RFC 9497: Oblivious Pseudorandom Functions Using Prime-Order Groups

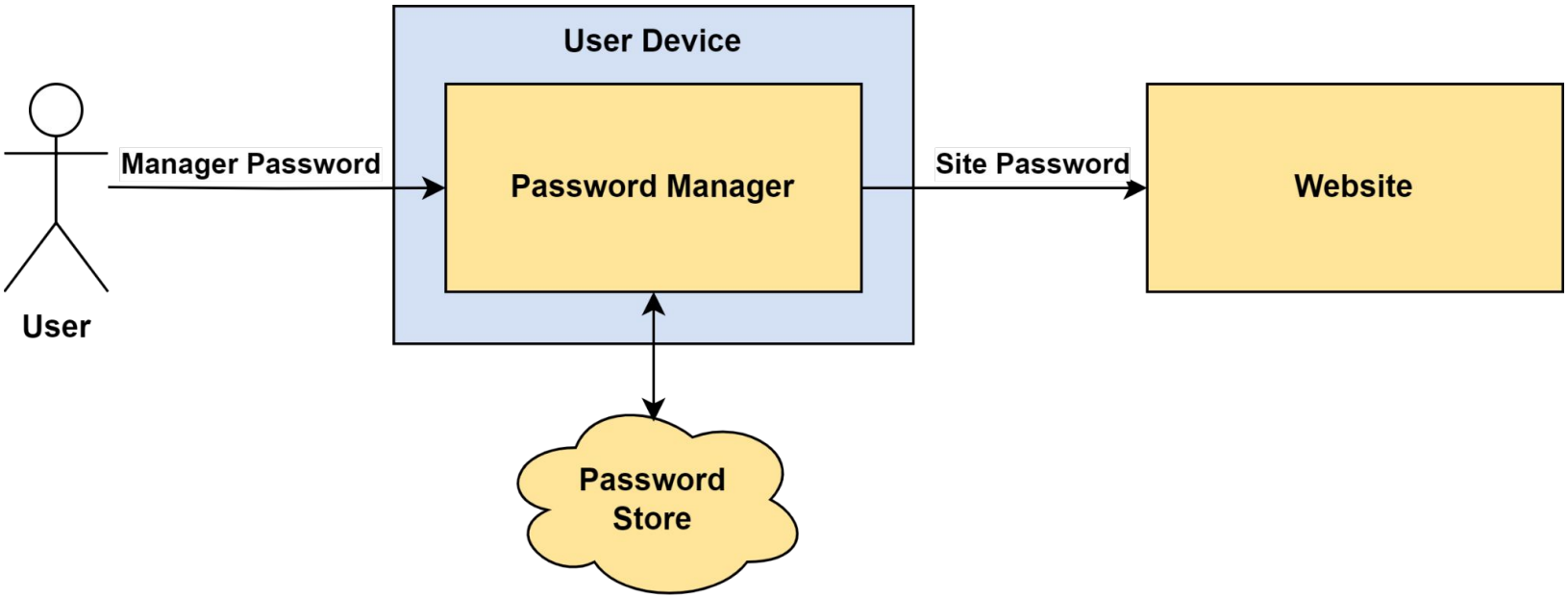
Leonard Wilhelm
Lorenz Gerk
Srividya Subramanian

07.04.2025

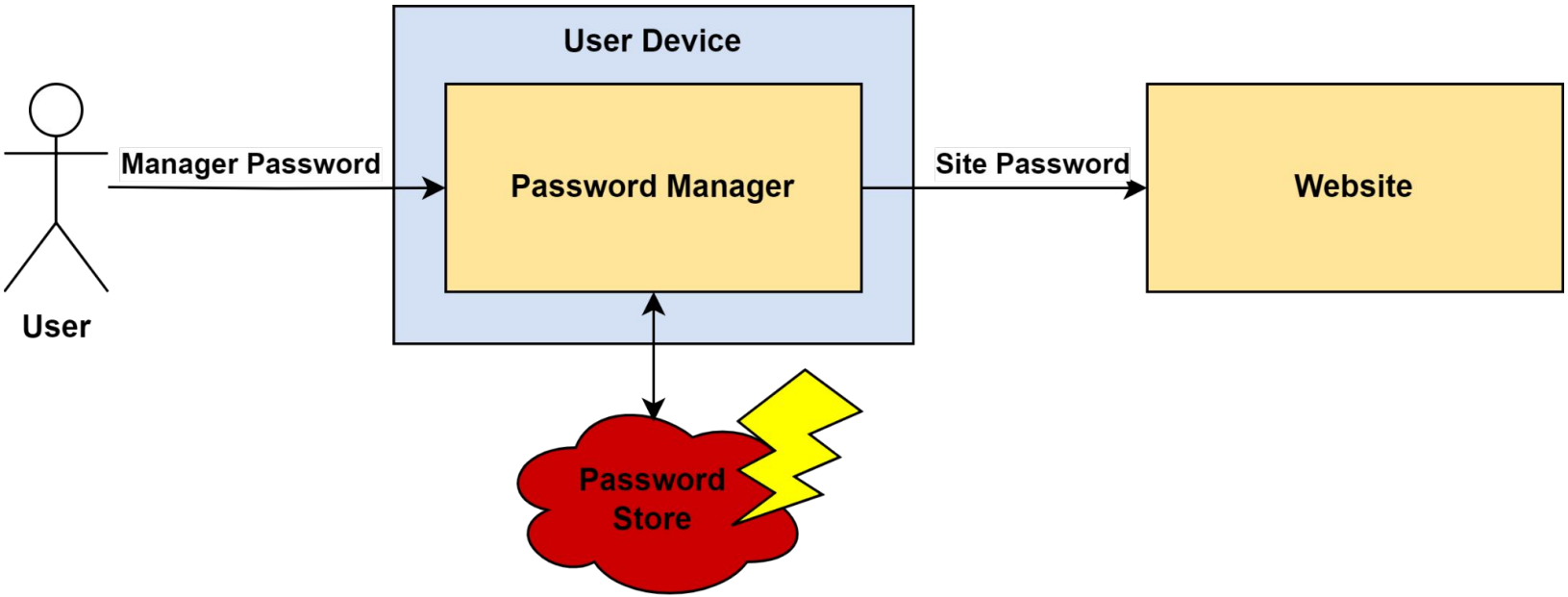
COM-506

Introduction

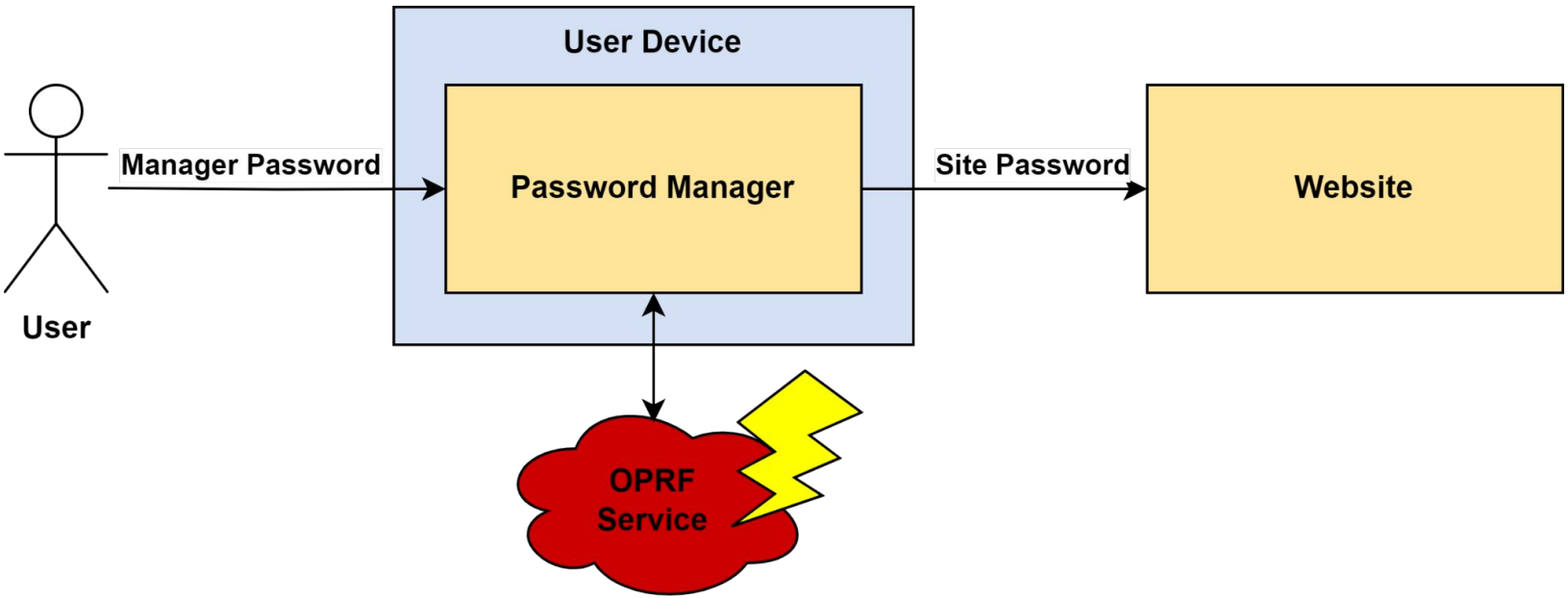
Example


User Device
Password Manager
Manager Password
Site Password
Website
User
Password
Store

