Harden Zero Knowledge Password Proofs Against Offline Dictionary Attacks

Gijs Van Laer
gvanlae1@jhu.edu

Rono Dasgupta
rdasgup4@jhu.edu

Aditya Patil
apatil4@jhu.edu

Matthew Green
mgreen@cs.jhu.edu

December 12, 2016

Abstract

Traditional authentication systems offer ease of use but the security they provide often proves to be inadequate. Hackers use means to gain access to company servers and steal entire databases of password hashes. These hashes are then subject to offline dictionary attacks resulting in the disclosure of millions of passwords. Such password breaches have become commonplace and have caused several major companies to face major losses. Password reuse is a common practice among users and a password breach disclosing a single password on a particular service could result in a user also losing access to their other accounts. Solutions such as multi-factor authentication add some level of security but do not completely solve the problem. There is a need to move towards stronger authentication schemes that do not compromise on ease of use, both for the user and the service provider.

In this paper, we propose a novel authentication protocol that is proven hard against offline dictionary attacks. Our protocol implements a combination of a Zero Knowledge Password Proof and a sequentially memory hard hash function. In a concrete instantiation of the protocol, we use Schnorr's Zero Knowledge Password Proof combined with the Fiat-Shamir Heuristic for the Zero Knowledge Password Proof and scrypt for the sequentially memory hard hash function. We also describe a library implementing our protocol that we have developed along with an example web application that uses the library. Lastly, we provide performance tests for the various components of our protocol and show that the protocol is extremely efficient.

1 Introduction

The most commonly used method of authenticating a user involves a user chosen password that is sent to a server, which then computes the hash of the password and stores it securely in a database along with a randomly generated salt value. This type

of authentication system has been in use for several years and has become the default standard in the industry. While this approach to authentication can be considered safe and acceptable in some cases, it has several major drawbacks. Service providers often make errors while implementing this kind of authentication system. These errors can range from not using a strong hashing algorithm to not storing the password with a salt value. Attackers then use other methods to steal entire databases of password hashes and perform successful brute force or dictionary attacks with minimum effort. Large corporations such as LinkedIn and Sony Pictures have made errors of this nature in recent years and have become victims of enormous password breaches resulting in major losses [Kam12, Fei14]. However, even under the most ideal conditions, it has been observed that even strong hashes of passwords used along with salts can be cracked using a password cracker such as hashcat [Has16] and efficient hardware resources such as Graphics Processing Units (GPUs). The problem here is that currently accepted strong hashing algorithms such as the SHA family of algorithms or even MD5 are extremely fast. These hash algorithms were initially designed to be computationally fast because they were designed to quickly check data integrity rather than for password storage purposes. As such, there is a need to shift towards more memory intensive and slow hashing algorithms such as PBKDF2, bcrypt, scrypt and Argon2. [Hun12] These algorithms fall under a category of algorithms called Password-Based Key-Derivation Functions which will be discussed later in the paper.

There are also other factors that directly affect the security of authentication systems. These factors include the availability of a trusted third party and a secure transmission network. For these purposes, several stronger authentication protocols have been proposed that try to solve the issue of passwords being stolen as a result of server compromise. The common goal that these protocols try to achieve is preventing the transmission of passwords over the wire and/or storage of passwords in a hash database.

In doing so, these protocols can ensure that the password is never made available to a third party. In other words, if a user A authenticates itself to a server B, then the server B or a malicious third party C cannot re-use the credentials of A to authenticate themselves. These types of authentication protocols achieve this by using the concept of a user sharing knowledge of a secret with the verifying server instead of sharing the secret itself. One way of doing this is using some form of challenge-response mechanism to authenticate the user. In this kind of authentication scheme, the authentication server sends the user a challenge (usually just a random byte sequence) and the user computes a cryptographic function on this challenge along with the secret held by them and sends it back to the server. The server verifies the value computed by the user and authenticates the user if the value matches the one stored on the server. The cryptographic function used in such protocols can be based on Message Authentication Codes for private-key systems and Digital Signatures for public-key systems. Kerberos [NT94] and the Needham-Schroeder protocol [NS78] are examples of this kind of authentication system. These challenge-response schemes can however reveal some partial information to an adversary who can try to select chosen challenges and obtain information from the responses to them.

To solve this, authentication protocols employ a stronger form of authentication

schemes called Zero Knowledge identification schemes. Zero Knowledge schemes allow a prover (user) to prove knowledge of a secret while not revealing any information about the secret itself to the verifying server. This type of authentication scheme is safe to use when a user does not want to trust the server with their password. This, in turn, also prevents any adversarial third party from stealing the password from the server. There are several Zero Knowledge identification schemes in existence such as the Feige-Fiat-Shamir identification protocol [FFS88], the Guillou-Quisquater identification protocol [GQ88] and the Schnorr identification protocol [Sch91]. These protocols are often called Zero Knowledge Password Proofs (ZKPP). ZKPPs are used in conjunction with other secure methods to make a newer and stronger type of authentication protocols called Password- authenticated key agreement protocols. [AMV96]

Password-authenticated key agreement (PAKE) is a cryptographically secure method where two parties can establish a shared cryptographic key based on the condition that one or more parties has knowledge of a password. Augmented PAKE protocols are a type of PAKE that are more specific to client/server authentication scenarios. Augmented PAKE protocols have the unique property where the data stored by the server is not plaintext-equivalent to the password chosen by the user. This means that an attacker cannot directly use credentials stolen from the server to login. Instead, the attacker would have to perform a brute force on the password itself which can be solved easily using methods such as rate limiting. Common examples of Augmented PAKE protocols are Secure Remote Password(SRP) [Wu97], Augmented EKE [SMB16] and B-SPEKE [Jab96].

2 Literature review

2.1 Secure Remote Password (SRP) Protocol

The SRP protocol is a PAKE protocol first proposed by Thomas Wu from Stanford University. The SRP protocol is ideal for authentication over an untrusted network. It is based on a new construction which the author calls Asymmetric Key Exchange. This construction exchanges keys between the client and the server and uses this key to verify that both parties have knowledge of their passwords. Asymmetric Key Exchange does not perform any kind of encryption on any of the protocol flows. It uses predefined mathematical relationships and combines the exchanged ephemeral values with the defined password parameters. The security of SRP relies on the hardness of Discrete Logarithms and requires the use of a safe prime of the form $n = 2p + 1$ where p is a large prime number. SRP is also significantly faster than other comparable PAKEs. The SRP protocol's working can be briefly described in the following steps:

1. The user sends their username to the server.
2. The server looks up the user's password entry and fetches the corresponding password verifier v and salt s and sends s to the user.
3. The user computes their private key x using s and password P.
4. The user generates a random number a, computes an ephemeral public key A, and sends it to the server.

