COCO: Coconuts & Oblivious Computations for Orthogonal Authentication

Yamya Reiki

December 6, 2024

Abstract

Authentication often bridges real-world individuals and their virtual public identities, like usernames, user IDs and e-mails, exposing vulnerabilities that threaten user privacy. This research introduces COCO (Coconuts & Oblivious Computations for Orthogonal Authentication), a framework that segregates roles among Verifiers, Authenticators, and Clients to achieve privacy-preserving authentication.

COCO eliminates the need for Authenticators to access virtual public identifiers or real-world identifiers for authentication. Instead, the framework leverages Oblivious Pseudorandom Functions (OPRFs) and an extended Coconut Credential Scheme to ensure privacy by introducing separate unlinkable orthogonal authentication identifiers and a full-consensus mechanism to perform zero-knowledge authentications whose proof-s are unlinkable across multiple sessions. Authentication process becomes self-contained, preventing definitive reverse tracing of virtual public identifiers to real-world identifiers.

Introduction

Authentication, by definition, creates a bridge between a real-world individual and their virtual public identity, like usernames, e-mails etc., making it a critical point of vulnerability in preserving user privacy.

Usernames, e-mail addresses, and other virtual public identifiers can often be linked to real-world individuals through associated public content the user may share or inherent characteristics—such as personally identifiable substrings within the identifiers themselves.

Now, when an authenticator has knowledge of the authenticatee—such as personal information (e.g. secrets, or biometric data)—and also has knowledge of their virtual public identity, it leads to a two-fold risk to user privacy: first, by linking the virtual identity to public content or characteristics that reveal personal information, and second, by exposing sensitive personal data through

reverse-engineering or brute-force to identify the individual. Against a legal adversary, or otherwise, this two-fold vulnerability suffices to prove a definitive link between a virtual public identity and a real world individual. However, if the authenticator operates without any knowledge of the authenticatee—neither their virtual public identifier nor their personal information—it becomes possible to safeguard user privacy at its core.

In this work, I introduce COCO (Coconuts & Oblivious Computation for Orthogonal Authentication) as a solution to these vulnerabilities. COCO defines distinct roles: Verifier, Authenticators, and Client. The Client represents the real world user, and temporarily knows the user's real world identifiers, such as their secrets, or biometrics. However, the Client never retains such information. Verifier is an entity responsible for validating if a Client has full-consensus authenticity from all the required authenticators. Verifiers can simultaneously be the resources or services the Client wishes to access. Verifiers manage virtual public identifiers, like usernames, and user IDs, but cannot access or verify authentication data, such as authentication identities, neither do they have any knowledge of a user's personal information or secrets. Authenticators, in contrast, generate credentials (access-tokens) without knowing either virtual public identifiers (e.g., usernames, user IDs), or personal information (e.g., secrets or biometrics). Instead, the Authenticators make use of separate authentication identities that are unlinkable to any of the aforementioned identities known by either Client or Verifier. This careful separation of roles ensures that authentication remains valid while unlinkable to the user's real-world identity or any publicly known virtual identities.

As a result, even if a username, e-mail address, or user ID is associated with public content or personally identifiable information (PII), COCO prevents tracing it back to the individual through authentication. Authentication becomes a self-contained mechanism, incapable of acting as a back-traceable bridge to the real person. By design, COCO disrupts the association between digital fingerprints (such as usernames or user IDs) and real-world identities, delivering privacy and unlinkability in the authentication process.

COCO aspires to create a digital ecosystem where users have greater control over their online presence. It empowers individuals to participate in online activities without the fear that their real world identities could be exposed or traced. Whether users choose to share their personal information or keep it private, COCO wishes to ensure their autonomy remains intact.

Overview of COCO

Coconuts & Oblivious Computations for Orthogonal Authentication (COCO) is a full consensus zero-knowledge authentication protocol that decouples the real world identifiers from digital fingerprints by introducing unlinkable authentication identifiers.

It has been used to implement an Identity Management System that supports user registration, login, deletion, setting up a second factor authentication,

and username and password updates. We enlist the design goals of COCO Authentication Protocol:

- **Full Consensus Authentication:** If a Client chooses a subset of Authenticators during account creation with a cardinality of n, then all of the n Authenticators are required to consent to subsequent authentications. This accounts for a theoretical communication complexity of $\mathcal{O}(n)$ during account creation and authentication phases.

- **Orthogonal, Unlinkable & Zero Knowledge Authentication:** Instead of asking the question, *"who owns a link to what?"*, the Authenticators must only be concerned about the proof of ownership of the *link* between the *who*, and the *what*, and the *link* in-itself must not reveal any information about either-or too. The *who* is the real world indentity, the *what* is the virtual public identity or digital fingerprint and the *link* is the authentication identity. Hence, in essence, both the *link* and its ownership appear orthogonal of the *who*, and the *what*. Further, multiple reveals of the ownership of the link are unlinkable to each other as well as the record/metadata generated during issuance of the reveal (zero-knowledge ownership proof).

- **Asynchrony:** Given common knowledge of parameters related to Coconut Credential Scheme used in the protocol, the Authenticators are able to operate independently and asynchronously.

- **Scalability:** It is necessary to assume only one Verifier per set of Authenticators as a preventive measure against possible user identity collision attacks. However, within a single Verifier's domain, there can be any number of independent and asynchronous Authenticators as well as Clients. Further each Client may interact with any subset of Authenticators asynchronously and independently. Hence, the system remains scalable as long as Authenticator subset overuse is prevented and Authenticators can handle the communication load imposed by multiple Clients.

- **Liveness:** Given that per-Client Authenticator subsets remain honest, COCO guarantees liveness for those Clients without any synchrony assumptions.

- **Efficiency:** The theoretical communication and computational complexity of COCO is, for the most part of it, a direct inheritance from the complexities of Coconut Credential Scheme and OPRF implementation. Hence, assuming 2HDH PRF from RFC9497[7] for OPRF, which is $\mathcal{O}(1)$ in a broader high-level abstraction, COCO accounts for an overall theoretical communication and computational complexity of $\mathcal{O}(n)$, where n is the subset of authenticators chosen by a Client for its COCO authentication with a Verifier. This is because the credential *request* and *issue* protocols in Coconut Credential Scheme are $\mathcal{O}(n)$ for n issuers, the protcol uses

short and efficient credentials and zero-knowledge proofs, and the credential *aggregation*, *re-randomization*, *showing* and *verification* protocols are all $\mathcal{O}(1)$ both computationally and communicatively[11].

To ensure the protocol's resistance to GPU- and ASIC-optimized brute-force attacks during Client-side operations, COCO incorporates a memory-hard hash function for specific hash-based operations. The use of a memory-hard hash function accounts for $\mathcal{O}(m)$ during hash computations, where m is the memory size parameter.