Example

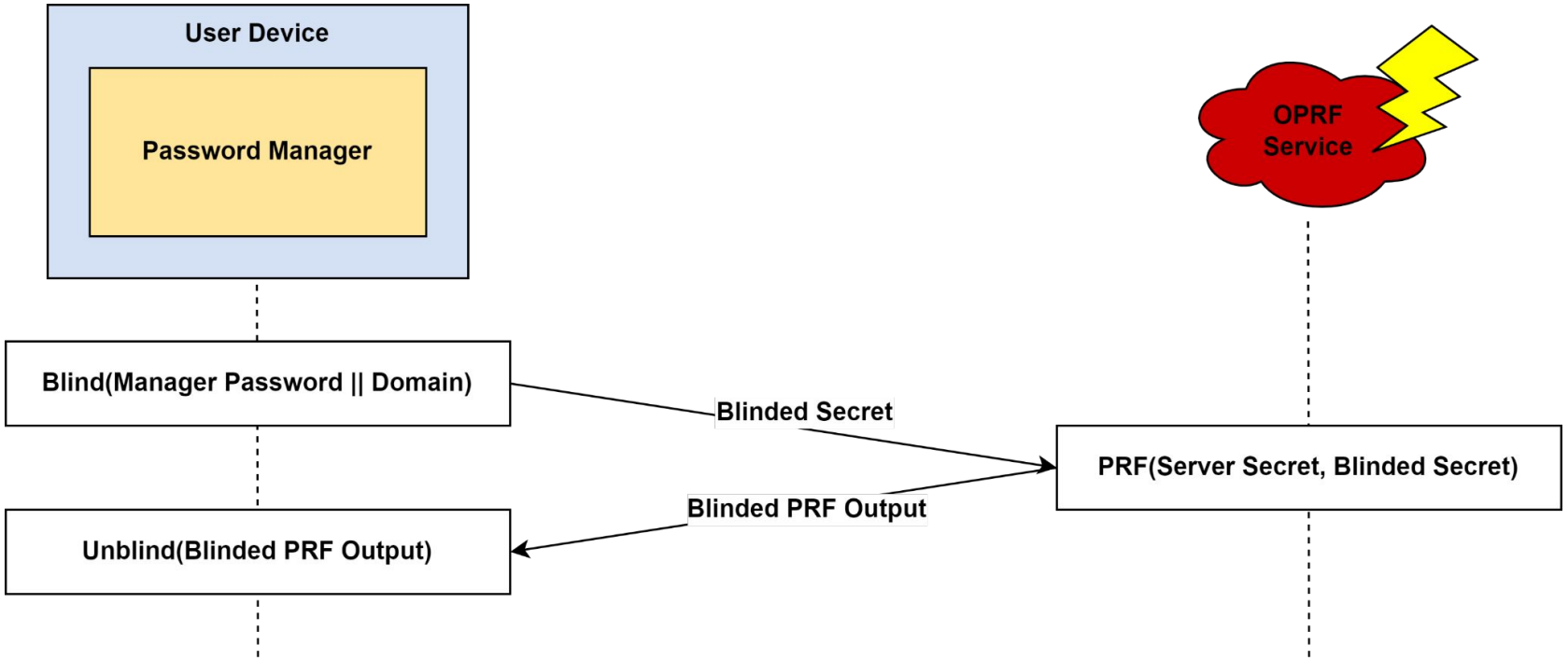

User Device
Manager Password
Password Manager
Site Password
Website
User
Password
Store

Example



Shirvanian, Maliheh, et al. "Sphinx: A password store that perfectly hides passwords from itself." *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017.

Example


User Device
Password Manager
OPRF Service
Blind(Manager Password || Domain)
Blinded Secret
PRF(Server Secret, Blinded Secret)
Blinded PRF Output
Unblind(Blinded PRF Output)

Pseudorandom Functions (PRFs)

	Client Knows	Server Knows
Oblivious Pseudo-random Function (OPRF)	input, output	secret key
Verifiable Oblivious Pseudorandom Function (VOPRF)	input, output, server's public key	secret key
Partially Oblivious Pseudorandom Function (POPRF)	input, output, server's public key, info	secret key, info

F(input, skS) = output

F(input, skS, info) = output

skS = secret key Server

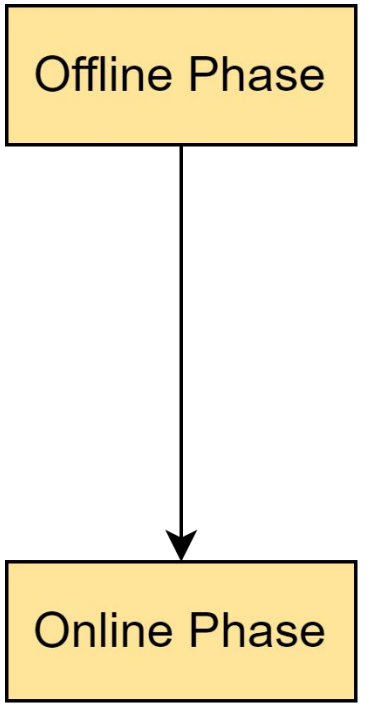
Background

Prime Order Groups

- Group consisting of:
  - Set with prime number of **elements**
  - Operation $+$: “addition” (not in the classical sense)
- **Scalar** multiplication: $k * x = x + x + x + \ldots$
  - Discrete logarithm problem: $y = k * x \rightarrow$ hard to compute k (in certain groups)
- Contains a `generator` element
  - Generates the group with scalar multiplication
- Examples:
  - Prime order groups over elliptic fields: $+ \triangleq$ Pointaddition
  - Prime order subgroups of $\mathbb{Z}_p^*$: $+ \triangleq$ Multiplication