5. The server generates a random number b, computes an ephemeral public key B, and sends it back to the user, along with the randomly generated parameter u.
6. The user and the server compute the common exponential value S using the values available to them.
7. Both the user and the server hash the exponential S into a cryptographically strong session key.
8. The user sends the server $M[1]$ as evidence that they have the correct session key, where $M[1]$ is the hash of the corresponding public keys of the user and the server, and the session key.
9. The server computes $M[1]$ and verifies that it matches the user's value.
10. The server sends the user $M[2]$ as evidence that it has the correct session key. The user also verifies $M[2]$, accepting it only if it matches the server's value, where $M[2]$ is the hash of the public key of the user, $M[1]$ and the session key.

The protocol is resistant to active dictionary attacks [Wu97]. The issue with the SRP protocol is that it does not provide complete protection against offline dictionary attacks. The password hash is not computed using a memory-intensive function which makes it susceptible to password cracking using very inexpensive hardware. If an adversary gains access to the statement that is stored on the server, they can brute force using a dictionary of passwords and be able to guess and retrieve the password in a reasonable amount of time.

2.2 Password-Based Key-Derivation Functions

Password-Based Key-Derivation Functions (PBKDF) are key derivation functions that can derive one or more cryptographic keys from a password. These Password-Based Key-Derivation Functions are intended to be computationally intensive, so that they take longer to compute than commonly used one-way hash functions, adding hundreds of milliseconds or more to the computational time. The advantage of using these functions for authentication purposes is that a user who has knowledge of their password will only make the function computation once or a couple of times. However, an attacker trying to guess the password using brute force techniques will have to compute the function several billions of times, at which point the time it requires to compute all possibilities for a single password becomes completely restrictive. Brute-force and dictionary attacks thus become non-feasible, provided that the password is not easy to guess and is stored along with a salt. There are several widely accepted Password-based key derivation functions such as bcrypt [NP99], Argon2 [AB15] and scrypt [Per09]. The issue with some password-based key derivation functions is that while they are time-intensive, they are not necessarily memory-intensive. These functions do not have high resource demands. As a result, functions such as PBKDF2, introduced by RSA Laboratories, can be implemented using hardware such as Application-specific integrated circuits (ASIC) and Field-programmable gate arrays (FPGA) which are relatively cheap to obtain. This allows an attacker to implement the key derivation function in hardware and parallelize their attack on a large scale to search on various parts of the key

space. This could potentially reduce the time frame to one that is feasible and make the entire attack not too financially prohibitive. [Per09]

Scrypt is a type of Password-based key derivation function which is designed to have higher memory resource usage demands compared to others. This can greatly limit the kind of large scale parallel attacks that an attacker could deploy due to much greater financial resource requirements. Scrypt does this by implementing a kind of time-memory trade-off. The algorithm generates a large vector of pseudorandom bit strings. Each element of the vector can be generated dynamically and stored only one at a time in memory. However, the generation of each element is computationally expensive which makes this infeasible. Moreover, each element of the vector is accessed by the algorithm multiple times during its execution life cycle. Thus, for an attacker to reduce the memory requirements, they would have to deal with a significant trade-off in the time required. The attacker either has to make a hardware implementation with very heavy memory resources to run the algorithm faster or the attacker has to rely on a much slower algorithm if they do not have the required memory resources. Scrypt is thus ideal for use as a hashing algorithm in an authentication system. [ACP+16]

3 Preliminaries

First, we will look at the following two concepts that we need for our construction. We need sequential memory-hard functions as described in [Per09]. These are efficient functions to protect passwords against dictionary attacks, as these functions are configurable to keep up with the ever improving hardware. The second concept that we need is zero-knowledge proofs of knowledge, which is a protocol where the client will prove to the server that it knows the password without revealing the password directly to the server.

3.1 Sequential memory-hard functions

The concept of sequential memory-hard functions was first described by Percival in [Per09]. He introduces this novel technique to reduce the attackers advantage of being able to parallelize the computations. This is done by adding the use of more RAM, therefore, while computing power is often easily and cheaply available (by using GPUs), it is much more expensive to provide enough RAM for parallel computations that require large amounts of memory.

To parameterize not only the operation count, but also the memory usage, Percival introduced the following definition:

Definition 1 *A* memory-hard algorithm *on a Random Access Machine is an algorithm which uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\epsilon})$.*

Such an algorithm uses roughly the same amount of memory space as it uses operations. This balance ensures the efficiency for normal users, because they have enough memory available for one such computation, but makes it much harder to do a brute-force attack, where a lot of memory is needed for all the operations that need to be performed.

There exist methods to translate an algorithm that uses memory, into an algorithm that uses more operations. Therefore, a memory-hard algorithm might still be highly parallelizable. To mitigate this problem, Percival introduces the following definition of a *sequential* memory-hard function:

Definition 2 *A* sequential memory-hard function *is a function which*

(a) can be computed by a memory-hard algorithm on a Random Access Machine in $T(n)$ operations; and

(b) cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.

This means that not only is the fastest sequential algorithm memory-hard, but any parallel algorithm cannot significantly decrease the running time.

The construction that Percival introduces is called *scrypt*, and is efficiently implemented in most languages, which makes this a very good candidate to use for our construction. Moreover, scrypt has been proven maximally memory-hard by Alwen et al. [ACP+16].

In the meantime, some other memory-hard functions were invented and these can all be used to implement the protocol that is presented in this paper. [BDK15, BK15]

3.2 Zero-knowledge proofs of knowledge

Zero-knowledge proofs were first introduced in 1989 by Goldwasser, Micali, and Rackoff [GMR85]. It is a cryptographic protocol between two parties, a prover and a verifier, in which the prover convinces the verifier that a certain statement is true, without revealing any other information. It was proven that there exist zero-knowledge proofs for all languages in NP. This means that given a statement $x \in L$, where L is an NP-language, the prover who has a witness w for x can prove to the verifier that $x \in L$ without revealing any information about the witness w.

Definition 3 (Interactive Zero-Knowledge Proof) *A pair of ITMs (P, V) is an interactive zero-knowledge proof system for a language L if P and V are PPT machines and the following properties hold:*

- ***Completeness**: For every $x \in L$,*

$$Pr\left[Out_V[P(x) \leftrightarrow V(x)] = 1\right] = 1$$

- ***Soundness**: There exists a negligible function $\nu()$ s.t. $\forall x \notin L$ and for all adversarial provers P^*,*

$$Pr\left[Out_V[P^*(x) \leftrightarrow V(x)] = 1\right] \leq \nu(|x|)$$

- ***Zero-knowledge**: for every non-uniform PPT adversary V^*, there exists a PPT simulator S such that for every non-uniform PPT distinguisher D, there exists a negligible function $\nu(.)$ such that for every $x \in L, w \in R(x), z \in \{0,1\}^*, D$ distinguishes between the following distributions with probability at most $\nu(n)$:*

$$\{View_{V^*}[P(x,w) \leftrightarrow V^*(x,z)]\} \text{ and } \{S(1^n, x, z)\}.$$

This notion was extended to Non-Interactive Zero-Knowledge proofs which are preferred in most applications. In our setting we use a more specific proof system which is known as a proof of knowledge. Where the prover convinces the verifier he knows a certain secret x, such as a password.

