness of a password pwd' for user *uid*, the server first retrieves the tuple (uid, h) from the database, runs

  $$(b, data, h') \leftarrow verify(param, h),$$

  and accepts the password if and only if b equals *true*. He adds *data* to the database D.

He may also output an updated password hash h' that will replace the previous password hash h in the password database, to ensure that work is not repeated when verifying the same password multiple times. (Typically he will re-run $h' \leftarrow create(pwd, param)$.)

- *Finalize:* Finally, we use the data collected in D to solve the base problem. We run

$$x \leftarrow final(D)$$

to create the answer x. Typically it is very easy to determine when $final(\cdot)$ should be run, e.g., when there are enough entries in D, or $final(\cdot)$ could be run from time to time and checks itself it this condition is met.

For robustness it is desirable to only have simple access patterns to the global data (the database D and some global constants), because several processes will access this memory in parallel, possibly from different servers, so consistency can be an issue here. We assume that the server is interested in solving the base problem, e.g., we can tolerate limited overhead compared with solving the problem directly.

3.2 Terminology

The basic attacks we consider in this work are *offline guessing attacks*, where an attacker learns the content of a server's password database and thus does not require interaction with the server to test and verify password guesses. Hashing the passwords, as discussed above, basically means that the (legitimate) server is playing "against himself", as he is voluntarily spending computational work for verification of passwords that is not necessary (unless the password database leaks). A *lazy server* would decide not to follow that scheme and save computations by deviating from the scheme. (Note that by simply storing the passwords in plain he can always reduce his computational load to basically zero, so we need to assume some external motivation for him to follow the more secure password hashing scheme.)

The *overhead* measures how efficient the conversion from the base problem to the password hash is. Let t_{base} denote the time (e.g., measured in computing cycles or execution time on a specific platform) to solve the base problem using state-of-the art algorithms, and let t_{uph} denote the time to solve the problem using the UPH construction. Then the overhead is $\frac{t_{uph}}{t_{base}} - 1$. An overhead of 0 means that the runtime is the same, while an overhead of 1 means that the UPH construction requires double the time compared to state-of-the art algorithms.

3.3 Properties

Next, we discuss several properties that UPHs should have. Probably the most obvious one is correctness, where we distinguish the following two types of correctness. First, *correct verification* of the password hash, which means that if a user provides the correct password for verification, then the verification should succeed (at least with overwhelming probability). Second, *correctness of the computation* of the base problem, once it terminates. Here we can usually trust the legitimate server to correctly follow the protocol as he is at least semi-trusted and shares some interest in the outcome of the computation, e.g., we can assume that he stores correct data in the database D. (In addition, all problems we use for demonstration in the sequel have the property that solutions are easy to check for correctness.)

Another potential issue originates from using external input (the base problem, composed of the algorithm and it's input) in a security-critical problem. *Resilience against malicious data* means that the construction should be secure against attacks where the problem instance is chosen maliciously.

When selecting a base problem for a UPH, we need to consider the *incentives* of the involved parties. We need to ensure that we select a problem that the honest server has an advantage from, while the attacker trying to brute-force a password database has no advantage (or a far smaller one). For example, using Bitcoin mining as base problem is not ideal, as this would enable an attacker to basically break passwords for free, while mining Bitcoins. Ideally, we would use a problem that the server operator needs to perform anyway, e.g., parts of the SSL handshake, or database queries he has to perform anyway, as those provide practically no value for an attacker.

Furthermore, we need to avoid the possibility of *cherry-picking*, a notion from the context of proof-of-works which means that an attacker can try to select "easier" problem instances and solve only those.

Finally, a general concern (also with the current state-of-the art in password hashing) is that one uses cryptographic algorithms in a way they where not designed for. E.g., when using (iterated) MD5 we use the property that this construction (hopefully) cannot be substantially sped up, which is far from the standard assumption of pre-image resistance. Similarly, when using a UPH we assume that the underlying construction cannot substantially speed up (pre-image resistance of the UPHs is ensured by their construction).

