

DRAFT Guess Limited Password Protected Secret Sharing Proposal

Rolfe Schmidt

May 5, 2023

1 Overview

This is a protocol for *guess-limited password-based secure value recovery*. It allows clients to interact with servers to securely reconstruct a secret using a password while providing protection against both offline and online dictionary attacks - even in the event of server compromise. It protects against online dictionary attacks through guess limiting: after a configured number of failed reconstruction attempts, the secure value becomes unrecoverable.

1.1 Outline

This protocol is a variation of the PPSS protocol of [JKKX16] implemented with a *usage limited* version of the standards track 2HashDH OPRF of [JKK14] as specified in [AD] that can be used safely with smaller curves like Ristretto255.

After covering notation in section 2 we present an augmentation of the standards track OPRF of [AD] in section 3 that has servers generate per-client OPRF keys, enforces strict usage limits on these keys, and allows clients to rotate their keys to avoid running into usage limits.

In section 4 we use this usage limited OPRF to construct a secure PPSS. This protocol is close to that of [JKKX16], but does not mandate storage of masked shares on servers and eliminates the share commitment storage on servers. We discuss ways to obtain robustness in section 5.

Importantly, we observe that if the underlying OPRF limits clients to `max_uses` per key, then against a (t, N) threshold scheme an attacker will be limited to $\lfloor \frac{N}{t+1} \text{max_uses} \rfloor$ password guess attempts before the secret becomes unrecoverable. Thus our PPSS is *guess limited*. We also show how keys can be deleted from the server to offer a form of forward security in case of server compromise.

2 Notation

Algebraic objects. This protocol will use a prime order cyclic group, \mathbb{G} among those specified in [AD]. Since key use will be limited, we can take \mathbb{G} to be Ristretto255. We denote the order of \mathbb{G} by q and thus denote the set of scalars for \mathbb{G} by \mathbb{Z}_q . Group elements will be denoted by capital Latin letters, e.g. A, B, C, \dots . Scalars will be denoted by lower case Latin letters, e.g. a, b, c, \dots . G denotes a public generator of \mathbb{G} . Scalar multiplication will usually be denoted without a symbol - aG - but in places the infix operator $*$ will be used for clarity, as in $\text{sk}_{\text{oprf}} * G$.

Secret sharing will be performed using polynomials over a finite field, \mathbb{F} , that is not related to the group \mathbb{G} .

Domain separation. Throughout the protocol we will use `context` to denote a domain separation prefix unique to the application performing the protocol.

Server and client state. Each server will have state information captured in the variable `server_state`. The public part of this state is available in the variable `server`.

Similarly, each client will have persistent information captured in the variable `client_state`, and the public part of this state will be accessible through the variable `client`.

Function parameters. We will use a number of functions associated with \mathbb{G} and \mathbb{F} which we consider as protocol parameters. In an instantiation of the protocol the parameters will be identified in the `contextstring`. The function parameters are:

- All parameters for the OPRF specified in [AD]
- $\text{HashToField} : \{0, 1\}^* \rightarrow \mathbb{F}$

3 The OPRF

The PPSS protocol relies on the verifiable 2HashDH OPRF of [JKK14] as specified in the IRTF draft standard [AD]. We describe the protocol using the OPRF mode of the standard but in section 5 note situations where follow up calls to the VOPRF mode can be used to ensure robustness.

The IRTF standard specifies the following functions:

1. $(\text{blind}, \text{blindedElement}) \leftarrow \text{Blind}(\text{oprf_input})$: Used by a client to prepare the OPRF input to be sent to the server.
2. $\text{evaluatedElement} \leftarrow \text{BlindEvaluate}(\text{sk}, \text{blindedElement})$: Executed by the server to evaluate the OPRF parameterized by the server-secret key sk on blindedElement to compute a blinded output.
3. $v \leftarrow \text{Finalize}(\text{oprf_input}, \text{blind}, \text{evaluatedElement}, \text{blindedElement})$: Takes the values returned by BlindEvaluate along with original input, and values returned by Blind to compute the final PRF value, v .

3.1 Usage Limited Evaluation

The server for our protocol adds to these functions in two ways: it uses a random per-client OPRF key stored in the dictionary `server_state.oprf_keys` and it enforces a usage limit. Each OPRF key can only be used a fixed number of times. Setting and rotation of these keys is discussed in 3.2. This is done with the function `BlindEvaluateForClient`:

`BlindEvaluateForClient(server_state, client_id, blindedElement)`

```

1 : usage_count ← server_state.usage_count[client_id]
2 : server_state.usage_count[client_id] ← usage_count + 1
3 : if usage_count ≥ server_state.max_uses :
4 :   return ⊥
5 : (sk, pk) ← server_state.oprf_keys[client_id] // The client MAY obtain the pk corresponding to sk at registration
6 : return BlindEvaluate(sk, blindedElement)
```

3.2 OPRF Key Creation and Versioning

As noted in the previous section, OPRF keys are created per-client and each key is strictly limited to a fixed number of uses. The usage limitation has two useful purposes. First, since the security of the OPRF is based on the one-more Diffie Hellman assumption, the security of a key used for Q queries is reduced by $\log(Q)/2$ bits (see, e.g., 7.2.3 of the IRTF draft). So, for example, by limiting key usage to no more than 16 queries we only lose 2 bits of security and can safely use a group like Ristretto255. Second, this limit enforcement will be the basis of the guess limiting in the PPSS described in section 4.