To denote these protocols we will use the notation that was introduced by Camenisch and Stadler [CS97]. For example, $PoK\{(x) : y = g^x\}$ denotes a zero-knowledge proof of knowledge of an integer x such that $y = g^x$ holds, where everything on the left side of the colon is secret, while all other variables that occur on the right side are considered public.

In our instantiation we will use the more specific Schnorr protocol [Sch91], for the following system:

$$PoK\{(x, y) : z = g^x h^y\},$$

more precisely y will be of the form $f(x)$ for a certain function f. However, our protocol can be instantiated with any zero-knowledge proof of knowledge of which there are plenty of efficient examples in the literature [GS07, BCKL08, CDS94, Bra97, CNS07, CCS08, Bou00, Gro06].

3.2.1 Schnorr's Proof of Knowledge [Sch91]

Given a cyclic group G_q or order q, a generator g for this group, and $y = g^x$, Schnorr's protocol proceeds as follows:

Schnorr's protocol

Prover		**Verifier**
$r \xleftarrow{\$} \mathbb{Z}$		
$t = g^r$	$\xrightarrow{\text{t}}$	
	$\xleftarrow{c}$	$c \xleftarrow{\$} \mathbb{Z}$
$s = r + cx$	$\xrightarrow{\text{s}}$	accept if $g^s = ty^c$

This structure is also what is called a Σ-protocol, because it has three rounds:

- Commitment phase: where the prover commits to a certain value
- Challenge phase: the verifier sends a challenge to the prover
- Opening phase: the prover opens the commitment based on the challenge

The good thing about such a Σ-protocol is that by using the Fiat-Shamir heuristic [FS87] one can translate the protocol in a non-interactive version by using a hashed version of the commitment instead of the challenge. This heuristic can only be proven secure in the random-oracle model. For more details about this heuristic see [FS87].

4 Definitions

We propose the following definition for zero-knowledge password proofs that are hard against offline dictionary attacks. We denote the password proof by the triplet (K, C, S), where K is the setup algorithm, C is the client side of the protocol and is the same as the prover for the password proof, and S is the server side and corresponds to the verifier in the password proof.

Definition 4 (ZKPP hard against offline dictionary attack) *Given a password proof*

$$(K, C, S),$$

we say it is hard against offline dictionary attacks if it has the following properties:

- ***(Hard against eavesdropper)*** *The password proof has the zero-knowledge property, i.e.: for every non-uniform PPT adversary S^*, there exists a PPT simulator $\mathbb{S}$ such that for every non-uniform PPT distinguisher D, there exists a negligible function $\nu(.)$ such that for every $x \in L, w \in R(x), z \in \{0,1\}^*, D$ distinguishes between the following distributions with probability at most $\nu(|x|)$:*

$$\{View_{S^*}[C(x,w) \leftrightarrow S^*(x,z)]\} \text{ and } \left\{\mathbb{S}(1^{|x|}, x, z)\right\}.$$

- ***(Hard against server compromise)*** *Given the information on the server side d_S, i.e. all information that is stored on the server side. For every non-uniform PPT adversary A, there exists a negligible function $\nu(.)$ such that:*

$$Pr[A(D, d_S) = x] \leq \nu(|D|),$$

where D is a dictionary for x, and:

$$\text{Expected running time of } A \gtrsim \frac{|D|}{2} T(|x|),$$

where $T(.)$ is the running time of the proof system.

- ***(Hard against client compromise)*** *Given the information on the client side d_C, i.e. all information that is stored on the client side. For every non-uniform PPT adversary A, there exists a negligible function $\nu(.)$ such that:*

$$Pr[A(D, d_C) = x] \leq \nu(|D|),$$

where D is a dictionary for x, and:

$$\text{Expected running time of } A \gtrsim \frac{|D|}{2} T(|x|),$$

where $T(.)$ is the running time of the proof system.

Note that for data on client side d_C, we only consider data that is stored by the protocol, in particular, we do not consider storage of the password by the user's themselves. This last property might seem a little strange, as in a general authentication protocol usually there is no information stored on the client side. Nevertheless, in our construction we cache certain data on the client side for performance reasons, therefore, we need to prove security when this data is leaked.

5 Protocol

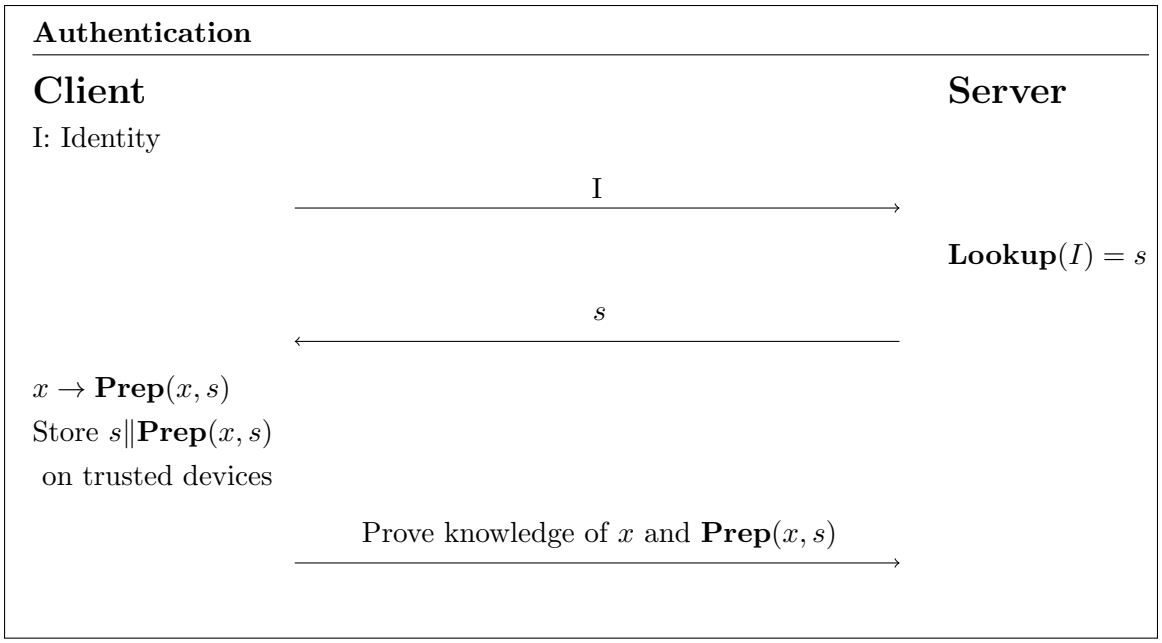
First we define the following algorithms:

- **Prep**(x, s): The input to this algorithm is a password x, and a random salt s, it will output $F(x\|s)$ with $F(.)$ a one-way function.
- **Gen**$(1^n, x)$: This algorithm takes a password x as input. It first generates a random salt s. Then, it generates a statement z for a proof of knowledge for witnesses x and $y = \textbf{Prep}(x, s)$. Note that z should be in an NP language, which is not in P. The algorithm outputs (z, s)
- **Proof**$((x, y), z)$: This algorithm creates a proof of knowledge Π for the statement z, given the witness (x, y).
- **Verify**(Π, z): This algorithm takes as input a proof Π for the statement z, and outputs 1 if the proof is valid, and 0 otherwise.