4. CONSTRUCTION 1: BRUTE-FORCING BLOCK-CIPHERS

Our first example for a UPH bases on the problem of a brute-force key-guessing attack against a block cipher with known cleartext/ciphertext pairs. This is a simple construction, but is well-suited to illustrate the basic idea and also demonstrates one way of constructing UPHs from a general class of problems.

4.1 Construction

Let $E_k(m)$ be an encryption scheme with key-space K, message space M, and ciphertext space C. For easier presentation we consider the DES encryption scheme with $|K| = 2^{56}$ and $|M| = |C| = 2^{64}$. We are given a cleartext-ciphertext pair $m_0, c_0 \in \{0,1\}^{64}$, and our task is to find a key $k_0 \in \{0,1\}^{64}$ such that $E_{k_0}(m_0) = c_0$. Randomly drawing an element x from a set X is denoted as $x \leftarrow^R X$.

- *Initialization:* We select a strength parameter γ such that $2^\gamma \cdot t_{DES}$ matches the "intended strength" of the password hash, i.e., the average time for verifying a password. (Here, t_{DES} is the time it takes to evaluate DES on a single block on the target computer architecture.) (The database D is not needed in this first construction.)

- *Store password:* To store a password pwd, first select a random salt r, compute a (pseudo-)random prefix $k_{pref} := H(pwd \parallel r) \in \{0,1\}^{56-\gamma-1}$ (truncating the output if necessary), select a random suffix $k_{suf} \leftarrow^R \{0,1\}^{\gamma+1}$, and let $k_1 := k_{pref} \parallel k_{suf}$. Compute $c_1 :=$

$E_{k_1}(m_0)$ and

$$h := H(pwd \parallel k_1)$$

and store

$$(uid, h, r, c_1)$$

in the password database.

- *Verify password:* To verify a password pwd' for user uid, we first retrieve the matching database entry (uid, h, r, c_1). Furthermore, we have access to the public parameters γ, m_0, c_0.

  The verifier computes $k_{pref} := H(pwd \parallel r)$, iterates over all $k_{suf} \in \{0,1\}^{\gamma+1}$ and tests, for each $k = k_{pref} \parallel k_{suf}$, if

  $$c_1 \stackrel{?}{=} E_k(m_0). \quad (1)$$

  If this condition is met, then (with high probability) we found the key used for preparing the challenge, and we can test if

  $$h \stackrel{?}{=} H(pwd \parallel k).$$

  If this test succeeds that password is correct, and incorrect otherwise.

  In addition, in Equation 1 we can additionally test if

  $$c_0 \stackrel{?}{=} E_k(m_0), \quad (2)$$

  which has almost no overhead. If this second condition is met then we have solved the original challenge, and we can add the key k to the database D.

  If the password verification is successful, a fresh password hash is generated using the routine *store password* using fresh randomness.

- *Finalize:* For the simple scheme described here there is nothing left to do, so we test if D is non-empty and if so we output the key k, which is, with high probability, the correct key. If we have additional message/ciphertext pairs we can test here if the found key matches these pairs as well to increase our confidence.

Intuitively, in the above construction we create a random salt k_1 from a small subset of the keyspace (those with a fixed prefix) and use this salt in the password hash as before. However, we do not store the salt in plain, but give a ciphertext/plaintext pair that identifies it. As the plaintext for this pair is the same plaintext as for the challenge, brute-forcing for the salt is effectively the same as brute-forcing the given challenge. So just adding one comparison per tested key solves the original challenge, virtually without any overhead. (The only drawback is that we cannot systematically search the entire keyspace, but have to rely on probabilistically selecting keys, but the average number of attempts is the same.)

4.2 Analysis

The basic idea in this construction is that we take a problem where we have a large space to search (K), divide this into smaller subsets of tractable size (those with a fixed prefix k_{pref}) that eventually span the entire space, and force the verifier to search that smaller spaces. The crucial question here was how we can manage to create a problem instance (the pair m_0, c_1) from the smaller space (quickly) that, when solved, helps us towards solving the original problem instance (the pair m_0, c_0).

