Review of RFC 9497: Oblivious Pseudorandom Functions Using Prime-Order Groups

Leonard Wilhelm Lorenz Gerk Srividya Subramanian

***Abstract*—The RFC 9497 documents a cryptographic protocol named Oblivious Pseudorandom Function (OPRF) that is used to compute a Pseudorandom Function (PRF) between two parties. During a run of the protocol, the client sends an input and the server processes it using a private key. The client receives the computed result, but never sees the key, while the server does not learn the input or output. This protocol is implemented using standard prime-order groups, such as groups over elliptic curves. The RFC also proposes an extension to the OPRF called a Verifiable OPRF (VOPRF) which allows a client to ensure that the server utilized a designated private key while carrying out the protocol. A VOPRF can be further extended to a partially oblivious form, known as a POPRF, which allows the client and server to include public inputs in the computation.**

I. Introduction

Password managers are commonly used today to store passwords that are otherwise hard to remember by users. Although this is convenient, they also pose a security risk if the password manager is compromised by an attacker. With the use of Oblivious Pseudorandom Functions (OPRFs), it is possible to create a password manager which effectively has no knowledge of the passwords by generating them on the fly. This was done by Shirvanian et al. [1] and is one of the multiple practical applications of OPRFs. Other applications are password-protected secret sharing schemes [2] and password-authenticated key exchanges [3].

OPRFs are an extension of Pseudorandom Functions (PRFs). A PRF is a function $F(k, i) = o$ that accepts a secret value k (later equal to the server's secret key) and an input value i and returns a pseudorandom value o. F is pseudorandom if given a fixed k its output is indistinguishable from the output of a true random function. In opposition to PRFs, OPRFs require interaction between two parties, a client and a server, in order to compute the output of the PRF. The protocol between the client and the server ensures that the client never learns k and the server never learns i or o.

When using a Verifiable OPRF (VOPRF), the server creates an asymmetric key pair. It announces the public key (`pkS`) to the client and uses the private key (`skS`) as k for the PRF. The client receives a zero-knowledge proof along with the response of the server which enables it to verify that the k used by the server for the PRF corresponds to the `pkS` without the server revealing k itself. VOPRFs have also been used in practice, for example, for privacy-preserving CAPTCHAs [4].

To extend a VOPRF to a Partially Oblivious PRF (POPRF), a public input p is added to the PRF input. As with a VOPRF the client can verify that the k used by the server corresponds to the `pkS` without the knowledge of k. If p is fixed, a POPRF is functionally equivalent to a VOPRF.

In the RFC 9497 [5] protocols, application considerations and security properties for OPRF, VOPRF and POPRF are specified with prime-order groups as their basis. The RFC was written by the Crypto Forum Research Group of the Internet Research Task Force and is not yet an IETF standard.

II. Background

In order to fully understand the content of the RFC, this section summarizes the mathematical background.

- **Prime Order Groups:** An OPRF uses prime order groups for its keys, computations, and zero-knowledge proofs. Such a group contains a prime number of elements, and an operation $+$ called addition. Adding an element k times to itself is referred to as scalar multiplication with k. One of the elements in the set is defined as the group generator `gen` and together with scalar multiplication, it can generate all elements of the group. In the RFC, the OPRFs are intended to be used with prime order groups instantiated over elliptic curves, so $+$ would be the point addition of two points of the curve.
- **Discrete Logarithm Problem:** The discrete logarithm on a prime order group is the inversion of scalar multiplication: given elements A and B, find the scalar k so that $k \cdot A = B$. The hardness of the (elliptic curve) discrete logarithm problem depends on the chosen group. With an appropriate prime order group, it is computationally infeasible to derive the `skS` from `pkS`.
- **Discrete Logarithm Equivalence Proof (DLEQ):** In VOPRF and POPRF, in order to provide verifiability of its computation, the server wants to prove to the client that it used the same $k =$ `skS` for the evaluation that was used to generate `pkS`. Naturally, this proof should not reveal k itself, thus a non-interactive, zero-knowledge proof for discrete logarithm equivalence (DLEQ) is used. Using this, the server can prove to the client that for two pairs of elements A, B and C, D, $(k \cdot A = B) \wedge (k \cdot C = D)$ without revealing k.