Using these algorithms we create the following protocol:

Registration

Client		**Server**
I: Identity x: password $s \xleftarrow{\$} \{0,1\}^n$: random salt z statement for proving knowledge of x and **Prep**(x, s)	$\xrightarrow{(I, s, z)}$	**Store**(I, s, z)



Note that the operations done by the client on registration can be grouped in the algorithm **Gen** as described before. The proof of knowledge in the last step is done by the algorithms **Proof** and **Verify**.

Given the above protocol we proof the following theorem:

Theorem 1 *Given a password x, and the algorithms as defined above. Then, the given protocol is a* zero-knowledge password proof that is hard against offline dictionary attacks*, if the algorithms have the following properties:*

- ***Efficiency:*** *The algorithms* $\mathbf{Proof}((x, y), z)$ *and* $\mathbf{Verify}(\Pi, z)$ *are efficient.*
- ***Hardness against offline dictionary attacks:*** $\mathbf{Prep}(.,.)$ *is a sequential memory-hard function.*
- ***Hiding:*** $\mathbf{Prep}(.,.)$ *is a one-way function*
- ***Zero-knowledge:*** $(\Pi = \mathbf{Proof}((x, y), z), \mathbf{Verify}(\Pi, z))$ *is a zero-knowledge proof system.*

Proof.

We will prove the different properties of a ZKPP that is hard against offline dictionary attacks:

Password Proof First, note that by correctness and soundness of the proof of knowledge, we derive correctness and soundness of our protocol. The efficiency property of the algorithms **Proof** and **Verify** provide the efficiency of the authentication process, therefore, we can conclude that this protocol is an efficient password proof.

Hard against eavesdropper For every non-uniform PPT adversary S^*, we construct a PPT simulator $\mathbb{S}'$ in the following way:

Input: $(z, (I, s))$, where z is the statement of the proof of knowledge and (I, s) is auxiliary input, as these values are public values.

(1) Output I the auxiliary information

(2) Receive (s', z') from the adversary, check if $z' = z$ and $s' = s$, if not output $\perp$

(3) Use the simulator $\mathbb{S}$ for the proof of knowledge.

First, note that the simulator $\mathbb{S}$ in step (3) exists, because of the zero-knowledge property of the proof of knowledge. Second, because the input to the simulator should be valid, the first round of the protocol is identical with the real protocol. Therefore, by the indistinguishability of the simulator $\mathbb{S}$ with the real proof of knowledge, we can conclude that $\mathbb{S}'$ is also indistinguishable from the real protocol.

Hard against server compromise The information stored on the server side is (I, s, z), given the fact that z is an element of an NP language that is not in P, we can assume that the best approach to find x is a brute-force attack (Note that I and s do not contain any information about x). The best way to do this is to create pairs $(x, \mathbf{Prep}(x, s))$ and check if $z = g^x h^{\mathbf{Prep}(x,s)}$. Given a dictionary $|D|$ for possible passwords, an adversary is expected to find the password after $\frac{|D|}{2}$ tries. Because the function $\mathbf{Prep}(x, s)$ is a sequential memory hard function, this will define the running time of the whole protocol. Therefore, we can say that the running time of $\mathbf{Prep}(x, s) \approx T(|x|)$, where $T(.)$ is the running time of the protocol. Hence, the expected running time of the adversary is greater than or equal to $\frac{|D|}{2}T(|x|)$.

Hard against client compromise The only information that is possibly stored on the client side is the salt s and the function $\mathbf{Prep}(x, s)$. Again, brute force is the only way given the one-wayness of $\mathbf{Prep}(.,.)$. Therefore, similar to the previous step, we can say that the running time of $\mathbf{Prep}(x, s) \approx T(|x|)$, where $T(.)$ is the running time of the protocol. Hence, the expected running time of the adversary is greater or equal to $\frac{|D|}{2}T(|x|)$.

■

6 Instantiation

In our instantiation of the proposed protocol above, we use *scrypt* as the sequential memory-hard function. This is the most well-known and has a lot of very good implementations in many different programming languages that are ready to use. Therefore, this seemed the most appropriate choice.

For the proof of knowledge we use the zero-knowledge password proof of Schnorr [Sch91] combined with the Fiat-Shamir heuristic [FS87] to make it non-interactive and decrease the number of rounds in our protocol. We chose this proof system because of its simplicity, which is in particular important on the client side, where one can expect less computational power.

The whole construction has been described in the following diagrams:

Registration

Client		Server
I: Identity		
x: password		
$s \xleftarrow{\$} \{0,1\}^n$: random salt		
$y = scrypt(x\|\|s)$		
$z = g^x g^{scrypt(x\|\|s)}$		
	(I, s, z) $\longrightarrow$	**Store**(I, s, z)

Authentication

Client		Server
I: Identity		
	I $\longrightarrow$	
		Lookup$(I) = s$
	s $\longleftarrow$	
$x \rightarrow scrypt(x\|\|s)$		
Store $s\|\|scrypt(x\|\|s)$ on trusted devices		
$z = g^x g^{scrypt(x\|\|s)}$		
	$PoK\{(x, scrypt(x\|\|s)) : z = g^x g^{scrypt(x\|\|s)}\}$ $\longrightarrow$	

6.1 Implementation

We provided a library that implements our protocol. It exists of two parts, a client side written in JavaScript [VDP16c] and a server side written in Java [VDP16a]. We provided an example web application that shows how to implement an authentication system using our library [VDP16b]. Note that we only showed how to do a simple authentication, we did not show how to do proper session management afterwards, nor did we put general protections against online dictionary attacks such as rate limiting or limit the number of tries. We also did not provide a proper password policy. When using our library, we still advice to use extra protections against online dictionary

attacks, as well as a proper password policy, which can enlarge a possible offline dictionary significantly. Best practices are well known and documentation on these issues can be found easily online.

It is also important that all communication between the client and server happens over a secure connection, using TLS for example. The information that can get leaked by the protocol on the communication channel is not necessarily enough for an eavesdropper to learn the password, but it is information that a system better keeps secure. Also, during registration the statement is sent from client to server, given this statement an eavesdropper can start performing an offline dictionary attack. Although, this attack will not be efficient as proved above, it is not impossible and should definitely be avoided. Luckily, the tools to protect the channel are easily available and implementable by just setting up a server using TLS.