Parameter selection: The work the legitimate server has to do is to search the space with prefix k_{pref}, which contains $2 \cdot 2^\gamma$ keys, to find the index k_1 which was chosen randomly from that subspace. Regardless of the order he uses to pick the candidates, he will need an expected number of 2^γ guesses to find the correct index, and to test each guess he has to compute a DES encryption, so in total he requires time $2^\gamma \cdot t_{DES}$.

Correctness: For reasonable parameters γ the above scheme is correct with very high probability. Assuming DES behaves like a random cipher, we get an expected number of $2^{\gamma+1-64}$ keys in the search-space that correctly decrypt c_1, and even if we hit by accident such a bad key, if we choose the candidate keys (somewhat) randomly, we will likely hit another (hopefully the correct) key on the next login attempt, so no long-term harm is done. Also, we can easily incorporate a second message-ciphertext pair into the scheme to reduce the error rate.

Overhead: An important question is comparing the runtime for solving the base problem directly or solving it using the UPH. For the above scheme, we see that the overhead is minimal, i.e., solving the problem takes about the same computational resources in both cases.

First, note that the expected number of DES computations is the same in both cases, as k_1 is chosen uniformly at random. Furthermore, the main loop of the password hash is iterating over a consecutive block of DES keys, which is what a brute-forcer has to do as well.

So the main source of overhead in our implementation comes from storing the passwords, which he basically has to do once for each verification. Storing requires one DES operation and minimal other operations, which is negligible for any reasonable values of γ.

Another source of overhead is that unsuccessful login attempts do not allow the server to store a fresh copy of the password, which means that the same computations are performed on the next login attempt. This can be somewhat mitigated by storing several hashes for the same account and using a fresh or random one for each login attempt (also see the discussion in Section 7).

Lazy server: A server that wants to spend less time in the verification can skip the test in Equation 2, but anything else is required for the actual password verification. This one equality test is so fast that we do not think it constitutes a problem, as the server opted to deploy the system in the first place we assume that he has at least some interest in solving the base problem.

5. CONSTRUCTION 2: DISCRETE LOGARITHMS

Our second construction uses a variant of Pollard's Rho algorithm to solve discrete logarithms in cyclic groups.

5.1 (Parallel) Pollard's Rho algorithm

Let us briefly review the discrete logarithm problem. Let g be a generator of a finite cyclic group $G = \langle g \rangle$ of order N, and $h \in G$ another element. The *discrete logarithm* of h to the basis g is an element $x = \log_g(h)$ such that $g^x = h$, where we usually require that $0 \leq x < N$.

In the following we consider groups of prime order only, i.e., where N is a prime number, and the discrete logarithm is unique. Most applications in cryptography such as the Diffie-Hellman key exchange and ElGamal encryption are based on prime order groups. The hardness of solving discrete logarithms depends on the underlying group G, but several groups are known where it is believed to be hard.

For generic groups, *Pollard's Rho algorithm* is the fastest algorithm known (ignoring constant terms). It bases on the idea that, if we choose random elements of the form $g^{a_i} \cdot h^{b_i}$ for random numbers $a_i, b_i \leftarrow^R [0, \ldots, N-1]$, and we find two such elements that collide, then we can compute the discrete logarithm. I.e., if

$$g^{a_1} \cdot h^{b_1} = g^{a_2} \cdot h^{b_2}$$

then

$$dlog_g(h) = (a_2 - a_1) \cdot (b_1 - b_2)^{-1} \bmod N \qquad (3)$$

if $b_1 - b_2$ is invertible, i.e., if $b_1 \neq b_2 \bmod N$ (as N is prime).

Pollard's Rho algorithm improves the memory requirements of the direct approach to find those elements. It uses a function $f\colon G \to G$ which should resemble a "random function", a common choice being

$$f(x) = \begin{cases} h \cdot x & \text{if } x \in S_1, \\ x^2 & \text{if } x \in S_2, \\ g \cdot x & \text{if } x \in S_3. \end{cases}$$

for a partition S_1, S_2, S_3 of G that have roughly equal size. Now, starting at a random point $g^{a_1} \cdot h^{b_1}$ and successively applying $f(x)$, we get a sequence of group elements. This sequence will eventually reach a point that it has visited previously, and will necessarily repeat from that point on. One can use Floyd's cycle-finding algorithm to detect such cycles and find two elements that are the same but have, with high probability, different exponents a_i, b_i.