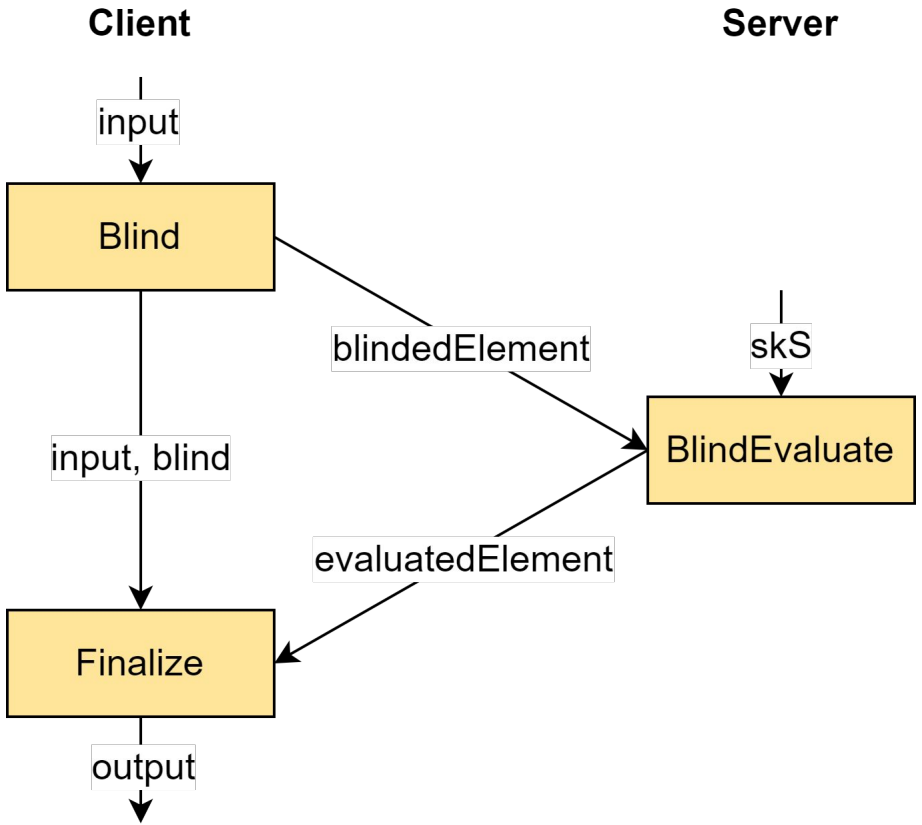
Protocol

Overview

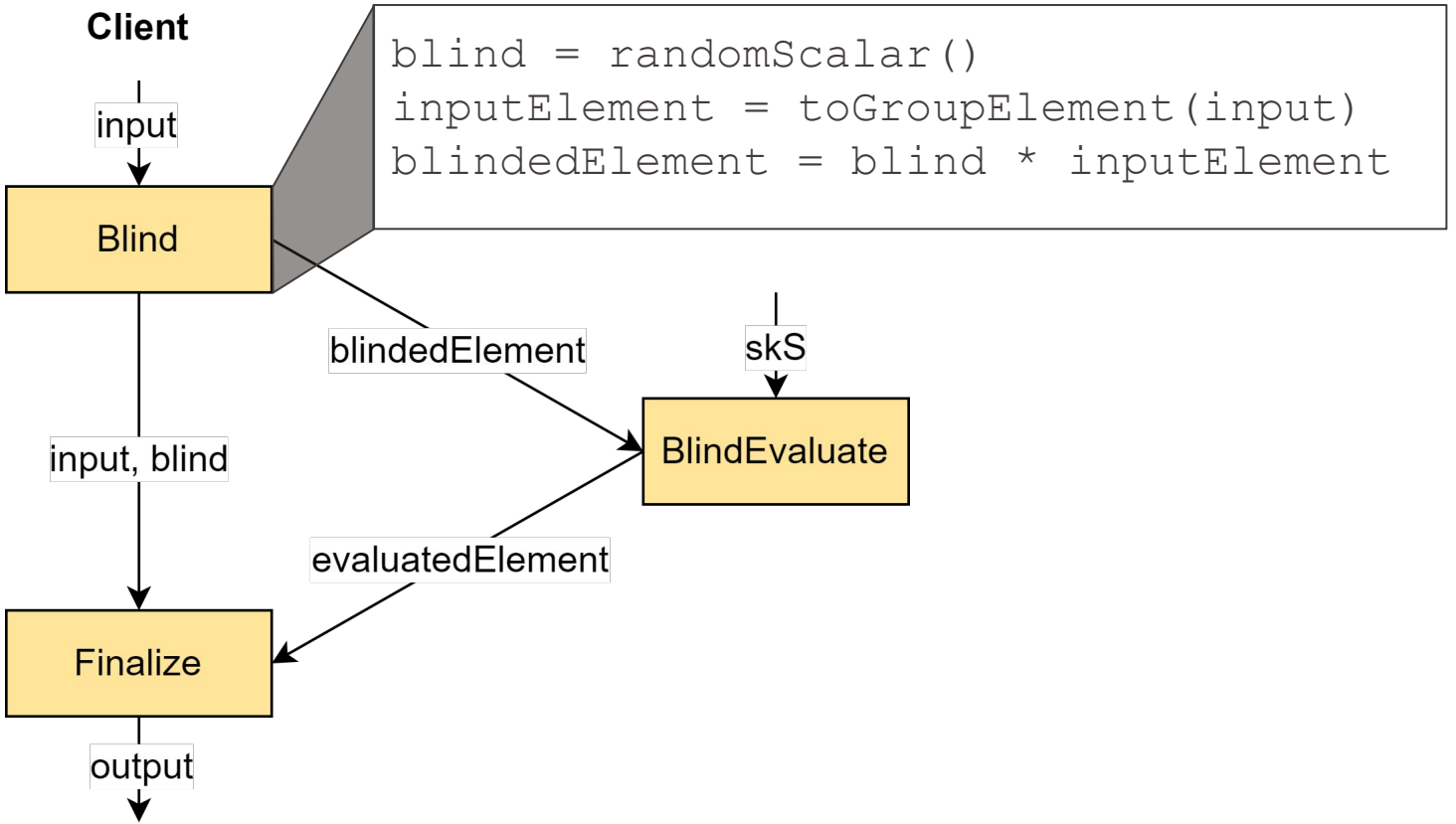


- Key generation (only server)
  - Randomly or deterministically derived
  - `skS = randomScalar()`
    `pkS = skS * generator`
- Obtain server's public key (only client for VOPRF & POPRF)
- Create protocol context consisting of:
  - Protocol variant
  - Protocol version
  - Chosen ciphersuite
  - Key material

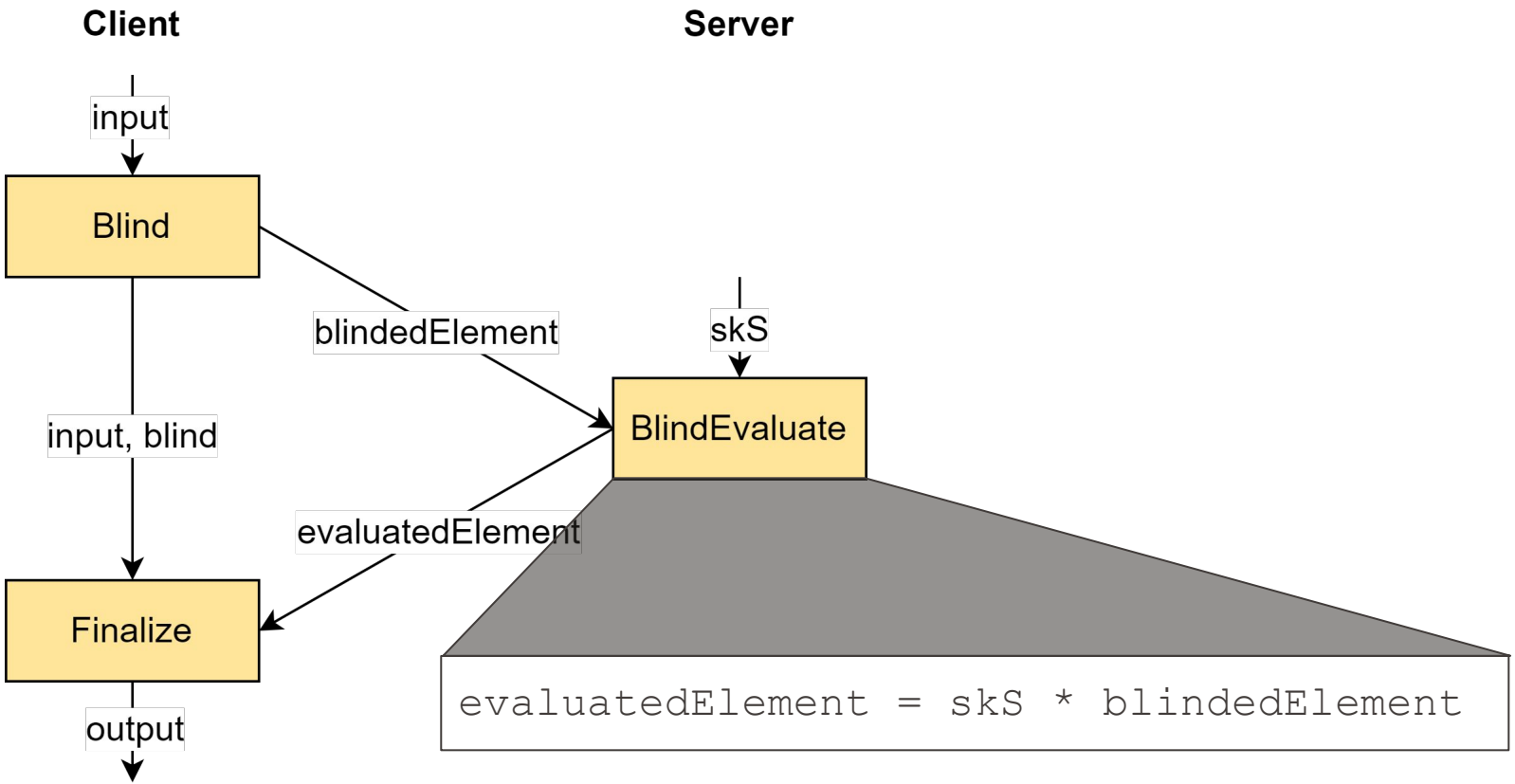
OPRF


Client
Server
input
Blind
blindedElement
skS
input, blind
BlindEvaluate
evaluatedElement
Finalize
output

OPRF


Client
input
Blind
blind = randomScalar()
inputElement = toGroupElement(input)
blindedElement = blind * inputElement
blindedElement
skS
BlindEvaluate
input, blind
evaluatedElement
Finalize
output