Clients in our PPSS will need to reconstruct their secret an unlimited number of times, though. To do this, upon successful reconstruction the client will create a new version of their OPRF key. This new version will be constructed with the function `ServerCreateOPRFVersion`, which creates a new key pair, stores it indexed by the client's identifier, clears the usage count, and evaluates the OPRF with the new key on a blinded element:

ServerCreateOPRFVersion(server_state, client_id, blindedElement)

```

1 : sk  $\leftarrow$   $\mathbb{Z}_q$ 
2 : pk  $\leftarrow$   $kG$ 
3 : server_state.oprf_keys[client_id]  $\leftarrow$  (sk, pk)
4 : server_state.usage_count[client_id]  $\leftarrow$  0
5 : evaluatedElement  $\leftarrow$  BlindEvaluate(sk, blindedElement)
6 : return (evaluatedElement, pk) // Return the public key in case NIZK proof is needed later

```

3.3 A Note About POPRF Mode

It is tempting to use the POPRF mode introduced in [TCR⁺21] rather than generating client specific keys. If usage limitation were not a requirement this would have a clear advantage - the server state would be no more than one secret OPRF scalar. However, once we introduce the need for key usage limits and key rotation this advantage disappears. Usage limitation requires storage of per-client state. Key rotation requires the use of a nonce or a counter for each client, effectively requiring the same storage as the proposed per-client key solution.

4 A Guess Limited PPSS from the OPRF

With these primitives in place we define the PPSS scheme with the functions PPSSStore and PPSSRecover. The idea is simple. To create a (t, N) threshold PPSS to store a secret s with N servers we

1. Create a degree t polynomial $f \in \mathbb{F}[x]$ with s as the leading coefficient, all other coefficients random.
2. Create a share for each server: $s_i = f(x_i)$ where $x_i = \text{HashToField}(\text{server}_i.\text{id})$
3. Use the OPRF values to mask the shares: $m_i = s_i + \text{server}_i.\text{OPRF}(\text{client_id}, \text{pwd})$.
4. Store the values m_i somewhere reliable, but confidentiality is not important.
5. To reconstruct, simply call $(t + 1)$ or more servers to get their OPRF values and use these to unmask the shares: $s_i = m_i - \text{server}_i.\text{OPRF}(\text{client_id}, \text{pwd})$.
6. These shares can now be used to reconstruct the secret s .
7. Upon successful reconstruction the client can create new key versions on all servers, refresh their guess counts, and create new masked shares. All of this can be done without changing the password or master secret.

In the following **servers** is a set of N **server** objects, **e** is a dictionary that will store masked shares of a secret s , and **pks** is a dictionary that stores server OPRF public keys.

PPSSStore(client_state, servers, t, pwd, s)

```

1 :  $r \| K \leftarrow H(\text{context} \| \text{"keygen"}, s)$ 
2 :  $\forall i \in [0, t-1] : f_i \leftarrow \mathbb{F}$ 
3 :  $f_t \leftarrow \text{EncodeToField}(s)$ 
4 : for server  $\in$  servers :
5 :   oprf_input  $\leftarrow \text{context} \| \text{server.id} \| \text{pwd}$ 
6 :    $x \leftarrow \text{HashToField}(\text{server.id})$ 
7 :    $y \leftarrow \sum_{i=0}^t f_i x^i$ 
8 :   (blind, blindedElement)  $\leftarrow \text{Blind}(\text{oprf\_input})$ 
9 :   (evaluatedElement, pk)  $\leftarrow \text{server.ServerCreateOPRFVersion}(\text{client\_state.id}, \text{blindedElement})$ 
10 :   $\rho \leftarrow \text{Finalize}(\text{oprf\_input}, \text{blind}, \text{evaluatedElement}, \text{blindedElement})$ 
11 :  // e and pks should be stored somewhere reliable, but confidentiality is not needed
12 :   $s[x] \leftarrow y$ 
13 :  client_state.e[x]  $\leftarrow y + \rho$ 
14 :  client_state.pks[server.id]  $\leftarrow \text{pk}$ 
15 :  client_state.C  $\leftarrow H(\text{context} \| \text{"commitment"}, \text{pwd}, \text{client\_state.e}, s, r)$ 
16 : return K