The original algorithm is inherently sequential, so a parallelizable version was proposed by Oorschot and Wiener [32]. It uses the notion of *distinguished points*, e.g., elements of G where the binary encoding starts with a certain number of 0s. Every processor starts with a separate uniformly chosen point $y^{(i)} = g^{a^{(i)}} \cdot h^{b^{(i)}}$, and uses the function f to produce further points. Once a processor reaches a distinguished point $y_j^{(i)} = g^{a_j^{(i)}} \cdot h^{b_j^{(i)}}$ it sends the triple $(y_j^{(i)}, a_j^{(i)}, b_j^{(i)})$ to a central server. Once the server sees two records with the same group elements he can compute the discrete logarithm in the same way as before.

5.2 Construction

The approach here is conceptually different from the above, as it is hard for this algorithm to construct the designated target point. Instead, we use another general idea that considers a couple of threads in parallel, where we have computed the output for one of the threads before.

- *Initialization:* As public parameters we have the description of a group G of prime order N, and two elements g, h where we want to learn the discrete logarithm of h to the base g in G.

  We select a security parameter γ such that $\gamma \cdot t_{seq}$ gives the desired computation time of the legitimate server, and t_{seq} is the average time to compute a sequence until the first designated point is reached.

- *Store password:* For storing a password pwd for a user uid we select a random salt r, compute (pseudo-)random elements $a^{(i)}, b^{(i)} := H(pwd \parallel r \parallel i) \in [0, \ldots, N-1] \times [0, \ldots, N-1]$ for $i = 1, \ldots, 2 \cdot \gamma$ (using an appropriate encoding), we choose one index $i_0 \leftarrow^R \{1, \ldots, 2 \cdot \gamma\}$, we iterate $y_{i_0} = g^{a^{(i_0)}} h^{b^{(i_0)}}$ for x times until finding the first designated point and split the output into two halves $s \parallel t := f^x(y_{i_0})$. We add the designated point to the database D.

  We compute

  $$h := H(pwd \parallel s)$$

  and store the tuple

  $$(uid, h, t, r).$$

- *Verify password:* To verify if a provided password pwd' is the correct one for user uid, first retrieve the correct entry (uid, h, t, r) from the database and re-compute the $a^{(i)}, b^{(i)} := H(pwd \parallel r \parallel i)$.

  Compute the $y^{(i)} = g^{a^{(i_0)}} h^{b^{(i_0)}}$ for $i = 1, \ldots, 2 \cdot \gamma$ and iterate f for x times to reach the first designated point (keeping track of the exponents as described above) and split the output as $s^{(i)} \parallel t^{(i)} := f^x(y^{(i)})$.

  If one of the $t^{(i)}$ matches t then use the corresponding other half $s^{(i)}$ to compute

  $$h' := H(pwd' \parallel s^{(i)}).$$

  If $h = h'$ then accept the password.

  In addition, we store all designated points with the corresponding exponents a, b in the database D. Finally, we store a fresh hash in the password database.

- *Finalize:* From time to time, we test if two entries exists with the same element x. In this case we can use Equation 3 to compute the discrete logarithm from the stored exponents.

5.3 Analysis

The basic idea in this construction is different from the first construction. Whereas in the first construction we could divide the search-space (of keys) and efficiently create one instance by encrypting with a random key, this does not really work here because the function f is pretty much random and we cannot (efficiently) sample a point from later in the sequence (this takes about as much time as actually solving it). So we have to take a different approach. We consider several chains in parallel (which we can do without loosing efficiency due to the construction of the parallel Pollard's Rho algorithm). We compute a random "salt" s by advancing one random chain until we get a designated point. This element serves as random element similar to the first construction: we use it as salt to randomize the actual password hash, and use the other half as identifier to allow the verifier to see when he found the correct group element. (Alternatively, we could use the hash of that element for the same purpose, or define that a password is valid if "any" of the candidate elements has the correct hash, but both alternatives would increase the overhead.)