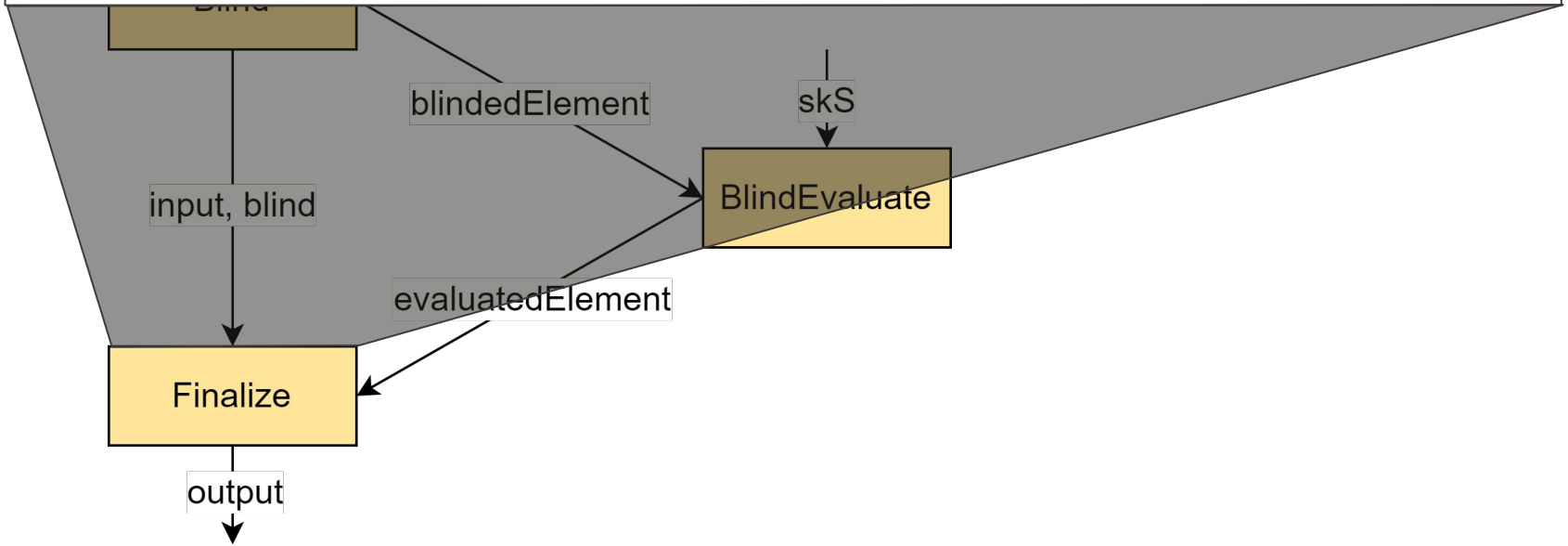
OPRF


Client
Server
input
Blind
blindedElement
skS
input, blind
BlindEvaluate
evaluatedElement
Finalize
output
evaluatedElement = skS * blindedElement

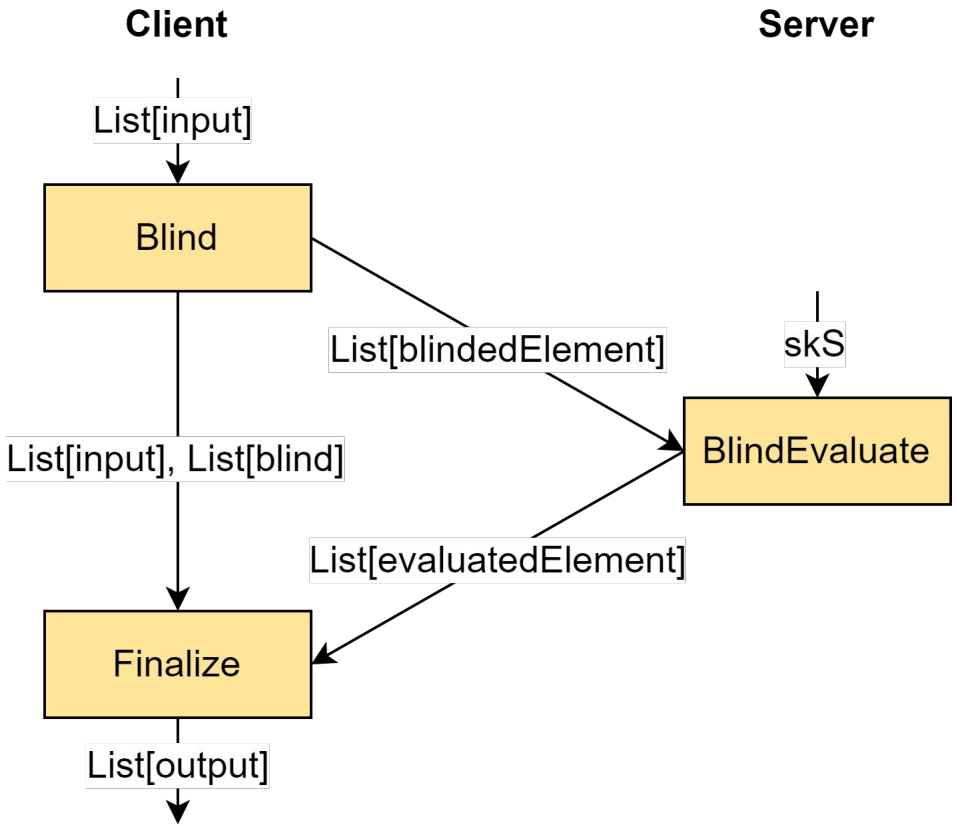
OPRF

```
unblindedElement = scalarInverse(blind) * evaluatedElement

output = Hash( len(input) || input || len(unblindedElement)
|| unblindedElement || "Finalize" )
```

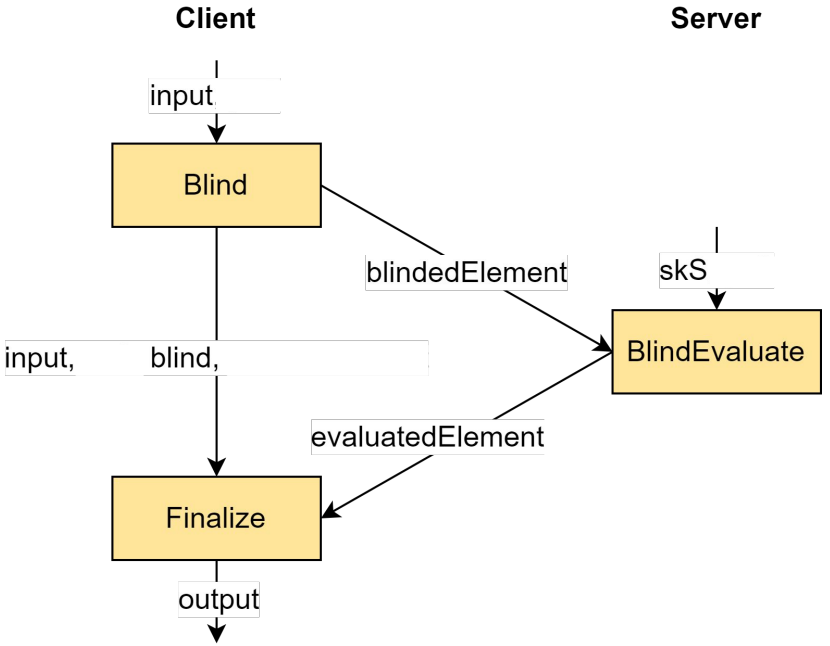


Batch Mode


Client
Server
List[input]
Blind
List[blindedElement]
skS
BlindEvaluate
List[input], List[blind]
List[evaluatedElement]
Finalize
List[output]

OPRF

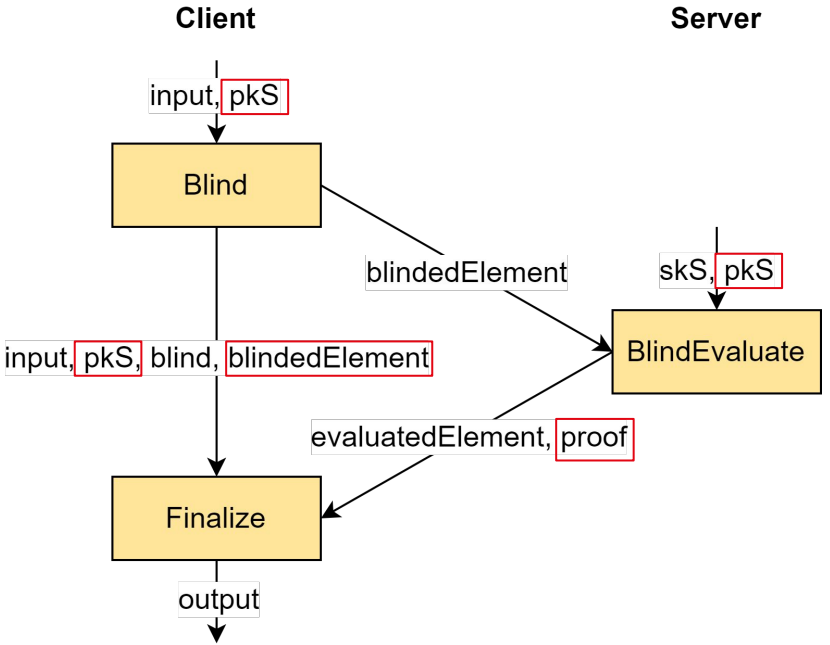
- **Problem:**
  - Server could use a value ≠ skS
  - Client has no way to notice this
- How to check if server follows the protocol, without revealing its key?