For practical implementations, this is typically a constant overhead tuned to balance security and efficiency. The inclusion of memory-hard functions does not directly affect communication complexity as their operations are local to the Client. Thus, the overall complexity of COCO remains predominantly $\mathcal{O}(1)$ for n authenticators, with the added consideration of $\mathcal{O}(m)$ for operations involving memory-hard hash functions, where m is chosen to reflect an appropriate trade-off between efficiency and attack resistance.

Technical Background

Foremost, COCO assumes usage of a memory-hard password hashing function $\mathcal{H}$, such as Argon2, or Scrypt, throughout its implementation. At its heart, COCO is based off of two major existing Privacy Enhancing Technologies- Oblivious Pseudo-random Functions, and Coconut Credential Scheme for Unconditional Privacy - hence the name, Coconuts & Oblivious Computations for Orthogonal Authentication! The OPRF protocol is assumed to be either directly confirmatory to the RFC 9497[7], or some probable implementation of Dodis-Yampolskiy (DY) PRF[8] with no required modifications. The Coconut Credential Scheme for Unconditional Privacy implementation is, however, based off of work by Sonnino et. al.[11], with an additional functionality tweak, **DeriveKey**11, for deterministically generating Coconut credential verification key from a user provided signing key seed value. This little functionality has been provided to facilitate COCO's personal requirement of Coconut Scheme-based self-signed credentials. We directly use the **Setup**, **KeyGen**, **PrepareBlindSign**, **BlindSign**, **Unblind**, **AggCred**, **ProveCred**, **VerifyCred**, **IssueCred** and **AggKey** functions for Unconditional Privacy from Sonnino et. al.[11], with no required modifications, to derive our API-s for Coconut protocol.

Memory-Hard Password Hashing Function

- **Definition:** Let $Hash : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function, where n is the output size in bits. A memory-hard password hashing function $\mathcal{F}$ is defined as:

$$\mathcal{F}(P, S, T, M) \to \{0,1\}^n$$

where:

- P: The password (or input string) from the user.
- S: A unique salt value to ensure output uniqueness for each P.
- T: The time cost parameter controlling the number of iterations.
- M: The memory cost parameter controlling the amount of memory used.

$\mathcal{F}$ is memory-hard if the amount of computational effort (measured as time-area product, TAP) required to compute $\mathcal{F}$ is minimized when the memory usage M is maximized for a fixed T.

- **API Overview:**
  - $\mathcal{H}(input)$: Computes a digest for given input using a memory-hard password hashing function.
- **Properties of API:**
  - $\mathcal{H}$ is computationally infeasible to invert.
  - $\mathcal{H}$ ensures resistance to brute-force attacks due to its controllable time and memory parameters.
  - $\mathcal{H}$ provides resistance to hardware-accelerated attacks (e.g., GPUs, ASICs) by imposing high memory costs.

Oblivious Pseudo Random Function

- **Definition:** An Oblivious Pseudo Random Function (OPRF) is a two party computation, where the two parties (suppose: a client and a server) jointly compute the output of a pseudorandom function $F_k(x)$ on an input x chosen by the client, without the server learning x and without the client learning the server's secret key k. It is an oblivious computation because of the client's input privacy and the server's key privacy.
  - **Two-Hash Diffie-Hellman (2HashDH) PRF:** Hereon, we take our definition for OPRF Protocol from 2HashDH PRF $f_k^{2H}(x)$ as in Casacuberta et. al.[5]:

$$f_k^{2H}(x) = H'(x, H(x)^k),$$

    where:
    - $G = \langle g \rangle$: A group of prime-order q.
    - x: The client's input, from $\mathbb{Z}_q$.
    - H: A cryptographic hash function mapping x to uniformly random elements in G.
    - k: The server's secret key, randomly chosen from $\mathbb{Z}_q$.
    - H': An additional cryptographic hash function applied by the client to ensure pseudorandomness and bind the input x explicitly to the output.

Oblivious evaluation of $f_k^{2H}(x)$ relies on the One-More-gap Diffie-Hellman assumption, which states that it is infeasible for an adversary to compute multiple DH values given access to an oracle solving the computational DH problem. Hence, the hash functions H and H' are modeled as random oracles, ensuring pseudorandomness and hiding properties. Furthermore, under appropriate assumptions, $f_k^{2H}(x)$ can be proven secure in the Universal Composability (UC) framework[4].[5]

- **Dodis-Yampolskiy (DY) PRF:** Alternatively, the OPRF Protocol can be based on DY PRF $f_k^{DY}(x)$ based on the Boneh-Boyen unpredictable function[1] and defined by Casacuberta et. al. [5] as:

$$f_k^{DY}(x) = g^{1/(k+x)},$$

where:

  - $G = \langle g \rangle$: A cyclic group of prime order q.
  - k: The server's secret key, randomly chosen from $\mathbb{Z}_q$.
  - x: The client's input, also from $\mathbb{Z}_q$.

Oblivious evaluation of $f_k^{DY}(x)$ uses additively homomorphic encryption schemes, enabling the client to evaluate $f_k^{DY}(x)$ without revealing x to the server. The pseudorandomness of $f_k^{DY}(x)$ relies on the q-Decisional Diffie-Hellman Inversion (q-DDHI) problem, which ensures its security over polynomial-sized domains.[5] Further, Zero-knowledge proofs can be integrated (e.g., Camenisch et al.[3]) to ensure malicious security, where both server and client operate on encrypted inputs.[5] These properties and assumptions make DY PRF more secure than 2HashDH PRF in many use cases, but at a significant cost of efficiency.

- **API Overview:** We design APIs for the OPRF protocol based on RFC 9497[7], itself built on the 2HashDH PRF of Jarecki et al.[9], balancing security and efficiency:

  - $\mathcal{OPRF}_{Blind}(input = x)$: The client computes a blinded representation of x using a random scalar blinding factor $r \in \mathbb{Z}_q$. The computation is:

$$a = r \cdot H(x),$$

where $\cdot$ is scalar multiplication in G. The client sends a to the server.

  - $\mathcal{F}_{OPRF}(blinded\_input = a)$: The server applies its private key k to the blinded input a, computing:

$$b = k \cdot a = k \cdot r \cdot H(x)$$

and sends b back to the client.

– $\mathcal{OPRF}_{Unblind}(blinded\_output = b)$: The client removes the blinding factor r, computing:

$$b/r = (k \cdot r \cdot H(x))/r,$$

where / denotes the inverse scalar operation in G, yielding $k \cdot H(x)$. Finally, the client applies $H'(x, k \cdot H(x))$ and computes the PRF output $f_k^{2H}(x)$.