Parameter selection: We need to compute, on average, $2 \cdot \gamma/2 = \gamma$ of the chains in order to find the correct designated (end-)point. As f behaves "randomly" it seems unlikely that

one can decide substantially earlier than when arriving at the designated point which chain is the correct one. This results in an average complexity of $\gamma \cdot t_{seq}$.

Correctness: Correctness is easy to see, as with high probability only the "correct" group element will have the correct tag t and thus we will use the correct salt s as input to the hash function.

Overhead: This construction is more general than the first one, but also has a higher overhead, as we have to evaluate one chain twice, once when storing and once when verifying a password. The number of parallel chains is a critical parameter here, as it directly influences the overhead (which is lower the more parallel chains we are using), but also the runtime of verification.

Lazy server: As before, a lazy server can avoid a couple of equality tests if he only is interested in verifying a password without doing additional work, which hardly decreases his computations.

6. CONSTRUCTION 3: INTEGER FACTORIZATION

Finally, we will sketch a construction based on the quadratic sieve for factoring integers. We provide a little less details on this construction to keep the presentation easy to follow and not obstruct the main ideas.

6.1 Quadratic sieve

We consider the problem of *factoring integers* that are composed of two prime factors of (approximately) equal size $N = p \cdot q$, where the task is to find the factors p and q, given N.

The *quadratic sieve* [26, 27] tries to find two integers $a \neq b$ such that

$$a^2 = b^2 \bmod N.$$

Then $\gcd(a \pm b, N)$ is a non-trivial factor with reasonable probability.

For finding such a and b, we consider the polynomial

$$Q(x) = (x + \lfloor\sqrt{N}\rfloor)^2 - N. \tag{4}$$

Our goal is to find values x such that $Q(x)$ is a smooth number, i.e., a number that has only small factors, where the bound needs to be specified later. (The prime numbers smaller than the bound are called the *factoring base*.) If we have found enough numbers such that the $Q(x_i)$ are smooth, we can find a subset $x_1, \ldots, x_r$ of those such that the product $Q(x_1) \cdot \ldots \cdot Q(x_r)$ is a square. Writing $\tilde{x} := x + \lfloor\sqrt{N}\rfloor$ this means

$$a^2 = Q(x_1) \cdot \ldots \cdot Q(x_r) = (\tilde{x}_1 \cdot \ldots \cdot \tilde{x}_r)^2 = b^2 \bmod N$$

which will allow us to factor N.

There is an efficient method to test smoothness of $Q(x)$ for an entire range of values of x, the so-called *sieving step*; the details are not relevant for our construction and we refer to [27]. For a smooth number this gives us, for each prime in the factor base, the parity of the exponent. There are different polynomials $Q(x)$ that can be used; this was used before for the "factoring by email", and we will use it to split the work into smaller work packages.

6.2 Construction

The construction works as follows:

- *Initialization:* Public parameters are the number N to be factored, and the parameters required for the quadratic sieve, i.e., the range of x to be checked and the smoothness bound. (Both parameters also influence the running time and the overhead, see discussion below.)

  Furthermore, we choose a hardness parameter γ such that $\gamma \cdot T_{QS}$ is the designated average time for a verification, where T_{QS} is the time required for running a complete sieving step (for a single polynomial $Q(X)$ and x from the chosen range).

  The database D, initially empty, will contain the smooth numbers found, including the vector giving the parity for the exponents of the prime numbers in the factor base.