6.1.1 Client side

The implementation of the client side has been done in JavaScript, as we want to focus first and foremost on web applications. Later, it would be beneficial to provide library implementations in other languages such as Swift and Java to support mobile applications too.

The JavaScript implementation contains two functions that can be called, namely `register` and `login`. The first two functions are asynchronous and take a callback function as their last parameter.

Calling `register` will perform the client side actions that need to be taken to do the registration part of the protocol as described above. It takes as input a username, password, and a boolean to denote if the user is currently on a trusted device. The output of this function, passed on through the callback function is an object containing all information needed on the server side. This object can be send to the back end using a preferred implementation, in our example we use a REST-call containing a JSON-object in the body.

`login` takes as input a username, password, salt, and a boolean to denote if the user is currently on a trusted device. Note that the salt needs to be retrieved from the server. In our example application we implemented a separate call to receive this information. The login function will return an object that contains all the necessary information to prove knowledge of the password and its hashed version. This object needs to be send to the server for verification. Again, in our example app this is done by a REST-call which contains the object that was returned by the library.

Note that these functions will take care of the cached version of the hash function by themselves depending on the boolean resembling if it runs on a trusted device or not. The only addition that one must make when using the library is the actual communication with the server. Our implementation is optimized to make use of JSON transcripts that can be send between client and server. We think the implementation is very intuitive and easy to use, which was one of the important design goals.

6.1.2 Server side

We wrote the server side using Java, given it is a very reliable language, and very common in back end systems. Eventually, it would be nice if our library would also get implemented in other common languages such as C type languages, and JavaScript.

The library itself only contains one API-call, named `login`. it takes as input a `User` object and a `Proof` object.

The `User` object contains the user's identification, the statement associated with its password, and the salt that was used to compute the hash-function. Typically, this information should be stored in a database in the back end and would be referred to based on the user ID.

The `Proof` object, contains a commitment, an actual proof (sometimes called opening of commitment), and a timestamp to avoid reusing the same proof transcript multiple times.

The `login` method will do a verification of the proof that was given as a parameter, according to the statement that was passed along through the `User` object.

As shown in our example app, typically on registration the user information gets stored in a database, which is mongoDB in our case. The server side should provide a way to give out the salt that was stored for the user. In our example, there is a REST-call that responds with the salt, given the user's identification. Next, a login-call was provided, taking in the necessary proof values, using this and the information from the database, the `login` call of the library is invoked. When verification is correct, it will respond with a 200 (OK) HTTP-status, and a 401 (Unauthorized) HTTP-status otherwise.

6.2 Performance

Performance testing was conducted on the implementation of the proposed protocol to establish a baseline performance metrics. Tests were performed using Jasmine [Lab16] which is a behavior-driven development framework for testing JavaScript code.

Test Environment

All the tests were performed on the same base system for consistency with no other user process running in background. Tests were repeated and results obtained are averaged over 10 iterations.

Since the proposed protocol is implemented in JavaScript, there is an inherent dependency on the browser used. Hence, benchmark tests were performed on Mozilla Firefox, Google Chrome and Apple Safari browsers to establish baseline metrics on test environment. For the purpose of this test, JetStream a JavaScript benchmarking suite was used from [Jet16].

System	MacBook Pro (Retina, 15-inch, Mid 2015)
Operating System	macOS Sierra Version 10.12.1
Processor	2.5 GHz quad-core Intel Core i7
Memory	16 GB 1600 MHz DDR3
Graphics	AMD Radeon R9 M370X 2048 MB Intel Iris Pro 1536 MB
Browser	Google Chrome Version 55.0.2883.75 (64-bit) Mozilla Firefox Version 50.0.1 Apple Safari Version 10.0.1

Table 1: Specifications of test environment

Figure 1: Chrome performance on test environment.

JetStream is a JavaScript benchmark suite focused on the most advanced web applications.
For more information, read the in-depth analysis. Bigger scores are better.

Score
169.33
± 13.466

Test Again

Benchmark	Average Score	Benchmark	Average Score	Benchmark	Average Score
Latency	**85.998 ± 1.4599**	regex-dna	83.54 ± 5.019	gcc-loops.cpp	643.8 ± 60.74
3d-cube	56.48 ± 4.342	splay-latency	192.8 ± 66.93	hash-map	404.1 ± 428.4
3d-raytrace	67.08 ± 7.444	tagcloud	73.97 ± 8.970	mandreel	288.7 ± 8.826
base64	54.09 ± 2.750	typescript	67.24 ± 6.714	n-body.c	335.7 ± 21.06
cdjs	156.8 ± 7.581	**Throughput**	**285.92 ± 36.497**	navier-stokes	231.6 ± 11.90
code-first-load	97.99 ± 5.753	bigfib.cpp	240.6 ± 67.92	pdfjs	107.0 ± 22.98
code-multi-load	86.21 ± 1.584	box2d	311.0 ± 4.138	proto-raytracer	443.8 ± 23.86
crypto-aes	56.63 ± 5.441	container.cpp	322.4 ± 62.57	quicksort.c	240.8 ± 11.66
crypto-md5	90.92 ± 3.420	crypto	166.6 ± 5.850	regexp-2010	254.7 ± 35.47
crypto-sha1	62.37 ± 11.66	delta-blue	449.2 ± 38.82	richards	224.3 ± 17.41
date-format-tofte	74.57 ± 5.327	dry.c	386.1 ± 27.36	splay	196.9 ± 22.88
date-format-xparb	63.91 ± 5.008	earley-boyer	123.4 ± 17.87	towers.c	455.2 ± 29.21
mandreel-latency	195.0 ± 6.277	float-mm.c	429.0 ± 46.51	zlib	307.0 ± 16.11
n-body	120.2 ± 15.40	gbemu	304.0 ± 57.37	**Geometric Mean**	**169.33 ± 13.466**

Figure 2: Firefox performance on test environment.

JetStream is a JavaScript benchmark suite focused on the most advanced web applications.
For more information, read the in-depth analysis. Bigger scores are better.