III. Related Work

OPRFs have been studied in various contexts, leading to many efficient and secure designs. The paper "SoK: Oblivious Pseudorandom Functions" [7] surveys existing OPRF

constructions, classifies them based on their cryptographic assumptions, efficiency, and security models, and discusses their use in applications like private set intersection and password-authenticated key exchange. Classically secure OPRF variants are primarily built on the (Gap) One-More Diffie-Hellman (OMDH) assumption. For example, the TOPPSS protocol [8] uses a threshold OPRF which is instantiated using a Diffie–Hellman-style function with an additively shared key.

In addition to these, there are also several post-quantum OPRF variants which are based on primitives like isogenies, lattices, symmetric cryptography and oblivious transfer for example, [9].

IV. Protocol

The RFC defines three protocol variants: OPRF, VOPRF and POPRF. All variants exchange two messages between the client and the server and perform an offline setup phase followed by an online phase. Fig. 1 shows the interfaces exposed by the ORPF protocol variant as an example.

Client :
SetupOPRFClient(`identifier`) $\rightarrow$ `context`
Blind(`input`) $\rightarrow$ `blind`, `blindedElement`
Finalize(`input`, `blind`, `evaluatedElement`) $\rightarrow$ `output`

Server :
SetupOPRFServer(`identifier`, `skS`) $\rightarrow$ `context`
BlindEvaluate(`skS`, `blindedElement`) $\rightarrow$ `evaluatedElement`

Fig. 1. ORPF protocol interfaces

During the offline phase, the server performs the key generation. The server key pair can be generated randomly or deterministically derived from a given seed value and some public information. If the former is chosen, the `skS` is a random scalar selected from $\{1, ..., p-1\}$, where p is the order of the chosen prime-order group. The public key is a group element and is calculated by scalar-multiplying the group's generator with the secret key.

In addition, the client and server set up their contexts for the protocol using **SetupOPRFClient** and **SetupOPRFServer**, respectively. For the VOPRF and POPRF variants, the function signature for the client must be extended by the `pkS`, which is required for verifying the zero-knowledge proof. The `identifier` describes the selected ciphersuite. Finally, the setup functions return a `context` object, which is implementation dependent and stores all relevant information for the online phase.

For the online phase, the RFC defines four functions (**Blind**, **BlindEvaluate**, **Finalize** and **Evaluate**) for each protocol mode. If an entity knows all private values of the client and the server, it can use the **Evaluate** function to compute the PRF result. However, we will not discuss this function in detail, as it voids all privacy and security guarantees that the protocol is supposed to provide. Additionally, all protocol variants also support a batch mode in which multiple PRF computations can be performed with a single request to the server. The following sections describe how the **Blind**, **BlindEvaluate** and **Finalize** functions work in non-batch mode during the online phase of each protocol variant.

A. OPRF

The online phase of the OPRF mode is shown in Fig. 2. It executes the following functions to compute the output of the OPRF.

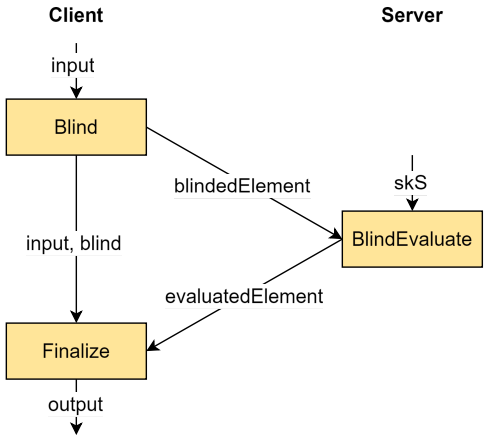

Fig. 2. Online Phase of OPRF Mode

- **Blind**: At the start of the protocol, the client blinds its `input` to ensure that the server does not learn it. The `blind` is a random scalar of the chosen prime order group. It is then scalar-multiplied with the `inputElement` that is derived from the `input` using a hash function. The `blind` is stored by the client and the `blindedElement` is sent to the server.