- *Store password:* To store a password *pwd* for a user *uid* we select a random salt r, compute $2 \cdot \gamma$ (pseudo-)random polynomials $Q_1(x), \ldots, Q_{2\cdot\gamma}(x)$ from the set of polynomials suitable for factoring, using $H(pwd \parallel r)$ as randomness for the selection algorithm (possibly extending if required) choose a random index $j \leftarrow^R \{1, \ldots, 2\cdot\gamma\}$ and perform quadratic sieving with $Q_j(x)$. If we find smooth integers we add this information to D. We hash the entire output of the sieving step and split it in two halves

  $$s \parallel t := H(data).$$

  As before, we compute

  $$h := H(s \parallel pwd)$$

  and store the tuple

  $$(uid, h, t, r).$$

- *Verify password:* To verify if a provided password pwd' is correct for the user *uid*, one first retrieves the matching entry (uid, h, t, r) from the password database, and re-computes the polynomials $Q_1(x), \ldots, Q_{2\cdot\gamma}(x)$ as before.

  He runs the sieving step for the polynomials $Q_1(x), \ldots, Q_{2\cdot\gamma}(x)$, hashes the output for each $s' \parallel t' := H(data)$ to obtain the values s', t' as above, and compares if $t' = t$.

  If a t' matches, we use the corresponding other half s' to compute

  $$h' := H(s' \parallel pwd'),$$

  and if $h = h'$ then accept the password.

  In addition, we add each smooth number to the database D including the factorization vector.

- *Finalize:* From time to time, we test if enough smooth numbers have been found. (We need as many smooth numbers as there are primes in the factoring base, so that the resulting matrix can be solves.)

  We then execute the so-called linear algebra step of the quadratic sieve to find a subset of the smooth numbers so that each exponent for the primes in the factor base is even, thus a square again. With reasonable probability this already gives the factorization. Otherwise we re-run the linear algebra step with another subset of smooth numbers and try again.

6.3 Analysis

The construction is similar to the previous construction.

Parameter selection: Again, we need on average $2 \cdot \gamma/2 = \gamma$ executions of the sieving step, and, as we need to reproduce the *entire* output of the sieving step, most likely there are no shortcuts in computing this. This results in an average complexity of $\gamma \cdot t_{QS}$.

Correctness: Correctness is easy to see, as finding an incorrect s' for the target t is extremely unlikely.

Calculation of the *overhead* and for the *lazy server* setting is the same as before.

7. DISCUSSION

Finally, we discuss some aspects of UPHs that are important for practical deployment.

Multiple servers: When multiple login servers are deployed, consistency across those servers could be an issue. Fortunately, the hashes are independent so we can write a fresh hash/problem on each server, thus we do not need consistency across the servers. This potentially prevents some optimizations that require consistent state between the instances. Careful implementations might be able to keep consistent state.

Changing to another problem instance: Once the base problem is solved, the usefulness of the construction is gone. However, as the password is still stored using the old problem, the hashing scheme still needs to be supported; if a user logs in rarely a scheme might need to be supported for years.

This can be prevented by storing a users password in several ways, for example once using the UPH and once with a "traditional" password hash. While the problem is not solved we use the UPH version, after the problem is solved we delete the UPH version and use the version stored using the traditional password hash, creating new instances of the UPH scheme (for a new base problem) as soon as the user logs in the next time.

Storing multiple versions of the same users password also has the advantage that, in case there is an error storing the new instance of the UPH, that we can resort to a stable hash that is never changed, potentially stored in another (static) database. This can also simplify the event when a user changes his password, where we can simply delete all UPH instances and update the static hash.

Other computational problems: An obvious and very interesting question is for what kind of computational problem UPHs exist. This is beyond the scope of this work, but we believe that a large number of parallelizable search problems can be cast as a UPH.

8. CONCLUSION

We have introduced the notion of *useful password hashes* UPHs, that actually put the computing cycles that are spent for computing password hashes to good use by solving various cryptographic problems. We showed constructions brute-forcing block-ciphers, computing discrete logarithms using Pollard's Rho algorithm, and factoring integers using the quadratic sieve. The constructions we have demonstrated should be adaptable to a number of search problems, and we conjecture that many more problems can be solved using UPHs.

We hope that this will motivate the server operators to make use of stronger password hashes, as the cycles are not actually wasted. Currently used password hashes offer hardly any (iterated MD5) to medium (bcrypt) security only, putting our passwords at risk.

Acknowledgments

The author would like to thank the anonymous reviewers, the shepherd Rainer Böhme, and all participants of NSPW 2013 for the valuable feedback and interesting discussions.