Score
216.41
± 5.4888

Test Again

Benchmark	Average Score	Benchmark	Average Score	Benchmark	Average Score
Latency	**148.56 ± 4.7539**	regex-dna	141.1 ± 9.697	gcc-loops.cpp	437.0 ± 16.30
3d-cube	160.3 ± 38.90	splay-latency	183.3 ± 4.304	hash-map	345.7 ± 75.65
3d-raytrace	175.4 ± 50.63	tagcloud	170.0 ± 14.94	mandreel	336.1 ± 51.48
base64	90.99 ± 3.718	typescript	183.7 ± 11.60	n-body.c	272.2 ± 6.773
cdjs	181.8 ± 14.61	**Throughput**	**289.44 ± 13.747**	navier-stokes	176.6 ± 10.44
code-first-load	127.6 ± 55.32	bigfib.cpp	359.9 ± 42.95	pdfjs	203.2 ± 27.81
code-multi-load	113.8 ± 28.40	box2d	280.4 ± 31.66	proto-raytracer	333.2 ± 73.98
crypto-aes	165.6 ± 44.70	container.cpp	295.8 ± 8.865	quicksort.c	289.8 ± 13.54
crypto-md5	142.0 ± 18.64	crypto	217.3 ± 4.922	regexp-2010	313.0 ± 14.15
crypto-sha1	91.14 ± 5.029	delta-blue	324.4 ± 17.20	richards	257.6 ± 47.71
date-format-tofte	140.7 ± 25.71	dry.c	362.7 ± 47.42	splay	262.9 ± 9.883
date-format-xparb	187.0 ± 13.56	earley-boyer	197.7 ± 7.466	towers.c	265.3 ± 14.76
mandreel-latency	208.1 ± 71.02	float-mm.c	333.4 ± 14.87	zlib	244.1 ± 10.83
n-body	135.4 ± 8.889	gbemu	436.2 ± 36.29	**Geometric Mean**	**216.41 ± 5.4888**

Figure 3: Safari performance on test environment.

Library

The library performs multiple operations and computations. We have tested the speed of the library for following operations: registration, login, computing random salt, compute hash, cache hash, create statement, translate string to number and create

proof. Since Scrypt is the only time consuming operation, we have grouped and tested all non-scrypt operations and scrypt operations seperately.

Following are the time in milliseconds for non-scrypt operations. It should be noted here that all three browsers perform quite well on these operations. Although, Safari seems a little slower, this difference is still rather negligible. This however does not hold true for the scrypt operations, as we will discuss further on.

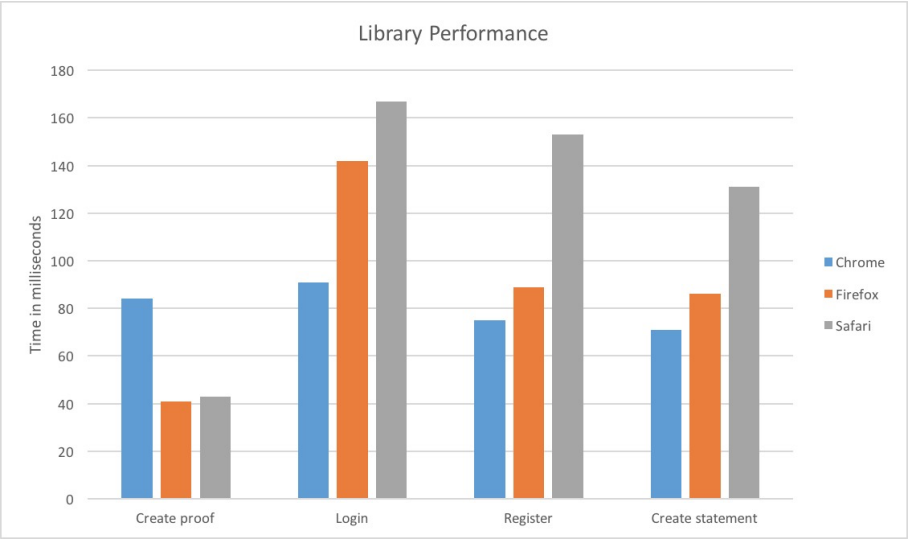


Figure 4: Library Performance

Scrypt

Scrypt accepts six parameters: passphrase, salt, CPU/memory cost, Block size parameter, parallelization parameter and output length. The suggested protocol assumes a fixed parallelization parameter of '1' and output length of '64'. For any given passphrase, salt, parallelization parameter and output length, the time taken to compute the output key varies largely based on CPU/memory cost and block size parameter chosen. Hence, we tested for a range of CPU/memory cost and block size parameter. CPU/memory cost was tested for N = 1024, 2048, 4096, 8192, 16384, 32768, 65536 and 131072 and block size r = 4, 8, 16 and 32. It was observed that Firefox and Safari ran out of memory and failed to complete computation when N=131072 and block size = 32. When tested on Google Chrome, Scrypt ran for the longest time compared to all other browsers.

Following are the time in milliseconds when Scrypt was tested on Google Chrome and averaged over 10 iterations. Notice how the computation time exponentially increases with increase in CPU/memory cost and block size.

CHROME	Block Size: 4	Block Size: 8	Block Size: 16	Block Size: 32
cpuCost:1024	242	347	724	1390
cpuCost:2048	436	682	1280	2480
cpuCost:4096	642	1267	2531	6417
cpuCost:8192	2313	4119	5564	10082
cpuCost:16384	2505	4910	9923	19463
cpuCost:32768	4961	9759	19326	38902
cpuCost:65536	9917	19556	38561	76994
cpuCost:131072	19594	38591	77271	153820

Figure 5: Scrypt performance on Google Chrome.

Following are the time in milliseconds when Scrypt was tested on Mozilla Firefox and averaged over 10 iterations. During the tests, Firefox prompted "Out of memory" when CPU Cost was 131072 and Block size was 32.

FIREFOX	Block Size: 4	Block Size: 8	Block Size: 16	Block Size: 32
cpuCost:1024	71	100	162	278
cpuCost:2048	91	146	247	465
cpuCost:4096	134	230	421	808
cpuCost:8192	230	416	767	1462
cpuCost:16384	408	780	1456	2819
cpuCost:32768	913	1610	2985	5764
cpuCost:65536	1583	3130	5872	14956
cpuCost:131072	3095	5776	12479	N/A

Figure 6: Scrypt performance on Mozilla Firefox.

Following are the time in milliseconds when Scrypt was tested on Safari and averaged over 10 iterations. Just like in Firefox, Safari was not able to complete its computation when CPU Cost was 131072 and Block size was 32.

SAFARI	Block Size: 4	Block Size: 8	Block Size: 16	Block Size: 32
cpuCost:1024	93	140	237	392
cpuCost:2048	120	200	718	795
cpuCost:4096	307	336	673	1450
cpuCost:8192	327	624	1436	2763
cpuCost:16384	649	1329	3007	5565
cpuCost:32768	1272	2960	6570	11918
cpuCost:65536	4866	8532	24969	22125
cpuCost:131072	8751	11805	27023	N/A

Figure 7: Scrypt performance on Safari.

Following are the time in milliseconds when Scrypt was tested in a C implementation and averaged over 10 iterations. It is evident from the results below that the C implementation is considerably faster thereby making it extremely important to choose parameters appropriately.

C	Block Size: 4	Block Size: 8	Block Size: 16	Block Size: 32
cpuCost:1024	5.98	10.72	20.73	40.22
cpuCost:2048	10.68	20.95	42.11	83.66
cpuCost:4096	21.32	41.17	84.39	171.14
cpuCost:8192	43.67	85.62	169.4	337.4
cpuCost:16384	87.98	171.25	336.65	654.31
cpuCost:32768	171.85	334.92	656.92	1303.06
cpuCost:65536	342.93	675.02	1323.68	2639.98
cpuCost:131072	685.38	1344.62	2637.68	5182.82

Figure 8: Scrypt performance for C implementation.