- **Properties of API:**
  - **Unlinkability:** The server learns no information about the client's input x. The client gains no information about the server's secret key k. The OPRF output $f_k^{2H}(x)$ is indistinguishable from a random value to any party without k or x.
  - **Autonomous:** Each of the OPRF server making use of $\mathcal{F}_{OPRF}()$ is essentially autonomous in its functioning.
  - **UC-Security:** Under appropriate assumptions and proper implementations, $f_k^{2H}(x)$ can be proven secure in the Universal Composability (UC) framework.[5][4]
  - **Efficiency Constraints:** Assuming scalar multiplication in G is constant, our OPRF API assumption accounts for computational and communication complexity of $\mathcal{O}(1)$.

Coconut Credential Scheme for Unconditional Privacy

- **Definition:** In accordance with Sonnino et. al.[11], the Coconut Credential Scheme can be summarized for our use-case as the following quadruple:

$$(\text{Setup}, \text{User}, \text{Issuer}, \text{Verifier}),$$

where:

  - **Setup =** (Setup, Inputs, Outputs) : is a triplet such that:
    - Setup(q) $\rightarrow$ (params) : is the function that takes an integer (q) representing the maximum number of attributes that can be embedded in the credentials from Inputs and maps the publicly known parameters (params) used throughout the protocol to Outputs.
  - **Issuer** = (KeyGen, BlindSign, Inputs, Outputs) is the quadruple, such that:
    - KeyGen($params$) $\rightarrow$ (sk_i, vk_i) : is the function that takes ($params$) from Inputs and maps the secret key (sk_i) and verification key (vk_i) of the Issuer, to Outputs.

* BlindSign(params, sk_i, Λ, public_m) $\rightarrow$ (σ'_i): is the function that takes the public parameters (params), secret key of this Issuer (sk_i), commitments and encryptions of attributes sent by the User (Λ), and optional array of public attributes sent by the User (public_m) as Inputs, to map a blinded credential (σ'_i) to Outputs.

– **Verifier =** (AggKey, VerifyCred, Inputs, Outputs) is the quadruple, such that:

* AggKey(params, vks) $\rightarrow$ aggr_vk: is the function that takes public parameters (params) and array of all verification keys (vks) from all the Issuers as Inputs, and aggregates them into (aggr_vk) as Outputs.
* VerifyCred(params, aggr_vk, Θ, public_m) $\rightarrow$ ret: is the function that takes public parameters (params), aggregated verification key (aggr_vk), credential and associated cryptographic material (Θ) provided by User, and the optional array of public attributes (public_m) provided by User as Inputs, and Outputs a boolean indicating whether the credential is valid.

– **User =** (PrepareBlindSign, UnblindSign, AggCred, AggKey, ProveCred, Inputs, Outputs) is the septuple, such that:

* PrepareBlindSign(params, private_m, public_m) $\rightarrow$ (Ls, Λ): is the function that takes takes the public parameters (params), the array of private attributes (private_m), and the optional array of public attributes (public_m)) as Inputs, and produces blinding factors (Ls) and commitments and encryptions (Λ) as Outputs for blind signing.
* UnblindSign(params, σ'_i, d) $\rightarrow$ σ_i: is the function that takes the public parameters (params), and a blinded credential (σ'_i), and user's El-Gamal private key as Inputs and Outputs the corresponding unblinded credential (σ_i).
* AggCred(params, sigs, Ls) $\rightarrow$ σ: is the function that takes the public parameters (params), the array of all unblinded partial credentials (sigs), the blinding factors (Ls) as Inputs and produces an aggregated unblinded credential (σ) as Outputs.
* AggKey(params, vks) $\rightarrow$ aggr_vk: is the function that takes public parameters (params) and array of verification keys (vks) from all the Issuers as Inputs, and aggregates them into (aggr_vk) as Outputs.
* ProveCred(params, aggr_vk, σ, private_m) $\rightarrow$ Θ: is the function that takes the public parameters (params), the aggregated verification key (aggr_vk), the aggregated unblinded credential (σ), and the array of private attributes (private_m) as Inputs and builds cryptographic material for credential proof (Θ) as Outputs.

- **Extension:** Coconut Credential Scheme as defined above has been extended to fascilitate integration into COCO Authentication Protocol. We extend the scheme to include a self-signing authority 'SelfIssuer' as follows:

$$(\text{Setup}, \text{User}, \text{Issuer}, \text{SelfIssuer}, \text{Verifier}),$$

where:

  - **SelfIssuer** = (DeriveKey, IssueCred, ProveCred, Inputs, Outputs) is the quintuple,
    such that:
    * $\text{DeriveKey}(params, input_{bytes}) \rightarrow (\text{sk}, \text{vk})$: is the function that takes the public ($params$), along with a deterministic seed for secret key ($input_{bytes}$) from Inputs and maps the secret key (sk) and verification key (vk) of the SelfIssuer, to Outputs. See Algorithm 11.
    * $\text{IssueCred}(\text{params}, \text{private}_m, \text{public}_m) \rightarrow (\sigma)$: is the function that takes the public parameters (params), the array of private attributes (private_m), and the optional array of public attributes (public_m)) as Inputs and maps an unblinded credential (σ) to Outputs.
    * $\text{ProveCred}(\text{params}, \text{vk}, \sigma, \text{private}_m) \rightarrow \Theta$: is the function that takes the public parameters (params), the verification key (vk), the unblinded credential (σ), and the array of private attributes (private_m) as Inputs and builds cryptographic material for credential proof (Θ) as Outputs.

- **API Overview:** Our extended Coconut Credential Scheme defines the following high-level APIs for its operation:

  - **BlindSign Request Generation:** Users can generate a blind signing request by providing their private and public attributes as inputs:

$$\text{Coconut}_{\text{BlindSignReq}}(\text{Attr}_{\text{priv}}, \text{Attr}_{\text{pub}}) \rightarrow \text{req},$$

    where $\text{req} = (\lambda, \text{public}_m)$, λ represents the commitments, and public_m is the optional array of public attributes.
  - **BlindSignature by Issuer:** Issuers generate a blinded signature in response to the user's request:

$$\text{Coconut}_{\text{BlindSign}}(\text{req}) \rightarrow (\text{vk}_i, \sigma_i),$$

    where vk_i is the Issuer's verification key, and σ_i is the blinded credential.
  - **Aggregated Credential Generation:** Multiple credentials can be aggregated into a single credential, and multiple verification keys into a single aggregated key:

$$\text{Coconut}_{\text{Aggr}}(\text{sigs}) \rightarrow \sigma, \quad \text{Coconut}_{\text{Aggr}}(\text{vks}) \rightarrow \text{aggr\_vk},$$

where σ is the aggregated credential, and aggr_vk is the aggregated verification key.

- **ProveCredential:** Users can generate novel cryptographic proofs from an aggregated credential and verification key after re-randomizing the aggregated credential:

$$\text{Coconut}_{\text{Randomize}}(\sigma) \rightarrow \sigma'$$

$$\text{Coconut}_{\text{Prove}}(\text{aggr\_vk}, \sigma') \rightarrow \theta,$$

where θ is the proof.