VOPRF

- **Verifiable** OPRF
- Additional ZK-proof:
  - Private key was used to generate `evaluatedElement`
- Client only accepts if proof holds

➡ Verifiable computation according to the protocol



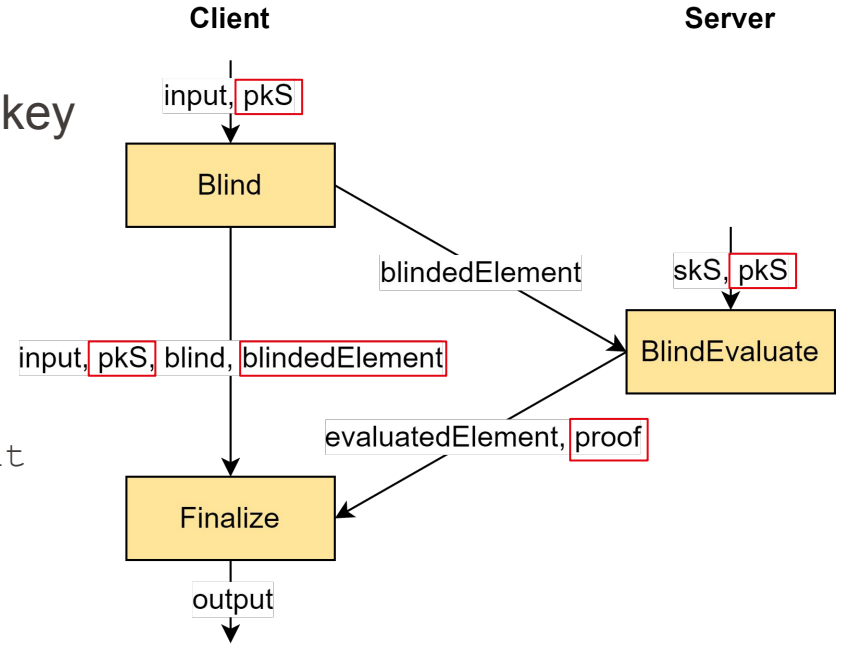
VOPRF

- Assume: Client obtained the public key

- How to show that skS was used?

```
pkS = skS * generator
evaluatedElement = skS * blindedElement
```

➡ Use zero-knowledge DLEQ proof!



Discrete Logarithm Equivalence Proof - DLEQ

- Proof that same **scalar `k`** was used to compute:
  - `B = k * A`
  - `D = k * C`
- Here:
  - `pkS = skS * generator`
  - `evaluatedElement = sks * blindedElement`
- Basic idea: return an encryption of **`k`**

Discrete Logarithm Equivalence Proof - DLEQ

- Proof that same **scalar `k`** was used to compute:
  - `B = k * A`
  - `D = k * C`
- Here:
  - `pkS = skS * generator`
  - `evaluatedElement = sks * blindedElement`
- Basic idea: return an encryption of **`k`**, e.g. `s = r - k`, `r * A`, `r * C`

```
s * A + B = r * A - k * A + k * A = r * A
s * C + D = r * C - k * C + k * C = r * C
```

- Does not reveal **`k`**: `r` not required for computation :D
- Trivially easy for the server to cheat :C

Discrete Logarithm Equivalence Proof - DLEQ

- Use transcript, i.e. `B`, `C`, `D`, `r * A` and `r * C`, to generate `c`
  - E.g. `c = H(B, C, D, r * A, r * C)`
  - Here: `c = H(pkS, blindedElement, evaluatedElement, r * generator, r * blindedElement)`
- Give Client an encryption `s` of **`k`**: **`s = r - c * k`** as well as `c`

```
s * A + c * B = r * A - c * k * A + c * (k * A) = r * A
s * C + c * D = r * C - c * k * C + c * (k * C) = r * C
```

- Client can then compute `H(B, C, D, r * A, r * C)` to verify against `c`
- Does not reveal **`k`**: `r` not required for computation :D
- Hard for server to cheat: preimage resistance of `H` :D

Discrete Logarithm Equivalence Proof - DLEQ

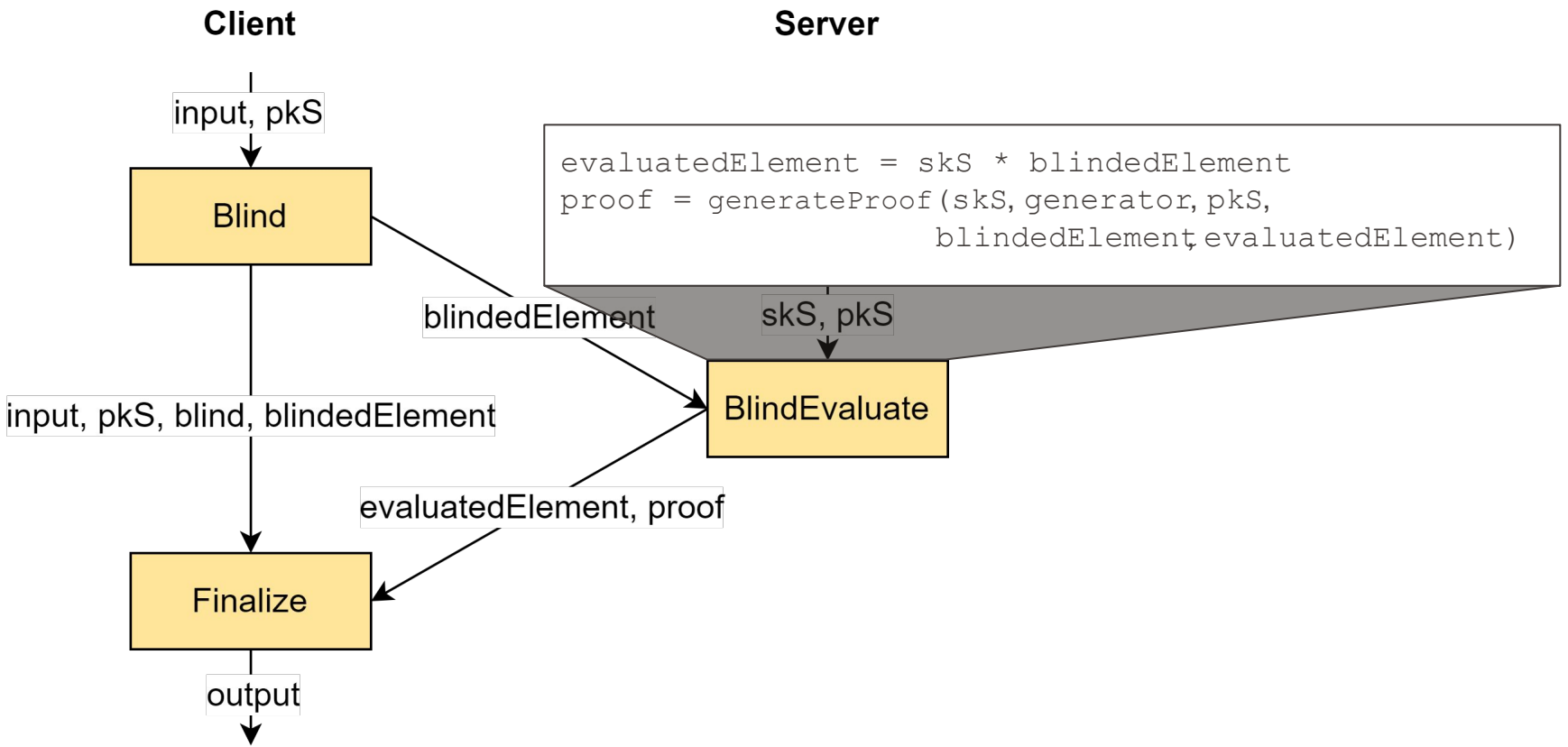
- Use transcript, i.e. `B, C, D, r *`
  - E.g. `c = H(B, C, D, r * A, r`
  - Here: `c = H(pkS, blinded`
    `r * generat`
- Give Client an encryption `s` of

`s * A + c * B = r * A - c *`
`s * C + c * D = r * C - c * k * C + c * (k * C) = r * C`