$$\texttt{blind} = \textbf{randomScalar}()$$
$$\texttt{inputElement} = \textbf{toGroupElement}(\texttt{input})$$
$$\texttt{blindedElement} = \texttt{blind} \cdot \texttt{inputElement}$$

- **BlindEvaluate**: The function takes the `skS` and the `blindedElement` as the input and returns an `evaluatedElement` to the client which is the scalar-product of `skS` with the `blindedElement` in the prime-order group.

$$\texttt{evaluatedElement} = \texttt{skS} \cdot \texttt{blindedElement}$$

- **Finalize**: The client unblinds the `evaluatedElement` by multiplying it with the inverse of the stored `blind`. The **Finalize** function returns the result of a hash function that takes a string that contains the private client `input` and the `unblindedElement` as its input.

$$\texttt{unblindedElement} = \textbf{scalarInverse}(\texttt{blind}) \cdot \texttt{evaluatedElement}$$

B. VOPRF

VOPRF differs from OPRF in its **BlindEvaluate** function. In addition to the computation of the evaluated element, it also generates a DLEQ proof to show that the secret key corresponding to its public key has been used in the evaluation. The client then only accepts the `evaluatedElement` if the proof is verifiable.

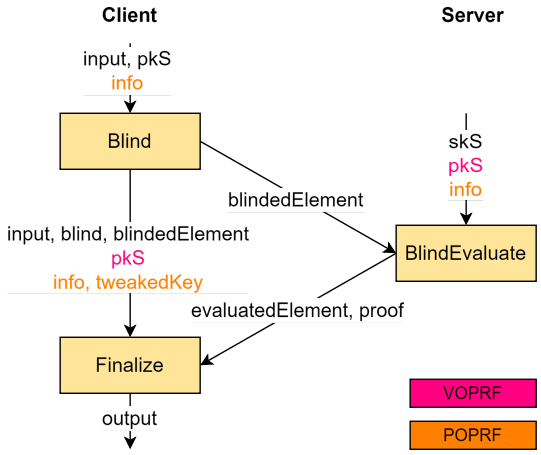

Fig. 3. Online Phase of VOPRF and POPRF Mode

In detail, during the evaluation, the server generates a DLEQ proof for the group generator, `pkS`, `blindedElement`, and `evaluatedElement`. First, after computing the `evaluatedElement`, the server pseudorandomly generates two group elements M and Z using `pkS`, `blindedElement`, `evaluatedElement` and additional context like the ciphersuite. This is done by generating a pseudorandom scalar di and then computing $M = di \cdot \texttt{blindedElement}$ and $Z = di \cdot \texttt{evaluatedElement}$. After that, the server chooses a private random value r and uses it to compute $t2 = r \cdot \texttt{gen}$ and $t3 = r \cdot M$. The proof that is then sent to the client consists of two scalars c and s. c is pseudorandomly generated from $B, M, Z, t2$ and $t3$ using a hash function, and s is the result of $s = r - c \cdot \texttt{skS}$.

To verify the proof, the client can recompute M and Z, as it has access to `pkS`, `blindedElement` and `evaluatedElement`. With M and Z, as well as the proof of the server, the client can recompute $t2$ and $t3$, even without knowledge of r using the following equation:

$$
\begin{aligned}
t2 &= (s \cdot \texttt{gen}) + (c \cdot \texttt{pkS}) \\
&= ((r - c \cdot \texttt{skS}) \cdot \texttt{gen}) + (c \cdot (\texttt{skS} \cdot \texttt{gen})) = r \cdot \texttt{gen} \\
t3 &= (s \cdot M) + (c \cdot Z) \\
&= ((r - c \cdot \texttt{skS}) \cdot M) + (c \cdot (di \cdot \texttt{evaluatedElement})) \\
&= ((r - c \cdot \texttt{skS}) \cdot M) + (c \cdot (di \cdot (\texttt{skS} \cdot \texttt{blindedElement}))) \\
&= ((r - c \cdot \texttt{skS}) \cdot M) + (c \cdot (\texttt{skS} \cdot M)) = r \cdot M
\end{aligned}
$$

Using these values, the client recomputes c using $B, M, Z, t2$ and $t3$ and compares it to the one sent by the server. If they match, the proof is verified successfully.