9. REFERENCES

[1] Atom. HashCat. Online at `http://hashcat.net/oclhashcat-plus/`.

[2] Jörg Becker, Dominic Breuker, Tobias Heide, Justus Holler, Hans Peter Rauer, and Rainer Böhme. Can we afford integrity by proof-of-work? scenarios inspired by the bitcoin currency. In *Proc. of Workshop on the Economics of Information Security (WEIS)*, 2012.

[3] M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.

[4] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, 2012.

[5] William E. Burr, Donna F. Dodson, and W. Timothy Polk. Electronic authentication guideline: NIST special publication 800-63, 2006.

[6] Claude Castelluccia, Abdelberi Chaabane, Markus Dürmuth, and Daniele Perito. Omen: An improved password cracker leveraging personal information. In submission, 2013.

[7] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from Markov models. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 2012.

[8] J. A. Cazier and D. B. Medlin. Password security: An empirical investigation into e-commerce passwords and their crack times. *Information Security Journal: A Global Perspective*, 15(6):45–55, 2006.

[9] Solar Designer. John the Ripper. Online at `http://www.openwall.com/john/`.

[10] Markus Dürmuth, Tim Güneysu, Markus Kasper, Christof Paar, Tolga Yalçin, and Ralf Zimmermann. Evaluation of standardized password-based key derivation against parallel processing platforms. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, volume 7459 of *Lecture Notes in Computer Science*, pages 716–733. Springer, 2012.

[11] D. Florencio and C. Herley. A large-scale study of web password habits. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

[12] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. The domino effect of password reuse. *Communications of the ACM*, 47(4):75, April 2004.

[13] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Proc. IFIP TC6/TC11 Joint Working Conference on Secure Information*

Networks: Communications and Multimedia Security (CMS 99), pages 258–272. Kluwer, B.V., 1999.

[14] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000. `http://tools.ietf.org/html/rfc2898`.

[15] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[16] D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proc. USENIX UNIX Security Workshop*, 1990.

[17] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *CHI 2011: Conference on Human Factors in Computing Systems*, 2011.

[18] Marcus Leech. Chinese lottery cryptanalysis revisited: The internet as a codebreaking tool. RFC 3607, available online at `ftp://ftp.rfc-editor.org/in-notes/rfc3607.txt`, 2003.

[19] S. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008.

[20] Robert Morris and Ken Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.

[21] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.

[22] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *23rd Annual International Cryptology Conference (CRYPTO 2003)*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[23] The password meter. Online at `http://www.passwordmeter.com/`.

[24] Password hashing competition. Online at `http://password-hashing.net`, 2013.

[25] Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan'09*, 2009.

[26] Carl Pomerance. Analysis and comparison of some integer factoring algorithms. *Math. Centre Tract 154, Computational Methods in Number Theory, Part I*, pages 89–139, 1982.

[27] Carl Pomerance. A tale of two sieves. *Notices of the AMS*, 43(12):1473–1485, 1996.

[28] Niels Provos and David Mazieres. A future-adaptable password scheme. In *Proc. of 1999 USENIX Annual Technical Conference*, pages 81–92, 1999.

[29] Jean-Jacques Quisquater and Yvo G. Desmedt. Chinese lotto as an exhaustive code-breaking machine. *Computer*, 24(11):14–22, 1991.

[30] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.

[31] E. H. Spafford. Observing reusable password choices. In *Proceedings of the 3rd Security Symposium*, pages 299–312. USENIX, 1992.

[32] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[33] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry 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 2010)*, pages 162–175. ACM, 2010.

[34] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy*, pages 391–405. IEEE Computer Society, 2009.

[35] Openwall Community Wiki. John the Ripper benchmarks, April 2012. `http://openwall.info/wiki/john/benchmarks`.

[36] T. Wu. A real-world analysis of kerberos password security. In *Network and Distributed System Security Symposium*, 1999.

[37] M. Zviran and W. J. Haga. Password security: an empirical study. *J. Mgt. Info. Sys.*, 15(4):161–185, 1999.