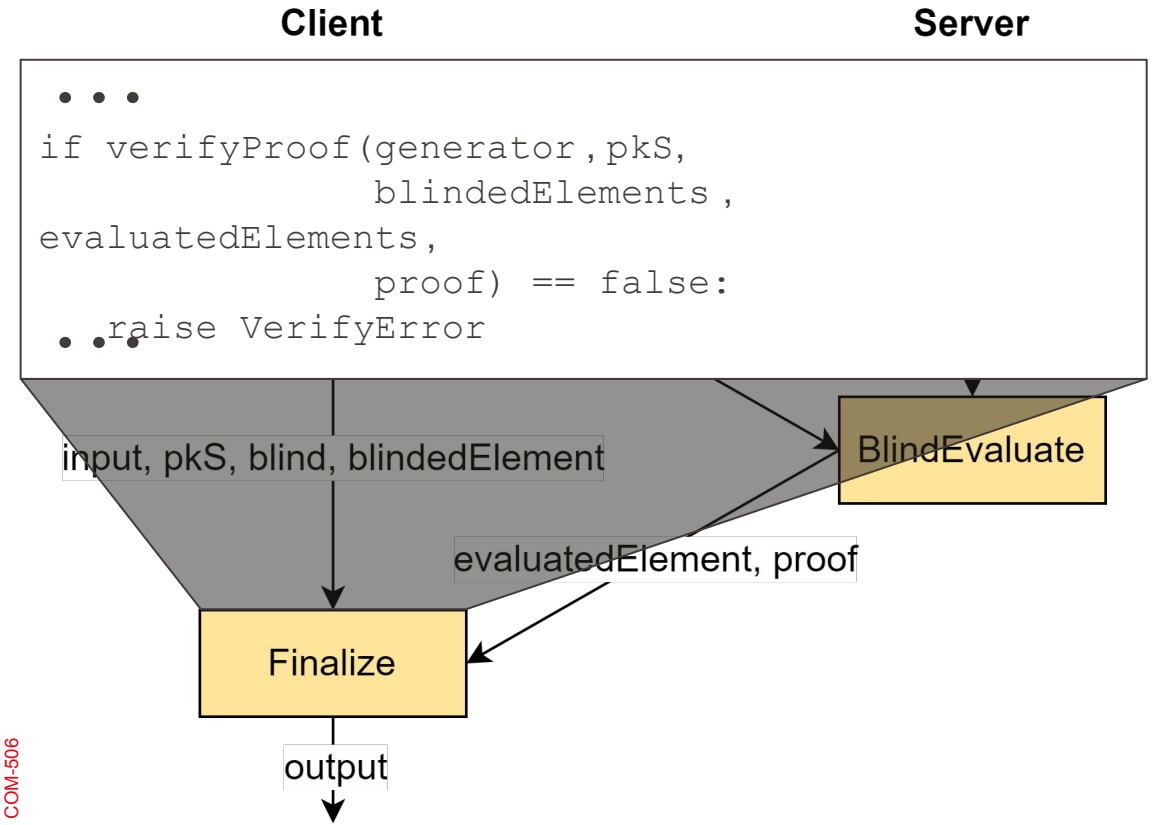
Simplified version!
- Elements first randomised:
  - r_2 `* blindedElement`, r_2 `* evaluatedElement`
  - Client can also compute r_2
- Hash function takes more inputs

- Client can then compute `H(B, C, D, r * A, r * C)` to verify against `c`
- Does not reveal `k`: `r` not required for computation :D
- Hard for server to cheat: preimage resistance of `H` :D

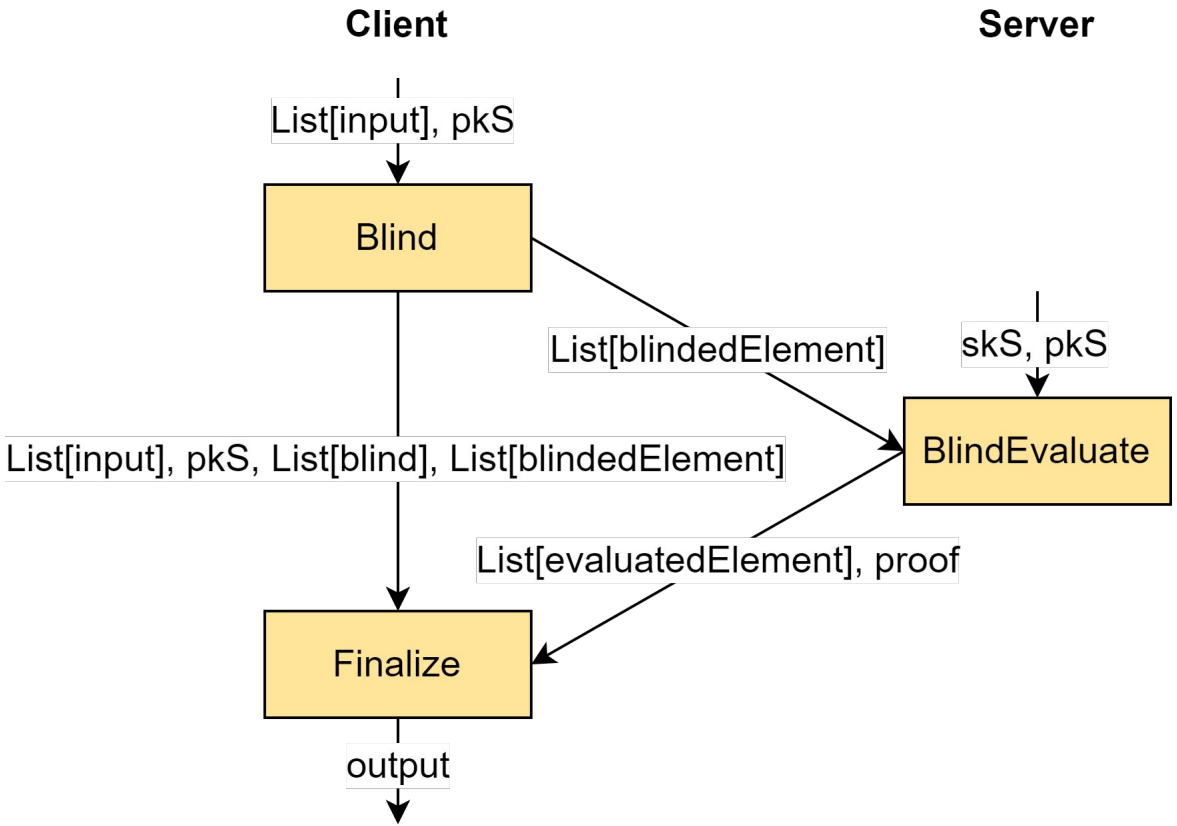
VOPRF


Client
Server
input, pkS
Blind
evaluatedElement = skS * blindedElement
proof = generateProof(skS, generator, pkS,
blindedElement, evaluatedElement)
blindedElement
skS, pkS
BlindEvaluate
input, pkS, blind, blindedElement
evaluatedElement, proof
Finalize
output

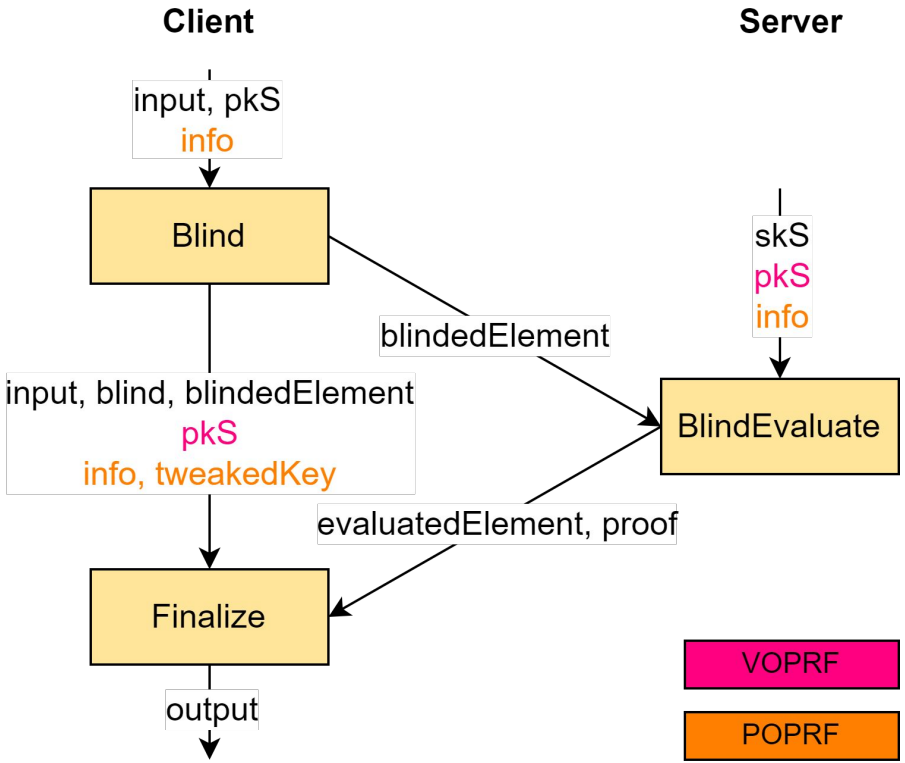
VOPRF


Client
Server
• • •
if verifyProof(generator,pkS,
blindedElements,
evaluatedElements,
proof) == false:
raise VerifyError
• • •
input, pkS, blind, blindedElement
BlindEvaluate
evaluatedElement, proof
Finalize
output