If the server tried to use a different key for the evaluation, the recomputed $t3$ would not match the one used to compute c so $c' \neq c$. In order to trick the client, the server would have to manipulate $t3$ and $t2$ so that the client obtains the same result. However, this is hard due to the preimage-resistance of the used hash function.

This addition provides verifiability that allows a client to detect if a malicious server outputs faulty results, thus ensuring that the `evaluatedElement` has been computed according to the protocol.

C. POPRF

Unlike the VOPRF protocol, POPRF uses a modified **Blind** function where the client also includes an `info` value. As shown in Fig. 3, this generates a `tweakedKey` which is stored locally by the client and used later in the **Finalize** function. The `tweakedKey` is derived by adding a value `T` to the server's public key. `T` is the scalar product between the group generator and another scalar value derived from a formatted version of `info`.

In the **BlindEvaluate** function on the server side, the `skS` is adjusted by adding a scalar derived from the formatted `info` to get a new value `t`. The `evaluatedElement` is calculated as the scalar inverse of `t` times the `blindedElement`. Unlike OPRF and VOPRF where the `evaluatedElement` is simply `skS` * `blindedElement`, this change is introduced to ensure that the same security properties still hold. The `tweakedKey` is then computed as the scalar product between the group generator and `t`. Therefore, unlike VOPRF the new values `t` and `tweakedKey` are used for proof generation instead of `skS` and `pkS`.

Correspondingly, the **Finalize** and **Evaluate** functions incorporate the additional public value. They use the `tweakedKey` to verify the proof or compute the PRF result, and include `info` in the final hashed output.

V. Application and Security Considerations

A. Application Considerations

The RFC defines several application constraints for the protocols such as input constraints, interface design, error handling, and public input usage in POPRF. Inputs, whether private or public, must not exceed $2^{16} - 1$ bytes, and longer inputs should be hashed to a fixed size, taking into account the input size limit of the hash function. While protocol functions internally operate on group elements and scalars, implementations can expose simpler, application-specific interfaces. Error handling is essential, as functions like **Finalize** and **BlindEvaluate** may fail due to invalid inputs, requiring explicit error management for verification failures, de-serialization issues, and invalid key adjustments. For POPRF, since public input is shared at the start of the protocol, it should incorporate domain separation techniques to prevent cross-protocol attacks. This is also important for OPRF and VOPRF when systems run multiple instances of the protocol.

B. Security Considerations and Assumptions

An OPRF protocol ensures several security properties, including those of a standard PRF. These are further extended with additional properties in the VOPRF and POPRF variants.

- **Pseudorandomness (all protocols):** The function $F(k, x)$ behaves like a truly random function when the key k is randomly sampled. An adversary without knowledge of k cannot distinguish its output from a uniformly random value. This ensures *non-malleability*, meaning an attacker cannot derive new function evaluations from observed ones. The pseudorandomness property is extended in POPRF to hold for all private and public input pairs $(x, info)$.
- **Unconditional Input Secrecy (all protocols):** The server learns nothing about the client's private input x, even with unlimited computational power. This guarantees *unlinkability*, preventing the server from correlating different PRF evaluations to the same client input. Furthermore, neither party gains knowledge of the other's secret values.
- **Verifiability (VOPRF and POPRF):** Clients must be able to confirm that the server correctly used its committed private key in computing the function output. This ensures integrity by requiring the server to prove it is following the protocol honestly. The VOPRF protocol and its security considerations are described in [6].
- **Partial Obliviousness (POPRF):** The POPRF variant allows both private and public inputs while maintaining the same security guarantees. The server learns nothing about the client's private input or function output, and the client still does not gain any knowledge of the server's private key. Public input is known to both parties, but does not compromise security.