To get a clear idea about how much time each browser and C takes to finish computation, we have plotted the following graphs for each of the four block sizes. It is evident that scrypt takes the longest time to finish computation on chrome and fastest on C (among browsers Firefox was the fastest).

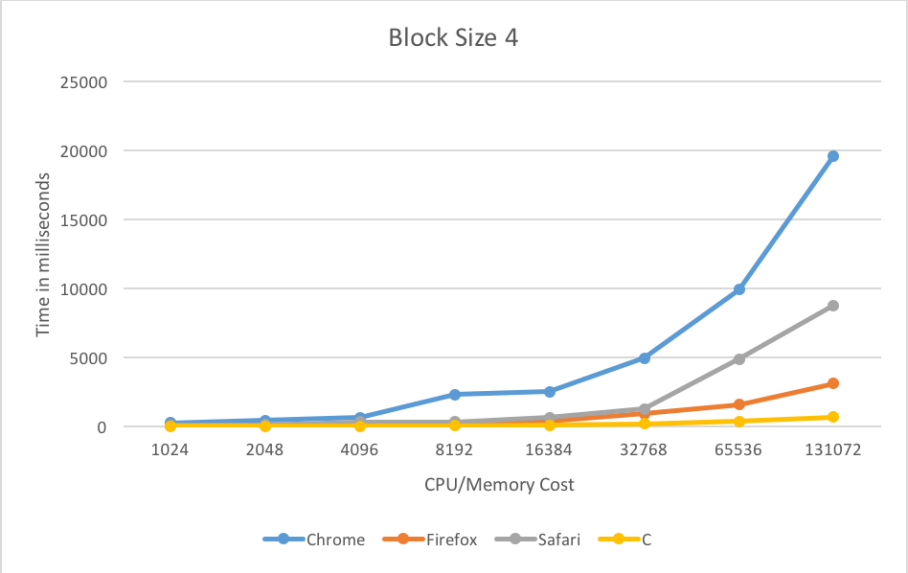


Figure 9: BlockSize 4

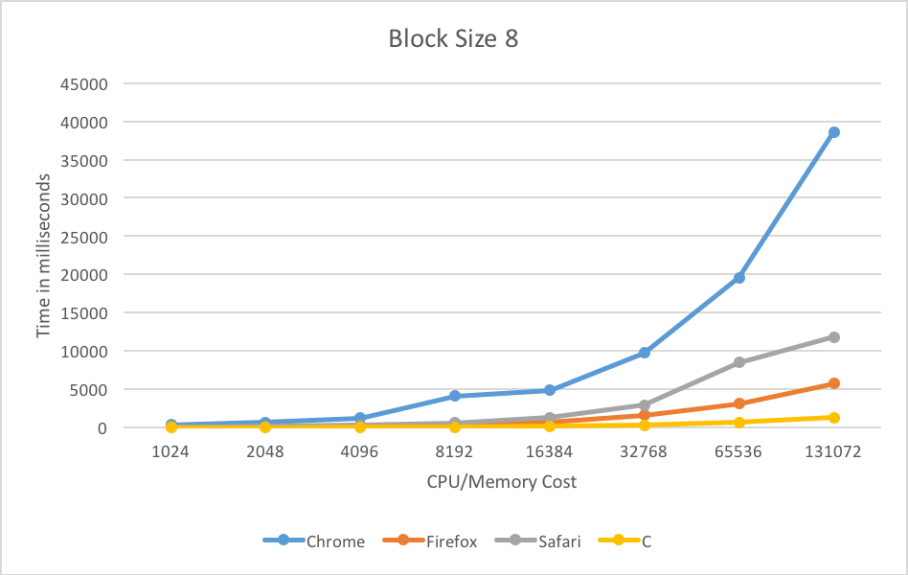


Figure 10: BlockSize 8

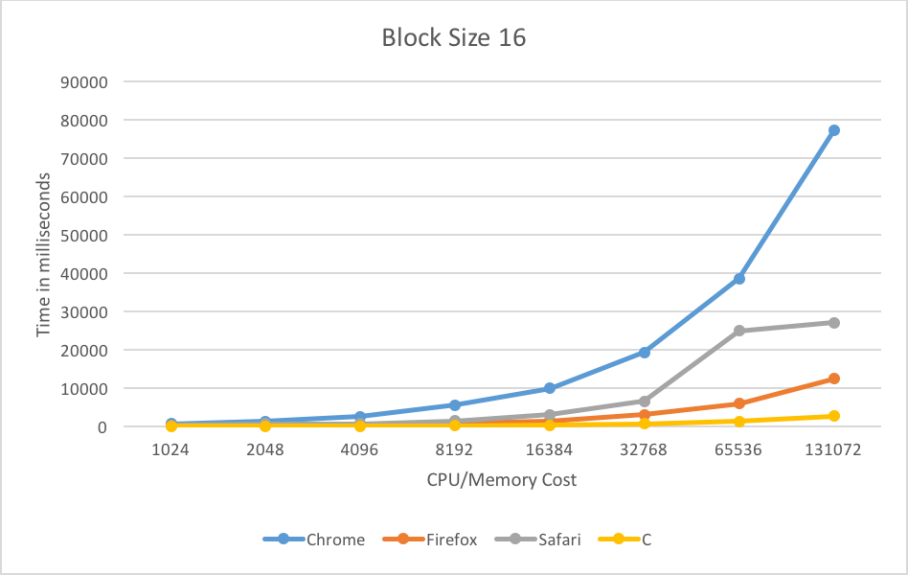


Figure 11: BlockSize 16

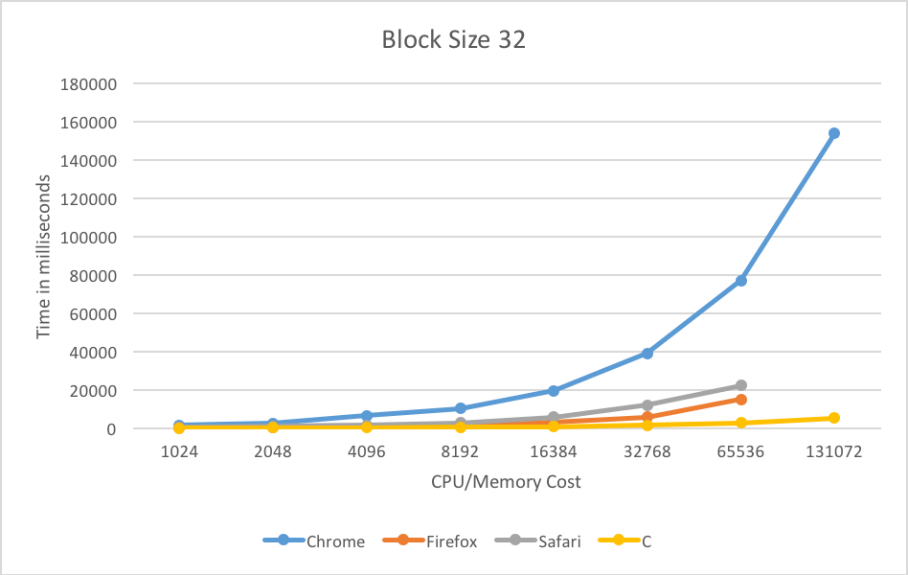


Figure 12: BlockSize 32

Performance test results

The performance tests of the implementation of the proposed protocol can be used as evidence to show that the implementation is efficient except for Scrypt operations which, being a memory hard problem, makes it extremely hard and discourages brute forcing of passwords. The test environment was chosen with specifications which made it an average computer thereby providing baseline performance metrics for the implementation.