- **VerifyCredential:** Verifiers can check the validity of a credential by verifying the proof against the aggregated verification key:

$$\text{Coconut}_{\text{Verify}}(\text{aggr\_vk}, \theta) \rightarrow \text{ret},$$

where ret is a boolean indicating success (true) or failure (false).

- **SelfSign:** Users can perform self-signature by taking public/private attributes and a seed for the private key, and generate a proof of credential, the credential itself, and the verification key:

$$\text{Coconut}_{\text{SelfSign}}(\text{Attr}, \text{Key}_{\text{signing}}) \rightarrow (\theta, \sigma, \text{vk}),$$

where θ is the proof of credential, σ is the credential, and vk is the corresponding verification key.

- **Properties of API:**
  - **Full Consensus Requirement:** Unlike the threshold-based approach, Unconditional Privacy requires all designated issuers to participate in the issuance process. This ensures maximum privacy by eliminating any threshold or subset participation.
  - **Asynchrony:** The issuance authorities generate their signing keys asynchronously using KeyGen. This differs from the threshold variant, which relies on a trusted third party for key generation (using TTPKeyGen[11]).
  - **Deterministic Liveness:** Liveness is guaranteed as long as all issuers remain functional, without reliance on weak synchrony assumptions or subsets of authorities.
  - **Blindness, Unlinkability and Unforgeability:** These properties are directly inherited from Sonnino et. al.[11], as below theorem restating:

    Theorem 1 (Sonnino et al., 2020) *Assuming LRSW, XDH, and the existence of random oracles, Coconut is a secure threshold credentials scheme, meaning it satisfies unforgeability (as long as fewer than t authorities collude), blindness, and unlinkability.*

– **Efficiency Constraints:** While communication and computational complexities scale with n, the number of issuers, $\mathcal{O}(n)$, in the first stages of requesting blind signatures and issueing them, the consequent computational complexity relies on short and efficient credentials and zero-knowledge proof-s.[11] Since Proving and Verification involves single aggregated credential, these procedures are computationally as well as communicatively $\mathcal{O}(1)$, irrespective of number of Issuers.[11]

Security Sketch

Orthogonality & Full-Consensus Authentication

The COCO Authentication Protocol guarantees the orthogonality of authentication identities through the integration of blinding techniques, rate-limited operations, and a multi-authenticator consensus mechanism. Specifically, the *Common Core* (Algorithm 1) constructs authentication identities that are unlinkable to both real identifiers (e.g., secrets, biometrics) and virtual public identifiers (e.g., usernames, user IDs). This orthogonality is achieved by applying a series of cryptographic transformations, which decouple authentication identities from their underlying sources.

The protocol hashes real identifiers and virtual public identifiers with additional salts and pepper to create authentication self-signing keys, and subsequently applies Oblivious Pseudorandom Functions (OPRFs) to these values. The final authentication identities must be necessarily derived for all authentication attempts by combining the OPRF evaluations from all Authenticators along with the respective self-signing keys, which ensures that these identities are not only distinct (and constant) for each Authenticator but also only derivable through full consensus among all Authenticators. This mechanism ensures that each authenticator's contribution to the authentication identity remains distinct, thus preventing the possibility of collusion or leakage of sensitive information through any single Authenticator.

Additionally, the Coconut Credential Scheme reinforces this security model by providing cryptographic guarantees of unlinkability, unforgeability, and blindness within rate-limiting operations. These properties ensure that OPRF evaluation request-tokens cannot be forged or traced back to their original sources unless all Authenticators are compromised. COCO's OPRF implementation is based on RFC9497's 2HashDH PRF, which is susciptible to static Diffie-Hellman Attacks if the OPRF can be directly queried by everyone.[7] [6][2] To protect our OPRF Protocol against static Diffie-Hellman attacks and further against possible pre-image attacks, rate-limiting queries to OPRF using time-based access tokens is necessary.

Therefore, the orthogonality of authentication identities is underpinned by the robust security guarantees offered by OPRFs and the Coconut Credential Scheme, which jointly prevent the re-identification of users across different sessions or authenticators.

Unlinkable/Zero Knowledge Authentication

The *Common Core* (Algorithm 1) also employs the Coconut Credential Scheme to enable a novel form of self-signature based unlinkable authentication. This process begins with the generation of self-signatures using the self-signing keys (generated as aforementioned by hashing real and virtual identifiers). The corresponding signatures are then used to generate re-randomizable, non-interactive zero-knowledge proofs (ZKPs) that attest to the validity of the signatures without revealing any underlying information. The verification keys for the proof-s along with the orthogonal authentication identities are stored by the Authenticators, while the re-randomized proofs are presented during each authentication attempt.

The key aspect of this design is that the Authenticators are only privy to the verification key and the unlinkable, re-randomized ZKPs, which they cannot use to trace the user's real or virtual identity or to correlate proof-s of ownership to a particular self-signature, or self-signing key among multiple authentication attempts. The re-randomized ZKPs can further provide security against replay attacks if the Authenticators black-list a proof after it has been shown. Consequently, the unlinkability of authentication identities, as well as the unlinkability between distinct authentication sessions, is ensured by the cryptographic properties of the Coconut Credential Scheme, specifically its resilience to linking and traceability.

Thus, the unlinkability of both the authentication identity and the authentication attempts is guaranteed by the inherent soundness and cryptographic properties of the Coconut Credential Scheme, which upholds the principles of zero-knowledge and secure user verification while preventing unauthorized re-identification.

Further Security

To further strengthen the security, the COCO Client utilizes the memory-hard hash $\mathcal{H}$ (modeled as a random oracle) to defend against offline brute-force attacks. This ensures that even if an attacker gains access to the hashed authentication identifiers or the self-signing keys, they are computationally difficult to reverse.

In addition, during the account creation phase, the COCO Client must present a Coconut blind credential proof as a registration access token to each of the Authenticators. This mechanism prevents virtual identifier collisions, as it guarantees that the registration process is validated through a unique access token per virtual identifier in a Verifier. This further secures the system by making it resistant to conflicts that could arise from multiple entities attempting to register the same virtual identifier.

Further, COCO uses Coconut Credential Scheme based re-randomized credentials prove authentication to Verifier, which prevents the Verifier from the knowledge of Authenticators involved in an authentication.