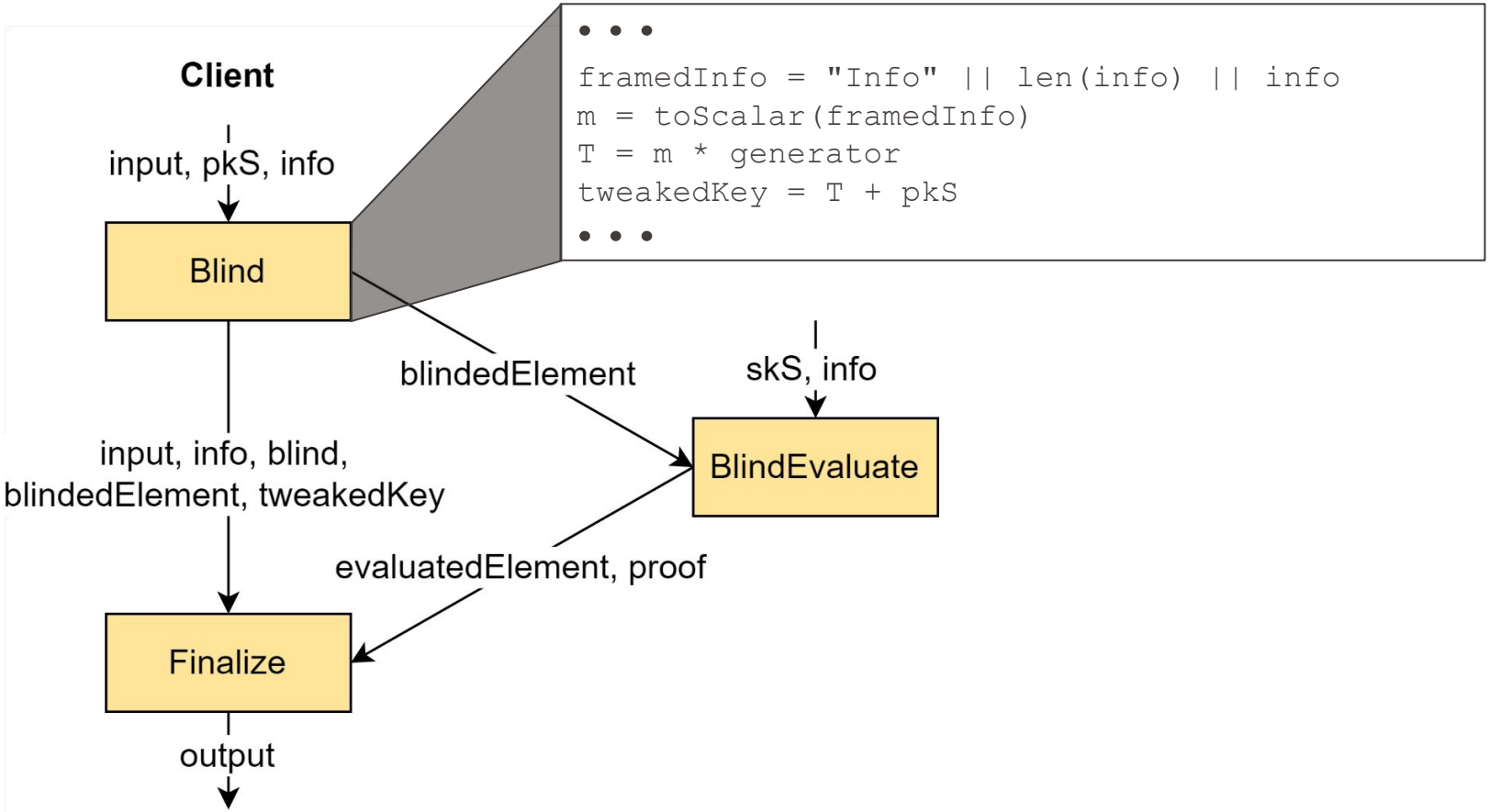
VOPRF - Batch Mode



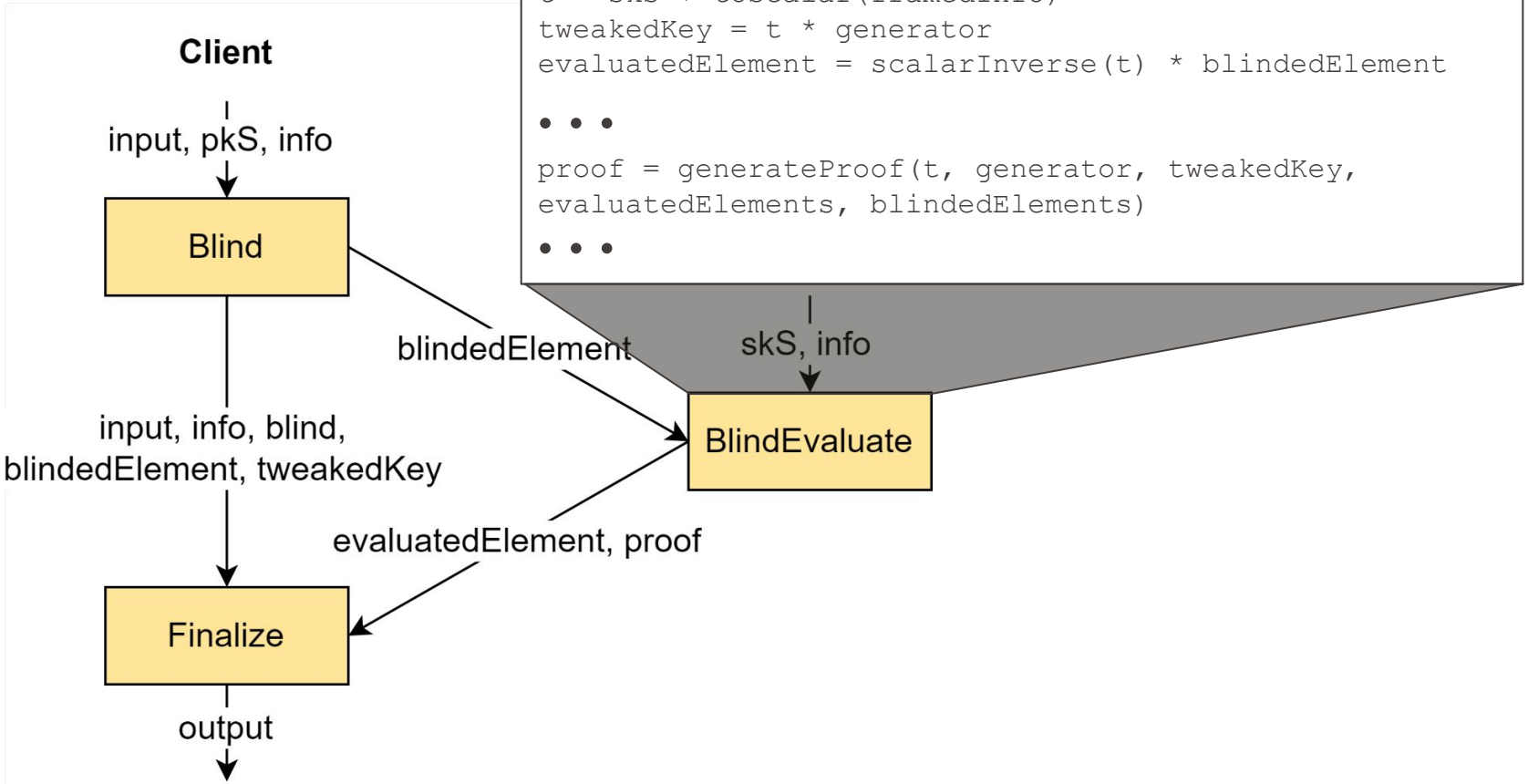
POPRF


Client
Server
input, pkS
info
Blind
skS
pkS
info
blindedElement
input, blind, blindedElement
pkS
info, tweakedKey
BlindEvaluate
evaluatedElement, proof
Finalize
output
VOPRF
POPRF