Overall, the implementation is extremely efficient but the choice of CPU/memory cost and block size will determine practicality and security against brute force attacks. The choice of parameters depends heavily on the developer. As choosing very high CPU/memory cost and block size would mean high amounts of wait time for clients to register/login thereby making it not practical and a poor user experience. But on the otherhand, in the interest of usuability developers should not choose low values for CPU/memory cost and block size as that will severly compromise the security against offline dictionary attacks. Keeping these two cases in mind, we set the default CPU cost to $N = 16384$ and the block size parameter to $r = 16$.

7 Conclusion

In this paper, we have proposed a novel authentication protocol that prevents offline dictionary attacks through the combined use of a non-interactive Zero Knowledge Password Proof and a sequentially memory hard one-way function. Our protocol implements Schnorr's Zero Knowledge Password Proof combined with the Fiat-Shamir Heuristic for the Proof of Knowledge and scrypt for the sequentially memory hard function. We have proven the correctness, soundness and security of our protocol and

have proven our protocol hard against eavesdroppers, server compromise as well as client compromise. We have also proven our protocol to be efficient.

We have also implemented our protocol in the form of a simple and lightweight library that includes a client side written in JavaScript and a server side written in Java. Our library is easy-to-use for developers and can be used in web deployments with little effort. To show this, we have created a example web application that uses our library to implement our authentication system.

Our performance test results show that our implementation is extremely efficient in terms of time and memory on several platforms. We observe scrypt to be the only time and memory consuming factor in our implementation as intended.

8 Future work

Features such as an inbuilt strong password policy check, two-factor authentication and session management can be introduced into our implementation at a later point to make it easier for developers to quickly create and deploy secure authentication systems. The library implementation can also be ported to other popular languages such as C, Swift or Python to make it available to a broader set of developers and users. Migration from legacy authentication systems to our system can also be improved upon to make it even easier to make the move. The performance of our system could possibly be further optimized in later versions with the appropriate parameters in place.

References

[AB15] Dmitry Khovratovich Alex Biryukov, Daniel Dinu. Argon2: the memory-hard function for password hashing and other applications. `https://password-hashing.net/argon2-specs.pdf`, 2015.

[ACP+16] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. Cryptology ePrint Archive, Report 2016/989, 2016. `http://eprint.iacr.org/2016/989`.

[AMV96] P. van Oorschot A. Menezes and S. Vanstone. *Identification and Entity Authentication*, pages 385–424. CRC Press,, 1996.

[BCKL08] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. *P-signatures and Noninteractive Anonymous Credentials*, pages 356–374. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[BDK15] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications. `https://password-hashing.net/argon2-specs.pdf`, 2015.

[BK15] Alex Biryukov and Dmitry Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. Cryptology ePrint Archive, Report 2015/946, 2015. `http://eprint.iacr.org/2015/946`.

[Bou00] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. pages 431–444. Springer Verlag, 2000.

[Bra97] Stefan Brands. Rapid demonstration of linear relations connected by boolean operators. In *EUROCRYPT 97*, pages 318–333. Springer Verlag, 1997.

[CCS08] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In *ASIACRYPT*, pages 234–252. Springer, 2008.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*, pages 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[CNS07] Jan Camenisch, Gregory Neven, and Abhi Shelat. Simulatable adaptive oblivious transfer. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, EUROCRYPT '07, pages 573–590, Berlin, Heidelberg, 2007. Springer-Verlag.

[CS97] Jan Camenisch and Markus Stadler. *Efficient group signature schemes for large groups*, pages 410–424. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Fei14] Ashley Feinberg. Sony kept thousands of passwords in a folder named "password". `http://gizmodo.com/sony-kept-thousands-of-passwords-in-a-document-marked-1666772286`, 2014.

[FFS88] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[FS87] Amos Fiat and Adi Shamir. *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*, pages 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[GQ88] L. C. Guillou and J.-J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*, pages 123–128, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[Gro06] Jens Groth. *Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures*, pages 444–459. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[GS07] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. Cryptology ePrint Archive, Report 2007/155, 2007. `http://eprint.iacr.org/2007/155`.

[Has16] Hashcat. Hashcat: Advanced password recovery. `https://hashcat.net`, 2016.

[Hun12] Troy Hunt. Our password hashing has no clothes. https://www.troyhunt.com/our-password-hashing-has-no-clothes/, 2012.

[Jab96] David P. Jablon. Strong password-only authenticated key exchange. http://grouper.ieee.org/groups/1363/passwdPK/contributions/jablon.pdf, 1996.

[Jet16] JetStream. Jetstream: a javascript benchmark suite focused on the most advanced web applications, 2016. http://browserbench.org/JetStream/.

[Kam12] Poul-Henning Kamp. Linkedin password leak: Salt their hide. http://queue.acm.org/detail.cfm?id=2254400, 2012.

[Lab16] Pivotal Labs. Jasmine: Dom-less simple javascript testing framework. https://jasmine.github.io, 2016.

[NP99] David Mazires Niels Provos. A future-adaptable password scheme. https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf, 1999.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[NT94] B. C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sept 1994.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions. https://www.tarsnap.com/scrypt/scrypt.pdf, 2009.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[SMB16] Michael Merritt Steven M. Bellovin. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. https://www.cs.columbia.edu/~smb/papers/aeke.pdf, 2016.

[VDP16a] Gijs Van Laer, Rono Dasgupta, and Aditya Patil. ZKAuth backend library. Bitbucket repository, 2016. https://bitbucket.org/zk_capstone/backend.

[VDP16b] Gijs Van Laer, Rono Dasgupta, and Aditya Patil. ZKAuth example application. Bitbucket repository, 2016. https://bitbucket.org/zk_capstone/example-app.

[VDP16c] Gijs Van Laer, Rono Dasgupta, and Aditya Patil. ZKAuth frontend library. Bitbucket repository, 2016. https://bitbucket.org/zk_capstone/frontend.

[Wu97] Thomas Wu. The secure remote password protocol. http://srp.stanford.edu/ndss.html, 1997.