The Protocol

Symbol	Description
n	Number of Authenticators chosen by Client.
s	User's real identity, such as a secret, or biometric.
vid	User's mutable virtual public identity, such as a username.
uid	User's immutable virtual public identity, such as user ID.
salt_i	A random value sampled from $\{0,1\}^{256}$, associated with the user's virtual identity, for i-th Authenticator.
pepper	A random value sampled from $\{0,1\}^{256}$, shared across all users' virtual identities associated with a Verifier.
rid_i	User's authentication self-signing key for i-th Authenticator.
X_i	User's private input for OPRF query with i-th Authenticator.
Y_i	The i-th Authenticator's blinded OPRF output.
rid'_i	The i-th Authenticator's unblinded OPRF output.
id_i	User's authentication identity for i-th Authenticator.
k_i	The i-th Authenticator's OPRF Key.
$\text{req}_{registration}$	The access-token request for beginning registration.
registration-token-validity	The validity of the registration access-token.
σ_{register}	The registration access-token.
θ_{register}	The proof of possession of σ_{register}.
$\text{vk}_{\text{register}}$	The verification key for θ_{register}.
req_{oprf}	The access-token request for accessing the OPRF evaluators.
oprf-token-validity	The validity of access-token for accessing the OPRF evaluators.
σ^i_{oprf}	The i-th Authenticator's access-token for accessing the OPRF evaluators.
$\text{vk}^i_{\text{oprf}}$	The i-th Authenticator's oprf access-token verification key for σ^i_{oprf}.
σ_{oprf}	The consolidated access-token for accessing the OPRF evaluators.
θ_{oprf}	The proof of possession of σ_{oprf}.
vk_{oprf}	The consolidated verification key for θ_{oprf}.
$\sigma^i_{\text{rid}_i}$	The self-signature generated using the user's self-signing key (rid_i).
$\theta^i_{\text{rid}_i}$	The proof of possession of $\sigma^i_{\text{rid}_i}$.
$\text{vk}^i_{\text{rid}_i}$	The verification key for $\theta^i_{\text{rid}_i}$.
req_{auth}	The access-token request to generate proof of authentication.
auth-token-validity	The validity of access-token to generate proof of authentication.
σ^i_{auth}	The i-th Authenticator's access-token to generate proof of authentication.
$\text{vk}^i_{\text{auth}}$	The i-th Authenticator's verification key for σ^i_{auth}.
σ_{auth}	The consolidated access-token to generate proof of authentication by all Authenticators.
θ_{auth}	The proof of possession of σ_{auth}.
vk_{auth}	The consolidated verification key for θ_{auth}.
σ'_{auth}	The re-randomized consolidated σ_{auth}.
θ'_{auth}	The re-randomized proof of possession of σ'_{auth}.

Table 1: Notation used in the protocol.

Definition

COCO Authentication Protocol can be defined as the quadruple,

$$(\text{Client}, \text{Authenticators}, \text{Verifier}, \text{GlobalDB})$$

where:

- **Client:** is any authenticatee, an entity requesting authentication to access a resource. There can be any number of Clients, and each Client can choose a subset of Authenticators for their personal authentication scheme.
- **Authenticators:** are a set of multiple servers that perform distributed authentication.
- **Verifier:** is a verifier of distributed authentication, also, any resource server hosting the resource the Authenticatee wants to access. For strict privacy goals, the system assumes only one Verifier per set of Authenticators as a preventive measure against possible user identity collision attacks.
- **GlobalDB:** a global database known to Authenticators and Verifier, used for storing and retrieving verification keys.

Initialization

Each of the Client, Authenticators and the Verifier are initialized with their own local database, we refer to them as LocalDB.

Authenticator: Each Authenticator secretly and independently chooses an OPRF key. We denote Authenticator i's OPRF key as k_i. Each Authenticator initializes an instance of the server-end OPRF protocol supporting $\mathcal{F}_{OPRF}$ API and an instance of Extended Coconut Credential Scheme for Unconditional Privacy supporting Coconut$_{BlindSign}$, Coconut$_{Aggr}$, and Coconut$_{Verify}$ API-s. The Authenticators mutually decide upon values for validity for a registration token: reg-token-validity, and validity for an oprf access token : oprf-token-validity, and let them known publicly.

Verifier: The Verifier independently chooses a pepper and initializes an instance of Extended Coconut Credential Scheme for Unconditional Privacy supporting the Coconut$_{BlindSign}$, Coconut$_{Aggr}$ and Coconut$_{Verify}$ API-s. The Verifier decides upon value for authentication token validity, auth-token-validity, and lets it known publicly.

Client: The Client begins by taking a user's real identity s, such as a password or biometric, their mutable virtual public identity vid, such as a username. The Client decides upon a number of Authenticators n that it wants to use for COCO Authentication. Further, the Client initializes an instance

of a memory-hard password hashing function $\mathcal{H}$, an instance of OPRF protocol supporting $\mathcal{OPRF}_{Blind}$ and $\mathcal{OPRF}_{Unblind}$ API-s, and an instance of Extended Coconut Credential Scheme for Unconditional Privacy supporting $\text{Coconut}_{BlindSignRequest}$, Coconut_{Aggr}, Coconut_{Prove} and $\text{Coconut}_{SelfSign}$.

GlobalDB: Finally, the common database GlobalDB is initialized and Verifier and Authenticator are granted read/write access to it.

Account Creation

First the Client invokes *Registration Initialization* (Algorithm 2) to compute necessary variables to begin registration procedure. The algorithm is executed with just vid as input. Upon completion, the algorithm returns the following outputs:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^{n}, \text{vk}_{register}, \sigma_{register}, \theta_{register})$$

Next the Client invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for authentication. The *Common Core* (Algorithm 1) takes its inspiration from Ryan Little et. al.[10], however, with significant specializations and modifications. The algorithm is executed with the following inputs:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^{n}, \text{s})$$

Upon completion, the algorithm returns the following outputs:

$$(\{\text{id}_i\}_{i=1}^{n}, \{\theta^i_{rid_i}\}_{i=1}^{n}, \{\text{vk}^i_{rid_i}\}_{i=1}^{n}, \text{req}_{auth})$$

Finally, the Client invokes the *Registration Finalization* (Algorithm 4) to finalize the registration procedure. The algorithm is executed with the following inputs:

$$(\text{uid}, \text{vid}, \{\text{salt}_i\}_{i=1}^{n}, \{\text{id}_i\}_{i=1}^{n}, \{\theta^i_{rid_i}\}_{i=1}^{n}, \{\text{vk}^i_{rid_i}\}_{i=1}^{n}, \text{vk}_{register}, \theta_{register}, \text{req}_{auth})$$

Upon completion, the algorithm returns the following outputs:

$$(\text{vk}_{auth}, \sigma'_{auth})$$

Now, the Client may use vk_{auth} and σ'_{auth} for token authentications.