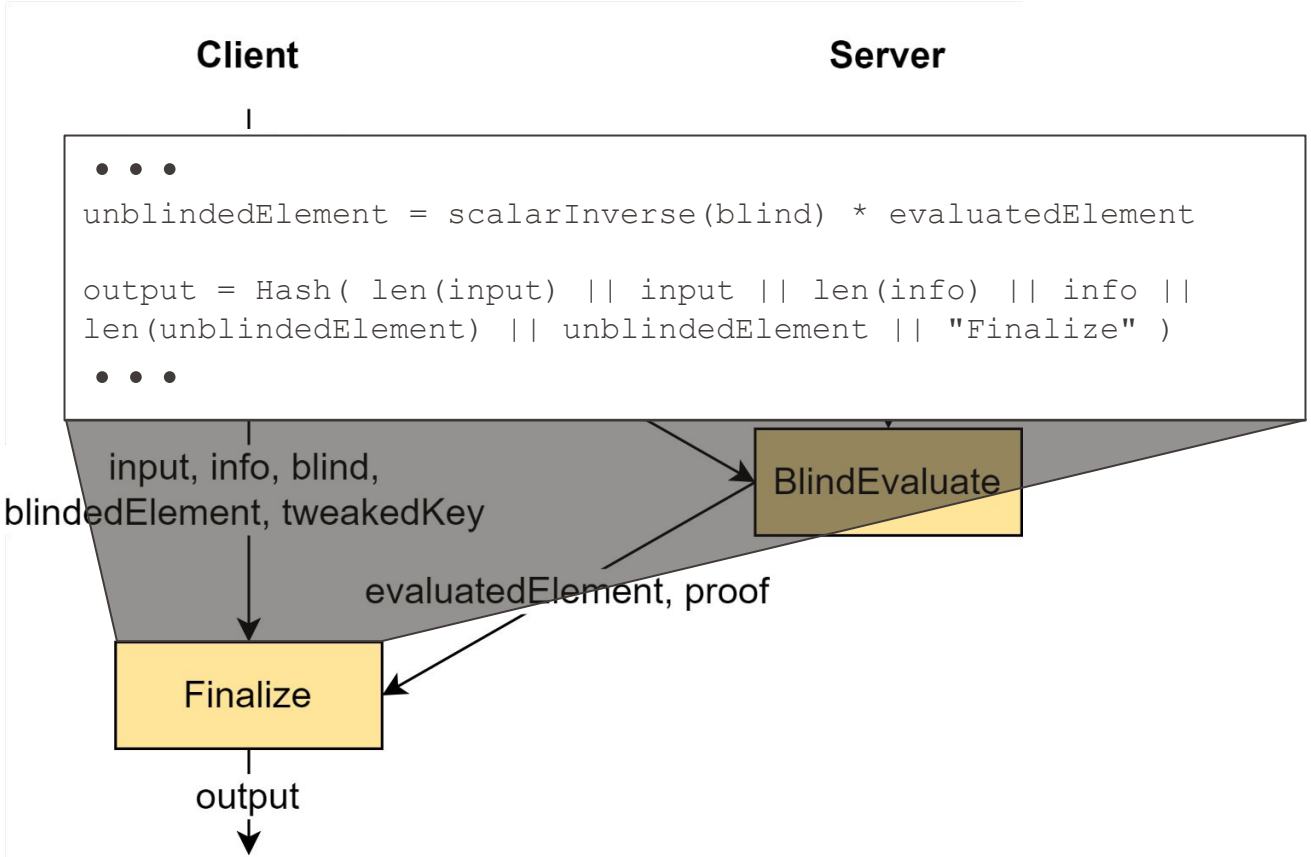
POPRF


Client
input, pkS, info
Blind
• • •
framedInfo = "Info" || len(info) || info
m = toScalar(framedInfo)
T = m * generator
tweakedKey = T + pkS
• • •
blindedElement
skS, info
BlindEvaluate
input, info, blind,
blindedElement, tweakedKey
evaluatedElement, proof
Finalize
output

POPRF


Client
input, pkS, info
Blind
blindedElement
input, info, blind,
blindedElement, tweakedKey
Finalize
output
skS, info
BlindEvaluate
evaluatedElement, proof
• • •
t = skS + toScalar(framedInfo)
tweakedKey = t * generator
evaluatedElement = scalarInverse(t) * blindedElement
• • •
proof = generateProof(t, generator, tweakedKey,
evaluatedElements, blindedElements)
• • •

POPRF


Client
Server
• • •
unblindedElement = scalarInverse(blind) * evaluatedElement
output = Hash(len(input) || input || len(info) || info ||
len(unblindedElement) || unblindedElement || "Finalize")
• • •
input, info, blind,
blindedElement, tweakedKey
BlindEvaluate
evaluatedElement, proof
Finalize
output

Security and Application Considerations

Security Assumptions

	OPRF and VOPRF	***POPRF***
Based on	Round-Optimal Password-Protected Secret Sharing and T-PAKE [1]	3HashSDHI [2]
Security Assumption	One-More Gap Computational Diffie-Hellman	One-More Gap Strong Diffie-Hellman Inversion
Security Assumption used for	Pseudorandomness, Input Secrecy, Verifiability	Pseudorandomness, Input Secrecy, Verifiability, Partial Obliviousness
Changes introduced by RFC	No session identifiers to differentiate instances of protocol; supports batching with multiple evaluations using one proof	Optionally perform multiple POPRF evaluations in one batch, using one DLEQ proof object

[1] Jarecki, S., Kiayias, A., 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
[2] Tyagi, N., Celi, S., Ristenpart, T., Sullivan, N., Tessaro, S., and C. A. Wood, "A Fast and Simple Partially Oblivious PRF, with Applications", Advances in Cryptology - EUROCRYPT 2022 pp. 674-705

1. Pseudorandomness *(OPRF and VOPRF)*

"For a random sampling of k, F is pseudorandom if the output y = F(k, x) on any input x is indistinguishable from uniformly sampling any element in F's range."

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

A consequence of this is ***non-malleability*** with high probability:
Given some output F(k, x), it should be infeasible to generate a valid output F(k, x′) for a related input x′, without knowing the key k.

1. Pseudorandomness *(Extended - POPRF)*

"For a random sampling of k, F is pseudorandom if the output y = F(k, x, info) on any input pairs (x, info) is indistinguishable from uniformly sampling any element in F's range."

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

A consequence of this is ***non-malleability*** with high probability:
Given some output F(k, x), it should be infeasible to generate a valid output F(k, x′) for a related input x′, without knowing the key k.

2. Unconditional Input Secrecy *(All Protocols)*

"The server does not learn anything about the client input x, even with unbounded computation."

Also known as ***unlinkability:*** if the server learns the client's private input at some time in the future, it cannot link any particular PRF evaluation to the input. Also applicable to the output y and the server's private key k.

3. Verifiability *(VOPRF and POPRF)*

"The client must only complete execution of the protocol if it can successfully assert that the output it computes is correct. This is taken with respect to the private key held by the server."

For verifiability to hold in practice, the server commits to its public key before the protocol execution takes place. Then, the client verifies that the server has used the key in the protocol using the proof.

4. Partial Obliviousness *(POPRF)*

Despite knowing the public input, the server learns nothing about the client's private input or the output, and the client learns nothing about the server's private key.

The involved parties must be able to perform the PRF on the client's private and public input. This property is useful while dealing with key management operations. For ex., rotation of the server's keys.

Security Considerations

- ***Static Diffie-Hellman Attack due to RFC variations***
  Possible to exploit `BlindEvaluate` queries to recover bits of the server's private key if the adversary can query the OPRF directly. Can be mitigated via rate-limiting, or stronger prime-order groups.

- ***Domain Separation***
  Any system that has multiple OPRF applications should distinguish client inputs to ensure the OPRF results are separate.

- ***Timing Leaks***
  All operations involving secret data — including group operations, `GenerateProof`, and `BlindEvaluate` — must run in constant time to prevent timing side-channel leaks during protocol execution.

Application Considerations

- ***Input Limits***
  Application inputs must be under $2^{16} - 1$ bytes; longer inputs should be hashed to a fixed length before use.

- ***External Interface***
  Protocols use group elements and scalars internally, but interfaces should expose clean, app-specific input/output formats.

- ***Error Handling***
  Operations like `BlindEvaluate` and `Finalize` may fail with defined errors implementations should handle these safely.

Conclusion

Conclusion

- OPRFs already used in praxis, e.g. Privacy Pass
- RFC is valuable contribution towards standardizing OPRFs
- Future Work:
  - Quantum secure OPRFs
  - Further formal analysis

Thank you!

Any questions?