The OPRF and VOPRF protocols are based on the One-More Gap Computational Diffie-Hellman (1MG-CDH) assumption. 1MG-CDH states that even if an efficient adversary can compute Diffie-Hellman (DH) values for several chosen group elements and has access to a Decisional Diffie Hellman oracle, they still cannot compute the DH value for one more new element that they did not query. 1MG-CDH ensures pseudorandomness, input secrecy and verifiability in OPRF and VOPRF. These guarantees hold in a single-key setting but lack formal analysis for multiple keys or batched evaluations. The POPRF variant builds upon this, leveraging the One-More Gap Strong Diffie-Hellman Inversion (1MG-SDHI) assumption, which strengthens security by reducing to the q-Discrete Logarithm (q-DL) assumption under the algebraic group model. In addition to the security properties offered by 1MG-CDH, 1MG-SDHI also offers partial obliviousness in POPRF.

A known limitation of OPRF-based protocols is their susceptibility to static Diffie-Hellman key recovery attacks, where repeated queries can gradually expose bits of the server's private key. The security loss follows a logarithmic reduction, so with sufficient queries, an adversary could marginally weaken security. To mitigate this, applications should enforce query rate limits or periodically rotate private keys. Despite these attacks, breaking a 128-bit security level remains computationally impractical in most cases. Finally, all operations involving sensitive data, including group computations and proof generation, must execute in constant time to prevent timing-based side-channel attacks.

Group	*Hash*	*Scalar Size*	*Hash Output Size*
ristretto255	SHA-512	32	64
decaf448	SHAKE-256	56	64
P-256	SHA-256	32	32
P-384	SHA-384	48	48
P-521	SHA-512	66	64

TABLE I
INITIALLY PROPOSED CIPHERSUITES

C. Supported Ciphersuites

The ciphersuites used in the protocols define the cryptographic components necessary for secure execution. Each ciphersuite consists of a prime-order group and a hash function; where the former determines the security level and the latter is responsible for mapping inputs securely within the protocol. Tab. I summarizes the initial set of ciphersuites proposed by the RFC.

VI. CONCLUSION

OPRFs are already used in practice, for example, for privacy friendly CAPTCHAs. The RFC provides a valuable contribution towards standardizing their usage which might lead to a more widespread adoption and prevent the spread of insecure deployments. However, it has the limitation of not offering a quantum secure OPRF protocol which will become more relevant in the future and some protocol variants still lack formal analysis.

REFERENCES

[1] M. Shirvanian, S. Jareckiy, H. Krawczykz, and N. Saxena, "SPHINX: A Password Store that Perfectly Hides Passwords from Itself," 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Jun. 2017.

[2] S. Jarecki, Aggelos Kiayias, H. Krawczyk, and J. Xu, "Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)," Mar. 2016.

[3] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks," Lecture Notes in Computer Science, pp. 456–486, Jan. 2018.

[4] S. Celi, A. Davidson, S. Valdez, and C. A. Wood, "Privacy Pass Issuance Protocols," Jun. 2024.

[5] A. Davidson, A. Faz-Hernandez, N. Sullivan, and C. A. Wood, "RFC 9497: Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups," IETF Datatracker, 2023.

[6] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model," Lecture Notes in Computer Science, pp. 233–253, 2014.

[7] S. Casacuberta, J. Hesse, and A. Lehmann, "SoK: Oblivious Pseudorandom Functions," 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P), Jun. 2022.

[8] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, "TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF," Lecture Notes in Computer Science, pp. 39–58, 2017.

[9] L. Heimberger, D. Kales, Riccardo Lolato, O. Mir, S. Ramacher, and C. Rechberger, "Leap: A Fast, Lattice-based OPRF With Application to Private Set Intersection," Cryptology ePrint Archive, 2025.