Authentication

The Client invokes *Authentication Initialization* (Algorithm 3) to compute necessary variables to begin authentication procedure. The algorithm is executed with just vid as input. Upon completion, the algorithm returns the following outputs:

$$\text{uid}, \{\text{salt}_i\}_{i=1}^{n}, \text{pepper}$$

Next the Client invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for authentication. The algorithm is executed with the following inputs:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^{n}, \text{s})$$

Upon completion, the algorithm returns the following outputs:

$$(\{\mathrm{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\mathrm{vk}^i_{rid_i}\}_{i=1}^n, \mathrm{req}_{auth})$$

Finally, the Client invokes the *Authentication Finalization* (Algorithm 6) to finalize the authentication procedure. The algorithm is executed with the following inputs:

$$\mathrm{uid}, \mathrm{vid}, \{\mathrm{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \mathrm{req}_{auth}$$

Upon completion, the algorithm returns the following outputs:

$$(\mathrm{vk}_{auth}, \sigma'_{auth})$$

Now, the Client may use vk_{auth} and σ'_{auth} for token authentications.

Token Authentication

The Client invokes the *Token Authentication* (Algorithm 5) to perform token-based authentication. The algorithm is executed with the following inputs:

$$\mathrm{uid}, \mathrm{vid}, \mathrm{vk}_{auth}, \sigma'_{auth}$$

Upon completion, the algorithms returns the following output:

$$\sigma'_{auth}$$

Now, the Client may use the already known vk_{auth} and the new output σ'_{auth} for subsequent token authentications.

2FA Creation

The Client invokes *Authentication Initialization* (Algorithm 3) to compute necessary variables to begin authentication procedure. The algorithm is executed with just vid as input. Upon completion, the algorithm returns the following outputs:

$$\mathrm{uid}, \{\mathrm{salt}_i\}_{i=1}^n, \mathrm{pepper}$$

Next the Client invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for authentication. The algorithm is executed with the following inputs:

$$(\mathrm{uid}, \mathrm{pepper}, \{\mathrm{salt}_i\}_{i=1}^n, \mathrm{s})$$

Upon completion, the algorithm returns the following outputs:

$$(\{\mathrm{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\mathrm{vk}^i_{rid_i}\}_{i=1}^n, \mathrm{req}_{auth})$$

Now, the Client takes the second real identity, let us call it s', and invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for 2FA Creation. This time the inputs are:

$$(\mathrm{uid}, \mathrm{pepper}, \{\mathrm{salt}_i\}_{i=1}^n, \mathrm{s}')$$

Upon completion, the algorithm returns the same variables as before, but with updated values. For clarity in subsequent steps, we rename them as follows:

$$(\{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \{\text{vk}^i_{nrid_i}\}_{i=1}^n, \text{req}'_{auth})$$

The renamed variables are used only in this context to distinguish the updated values. They retain the same roles and meanings as their original counterparts.

Finally, the Client invokes *2nd Factor Registration Finalization*(Algorithm 10) with the following inputs:

$$(\text{uid}, \text{vid}, \{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \{\text{vk}^i_{nrid_i}\}_{i=1}^n, \text{req}'_{auth})$$

Upon completion, the algorithm returns the following outputs:

$$(\text{vk}_{auth}, \sigma'_{auth})$$

Now, the Client may use vk_{auth} and σ'_{auth} for token authentications. The protocol for 2FA Creation can be generalized to support the creation of multiple authentication factors, limited only by the implementation's capacity. This necessitates the inclusion of functionality for managing these factors, including their deletion. Such capabilities can be integrated with minimal modifications to the existing protocols.

Real Identity Reset

The Client invokes *Authentication Initialization* (Algorithm 3) to compute necessary variables to begin authentication procedure. The algorithm is executed with just vid as input. Upon completion, the algorithm returns the following outputs:

$$\text{uid}, \{\text{salt}_i\}_{i=1}^n, \text{pepper}$$

Next the Client invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for authentication. The algorithm is executed with the following inputs:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^n, \text{s})$$

Upon completion, the algorithm returns the following outputs:

$$(\{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{vk}^i_{rid_i}\}_{i=1}^n, \text{req}_{auth})$$

Now, the Client takes the new real identity, let us call it s', and generates as many new salts $\{\text{nsalt}_i\}_{i=1}^n$. Then it invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for 2FA Creation. This time the inputs are:

$$(\text{uid}, \text{pepper}, \{\text{nsalt}_i\}_{i=1}^n, \text{s}')$$

Upon completion, the algorithm returns the same variables as before, but with updated values. For clarity in subsequent steps, we rename them as follows:

$$(\{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \{\text{vk}^i_{nrid_i}\}_{i=1}^n, \text{req}'_{auth})$$

The renamed variables are used only in this context to distinguish the updated values. They retain the same roles and meanings as their original counterparts.

Finally, the Client invokes *Real Identity Reset* (Algorithm 8) with the following inputs:

$$(\text{uid}, \text{vid}, \{\text{nsalt}_i\}_{i=1}^n, \{\text{id}_i\}_{i=1}^n, \{\theta_{rid_i}^i\}_{i=1}^n,$$
$$\{\text{nid}_i\}_{i=1}^n, \{\theta_{nrid_i}^i\}_{i=1}^n, \{\text{vk}_{nrid_i}^i\}_{i=1}^n, \text{req}'_{auth})$$

Upon completion, the algorithm returns the following outputs:

$$(\text{vk}_{auth}, \sigma'_{auth})$$

Now, the Client may use vk_{auth} and σ'_{auth} for token authentications.

Account Deletion

The Client invokes *Authentication Initialization* (Algorithm 3) to compute necessary variables to begin authentication procedure. The algorithm is executed with just vid as input. Upon completion, the algorithm returns the following outputs:

$$\text{uid}, \{\text{salt}_i\}_{i=1}^n, \text{pepper}$$

Next the Client invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for authentication. The algorithm is executed with the following inputs:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^n, \text{s})$$

Upon completion, the algorithm returns the following outputs:

$$(\{\text{id}_i\}_{i=1}^n, \{\theta_{rid_i}^i\}_{i=1}^n, \{\text{vk}_{rid_i}^i\}_{i=1}^n, \text{req}_{auth})$$

Now, the Client takes their second factor/second real identity, let us call it s', and invokes the *Common Core* (Algorithm 1) to compute the necessary outputs for 2FA. At present COCO requires the user to mandatorily have a 2FA created to be able to perform deletion. However, this can be changed with minor tweaks in the algorithm itself. This time the inputs to *Common Core* are:

$$(\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^n, \text{s}')$$

Upon completion, the algorithm returns the same variables as before, but with updated values. For clarity in subsequent steps, we rename them as follows:

$$(\{\text{nid}_i\}_{i=1}^n, \{\theta_{nrid_i}^i\}_{i=1}^n, \{\text{vk}_{nrid_i}^i\}_{i=1}^n, \text{req}'_{auth})$$

The renamed variables are used only in this context to distinguish the updated values. They retain the same roles and meanings as their original counterparts.

Finally, the Client invokes *Deletion Finalization*(Algorithm 9) with the following inputs:

$$(\text{uid}, \text{vid}, \{\text{id}_i\}_{i=1}^n, \{\theta_{rid_i}^i\}_{i=1}^n, \{\text{nid}_i\}_{i=1}^n, \{\theta_{nrid_i}^i\}_{i=1}^n, \text{req}'_{auth})$$

Upon completion, the algorithm returns **Deletion Successful**.