```

PPSSRecover(client_state, servers, t, pwd)

```

1 : Choose  $\mathcal{R} \subset \text{servers}, |\mathcal{R}| > t$ 
2 : pairs  $\leftarrow \{\}$ 
3 : for server  $\in \mathcal{R}$  :
4 :   oprf_input  $\leftarrow \text{context} \| \text{server.id} \| \text{pwd}$ 
5 :    $x \leftarrow \text{HashToField}(\text{server.id})$ 
6 :   (blind, blindedElement)  $\leftarrow \text{Blind}(\text{oprf\_input})$ 
7 :   (evaluatedElement, pk)  $\leftarrow \text{server.BlindEvaluateForClient}(\text{client\_state.id}, \text{blindedElement})$ 
8 :    $r \leftarrow \text{Finalize}(\text{oprf\_input}, \text{blind}, \text{evaluatedElement}, \text{blindedElement})$ 
9 :    $y \leftarrow \text{client\_state.m}[x] - r$ 
10 :   $s[x] \leftarrow y$ 
11 :  pairs  $\leftarrow \text{pairs} \cup \{(x, y)\}$ 
12 :   $(f_t, \dots, f_0) \leftarrow \text{Interpolate}_{\mathbb{F}}(\text{pairs})$ 
13 :   $s \leftarrow f_t$ 
14 :   $r \| K \leftarrow H(\text{context} \| \text{"keygen"}, s)$ 
15 :   $C \leftarrow H(\text{context} \| \text{"commitment"}, \text{pwd}, \text{client\_state.e}, s, r)$ 
16 :  if  $C \neq \text{client\_state.C}$  :
17 :    return  $\perp$ 
18 :  else
19 :    PPSSStore(client_state, servers, t, pwd, f_t) // store the secret again to reset keys, counters, and shares
20 :  return K

```

4.1 Usage Limits on the OPRF lead to Guess Limits on the PPSS

Now we can see how the usage limit we enforce on the OPRF naturally creates a guess limit on the PPSS that provides protection against online dictionary attacks. Consider the scenario where a client has constructed a (t, N) -sharing scheme to protect a secret s with password pwd using PPSSStore. Now an attacker trying to guess the password and recover the secret faces the following fact: each password guess requires using $t + 1$ OPRF calls, and only max_uses are possible on each of the N servers. Thus the attacker has no more than $\lfloor \frac{N}{t+1} \text{max_uses} \rfloor$ guesses before the secret becomes unrecoverable.

4.2 Deleting Keys

A client can protect themselves from future server compromise by deleting keys from the server. This can be done by simply calling `ServerCreateOPRFVersion` with arbitrary `opr_f_input` and discarding the result. For the user to have confidence that the keys were in fact deleted - now and during each `PPSSStore` call - server functions can be executed in an attested, confidential TEE.

Additionally, in the case that a TEE based server is being retired it can produce an attested certificate of secret deletion. If a client has confidence that their secrets have, in fact, been deleted from a server then they know that their (t, N) threshold scheme has become a $(t, N - 1)$ scheme and they can safely add a new server.

5 Robustness

Unlike [JKKX16] we do not store the commitment, C , and the masked shares, \mathbf{e} , on each server. Instead we will have clients store `client_state`, which includes both these values, in a reliable place as suggested in [JKK14]. We then rely on subset testing or follow-up VOPRF calls to detect incorrect servers.

The sole reason the server `pk` values are stored by the client in the protocol above is to allow follow-up NIZK proof verification if the VOPRF mode is used for robustness. If only subset testing will be used (e.g. for small values of N) then these public keys do not need to be stored.

6 Acknowledgements

We would like to thank Mark Johnson for helping to develop an earlier version of this protocol and Trevor Perrin for important feedback and pointers to the literature. We also thank Emma Dautermann, Vivian Fang, and Raluca Ada Popa for discussion that led to significant design decisions and simplifications of this protocol.

References

- [AD] N. Sullivan C. A. Wood A. Davidson, A. Faz-Hernandez. Oblivious pseudo-random functions (oprfs) using prime-order groups. <https://www.ietf.org/id/draft-irtf-cfrg-voprf-21.html>.
- [JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. Cryptology ePrint Archive, Paper 2014/650, 2014. <https://eprint.iacr.org/2014/650>.
- [JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). Cryptology ePrint Archive, Paper 2016/144, 2016. <https://eprint.iacr.org/2016/144>.
- [TCR⁺21] Nirvan Tyagi, Sofia Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious prf, with applications. Cryptology ePrint Archive, Paper 2021/864, 2021. <https://eprint.iacr.org/2021/864>.