Virtual Public Identity Reset

The Client invokes the *Virtual Identity Reset* (Algorithm 7) to perform token-based virtual public identity reset. Let *nvid* be a replacement mutable virtual public identity for *vid*. Note that only mutable virtual identity *vid* can be reset at present with COCO. Hence, the algorithm is executed with the following inputs:

$$\text{uid}, \text{vid}, \text{nvid}, \text{vk}_{auth}, \sigma'_{auth}$$

Upon completion, the algorithms returns the following output:

$$\sigma'_{auth}$$

Now, the Client may use the already known vk_{auth} and the new output σ'_{auth} for subsequent token authentications with new *vid*.

Evaluation

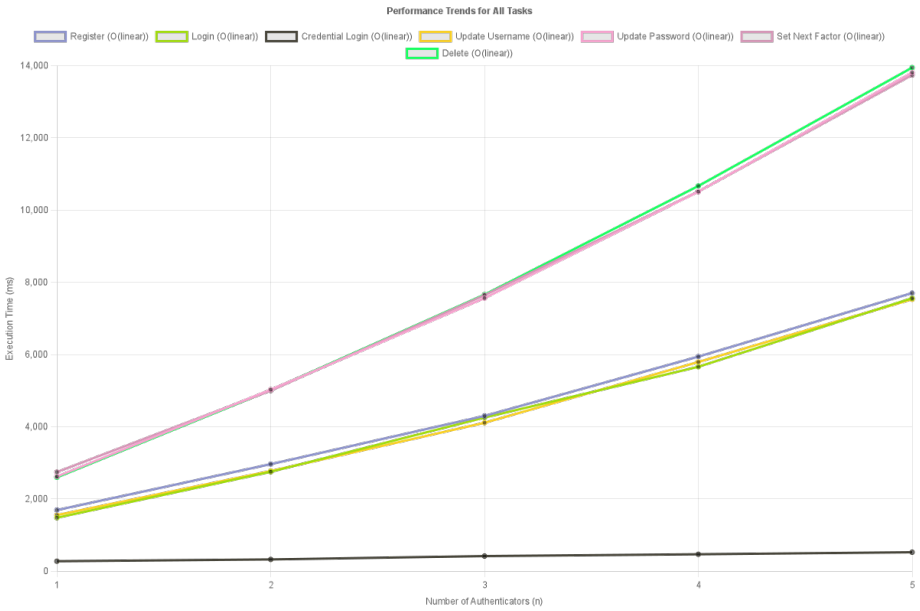

Figure 1: Performance trend of COCO protocol tasks. Execution times are plotted against the number of authenticators.

The performance evaluation of COCO protocol tasks was conducted in a resource-constrained environment using an ASUS Laptop M515DA. The system specifications included an AMD Ryzen 3 3250U processor (base speed: 2.60 GHz, 2 cores, 4 logical processors, L1 cache: 192 KB, L2 cache: 1.0 MB, L3 cache: 4.0 MB) and 8 GB of memory (hardware-reserved: 2 GB, speed:

2400 MT/s). A JavaScript implementation of an Identity Management System (IMS) based on the COCO protocol tasks was utilized, available in a Git repository[1]. This implementation relies on several libraries, including 'node-argon2'[2] for memory-hard hashing, a modified Rust-based Coconut library[3] for JavaScript compatibility, Cloudflare's TypeScript OPRF library[4], and 'uuid'[5] for RFC 9562-compliant immutable virtual public identity generation. The evaluation was constrained by JavaScript's single-threaded execution model and the processor's single-core performance.

The evaluation involved over 100 runs of common operations in the COCO protocols, such as account creation, authentication, and other identity management tasks. These tests excluded network overheads but included in-memory database operations for demonstration purposes. The memory-hard hash function was optimized with the following parameters: memory cost of 1024 KB, time cost of 2, parallelism of 1, and a fixed hash length of 256 bits to minimize time complexity overhead and focus on protocol performance. The effect of the number of authenticators (n) on performance was analyzed by incrementally increasing n from 1 to 5 (assuming an even more robust evaluation for future exploration) and measuring execution times for each task.

Task Name	$\mathcal{O}(T(n))$	$\sqrt{\sigma^2}$ (ms)
Account Creation	$\mathcal{O}(n)$	$\pm$89.30
Authentication	$\mathcal{O}(n)$	$\pm$49.01
Token Authentication	$\mathcal{O}(1)$	$\pm$91.00
	$\mathcal{O}(n)$	$\pm$60.94
Virtual Public Identity Reset	$\mathcal{O}(n)$	$\pm$69.34
Real Identity Reset	$\mathcal{O}(n)$	$\pm$90.10
2FA Creation	$\mathcal{O}(n)$	$\pm$100.55
Account Deletion	$\mathcal{O}(n)$	$\pm$98.59

Table 2: Performance metrics for COCO protocol tasks. $\mathcal{O}(T(n))$ denotes the time complexity of task performance, and $\sqrt{\sigma^2}$ represents the error margin in milliseconds, calculated as the standard deviation of observed execution times.

Table 2 presents the performance metrics, including time complexity ($\mathcal{O}(T(n))$) and error margins derived from observed data. The time complexity was determined by fitting execution times to various models (constant, linear, quadratic, cubic), with the model exhibiting the lowest variance selected. Most tasks demonstrated linear time complexity ($\mathcal{O}(n)$), where execution times increased proportionally with the number of authenticators. Token Authentication showed near-equivalence between constant and linear time complexities, warranting further analysis for optimization.

[1] `https://github.com/14morpheus14/coco`
[2] `https://www.npmjs.com/package/argon2`
[3] `https://github.com/nymtech/coconut`
[4] `https://www.npmjs.com/package/@cloudflare/voprf-ts`
[5] `https://www.npmjs.com/package/uuid`

Figure 1 illustrates the performance trends. Notably, the performance of the *Virtual Public Identity Reset* task was similar to that of the *Authentication* task. This similarity arises because the reset process included a complete authentication step followed by an update. Token-based authentication could significantly enhance the performance of the virtual public identity reset, representing a potential avenue for optimization.

Limitations

The COCO Authentication protocol inherits several limitations from its underlying schemes, including the Coconut Credential Scheme and Oblivious Pseudorandom Function (OPRF) implementations. Further limitations in the overall protocol also presents challenges that require further research to address effectively. Addressing these limitations to strengthen COCO's security and resilience against diverse attack vectors remains an open area for future exploration.

A critical concern is COCO's reliance on the OPRF server keys managed by the Authenticators. If an OPRF server loses its key, all authentication identities associated with it become invalid, resulting in users being permanently locked out of their accounts. This scenario underscores the need for a privacy-preserving mechanism for key rotation or recovery, which is not currently supported within the protocol.

Additionally, COCO's full-consensus mechanism raises concerns regarding its resilience to faults. While the mechanism ensures security, a more graceful and fault-tolerant approach to handling failures would significantly enhance its reliability.

Another significant limitation is the absence of an inherent account recovery mechanism. If users forget their credentials, COCO provides no native method to recover their accounts in a privacy preserving manner. Although the protocol's second-factor authentication offers a partial solution by enabling recovery in cases where users forget their real identifiers, traditional recovery mechanisms, such as email or SMS, are intentionally excluded to protect user privacy. Moreover, COCO presently refrains from implementing Secure Multiparty Computation (MPC)-based TLSv1.3 email or SMS recovery methods, as explored in MPC-Auth by Sijun Tan et. al. [12], due to concerns about metadata leakage and the potential logging of personally identifiable information (PII) by malicious Authenticators through final SMTP or SMS payloads.

To address potential future needs, COCO includes a repository of secure circuits[6] designed to facilitate the development of recovery mechanisms, such as MPC-Auth-based solutions, should they be deemed necessary in later iterations of the protocol. This repository serves as an informal foundation for incorporating recovery options without compromising COCO's core privacy guarantees.

[6] `https://github.com/14morpheus14/circuits`

References

[1] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 41–55, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[2] Daniel R. L. Brown and Robert P. Gallant. The static diffie-hellman problem. Cryptology ePrint Archive, Paper 2004/306, 2004.

[3] Jan Camenisch and Anja Lehmann. Privacy-preserving user-auditable pseudonym systems. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 269–284, 2017.

[4] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145. IEEE, 2001.

[5] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudorandom functions. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 625–646. IEEE, 2022.

[6] Jung Hee Cheon. Security analysis of the strong diffie-hellman problem. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 1–11, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups. RFC 9497, December 2023.

[8] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 416–431, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 233–253, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[10] Ryan Little, Lucy Qin, and Mayank Varia. Secure account recovery for a privacy-preserving web service. Cryptology ePrint Archive, Paper 2024/962, 2024.

[11] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers, 2020.

[12] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multi-factor authentication for distributed-trust systems. Cryptology ePrint Archive, Paper 2021/342, 2021.

Algorithms

Algorithm 1 Common Core

Require: $\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^{n}, \text{s}$
Ensure: $\{\text{id}_i\}_{i=1}^{n}, \{\theta^i_{rid_i}\}_{i=1}^{n}, \{\text{vk}^i_{rid_i}\}_{i=1}^{n}, \text{req}_{auth}$

for $i = 1$ to n **do**
    Client computes real identities:

$$\text{rid}_i \leftarrow \mathcal{H}(\text{uid} \| \mathcal{H}(\text{s} \| \text{salt}_i) \| \text{pepper})$$

    Client blinds the real identities:

$$X_i \leftarrow \mathcal{OPRF}_{Blind}(\text{rid}_i)$$

end for
Client generates an OPRF access request:

$$\text{req}_{oprf} \leftarrow \text{Coconut}_{BlindSignRequest}(\text{uid}, \text{oprf-token-validity})$$

for $i = 1$ to n **do**
    Client sends (req_{oprf}) to i-th Authenticator
    The i-th Authenticator computes:

$$\sigma^i_{oprf}, \text{vk}^i_{oprf} \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{oprf})$$

    The Authenticator i stores vk^i_{oprf} in GlobalDB,
    The Authenticator i sends (σ^i_{oprf}, vk^i_{oprf}) to Client.
end for
After receiving $\{\sigma^i_{oprf}\}_{i=1}^{n}$ and $\{\text{vk}^i_{oprf}\}_{i=1}^{n}$), Client computes proof of OPRF access:

$$\sigma_{oprf}, \text{vk}_{oprf} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{oprf}\}_{i=1}^{n}, \{\sigma^i_{oprf}\}_{i=1}^{n})$$

$$\theta_{oprf} \leftarrow \text{Coconut}_{Prove}(\sigma_{oprf}, \text{vk}_{oprf})$$

for $i = 1$ to n **do**
    Client sends $(\theta_{oprf}, \{\text{vk}^i_{oprf}\}_{i=1}^{n}, X_i)$ to i-th Authenticator.
    Authenticator i retrives all $\{\text{vk}^i_{oprf}\}_{i=1}^{n}$ from GlobalDB, verifies θ_{oprf} and performs OPRF:

$$\text{vk}_{oprf} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{oprf}\}_{i=1}^{n})$$

    if $\text{Coconut}_{Verify}(\text{vk}_{oprf}, \theta_{oprf})$ **then**

$$Y_i \leftarrow \mathcal{F}_{OPRF}(X_i)$$

    end if
    The Client receives Y_i and unblinds it:

$$\text{rid}'_i \leftarrow \mathcal{OPRF}_{Unblind}(Y_i)$$

    Client derives authentication identity:

$$\text{id}_i \leftarrow \mathcal{H}(\text{rid}_i \| \mathcal{H}(\text{rid}'_1 \| \ldots \| \text{rid}'_n \| \text{salt}_i))$$

    Client self-signs the authentication identity using real identity and derives $\theta^i_{rid_i}$, $\sigma^i_{rid_i}$, and $\text{vk}^i_{rid_i}$:

$$\theta^i_{rid_i}, \sigma^i_{rid_i}, \text{vk}^i_{rid_i} \leftarrow \text{Coconut}_{SelfSign}(\text{id}_i, \text{rid}_i)$$

end for
Client creates req_{auth}:

$$\text{req}_{auth} \leftarrow \text{Coconut}_{BlindSignRequest}(\text{uid}, \text{auth-token-validity})$$

Algorithm 2 Registration Initialization

Require: vid
Ensure: $\text{uid}, \text{pepper}, \{\text{salt}_i\}_{i=1}^{n}, \text{vk}_{register}, \sigma_{register}, \theta_{register}$

1: Client creates a registration access request:

$$\text{req}_{register} \leftarrow \text{Coconut}_{BlindSignRequest}(\text{vid}, \text{registration-token-validity})$$

2: Client sends $(\text{vid}, \text{req}_{register})$ to the Verifier.
3: Verifier checks the availability of vid in its local database.
4: **if** vid is available **then**
5: Verifier creates a blinded signature:

$$(\text{vk}_{register}, \sigma_{register}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{register})$$

6: **end if**
7: Verifier generates a unique user identifier:

$$\text{uid} \leftarrow \text{GenerateUID}()$$

8: Verifier outputs the following to the Client:

$$(\text{uid}, \text{vk}_{register}, \sigma_{register}, \text{pepper})$$

9: Verifier stores $\text{vk}_{register}$ in the GlobalDB for retrieval by Authenticators.
10: Client computes proof $\theta_{register}$ for $\sigma_{register}$:

$$\theta_{register} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{register}, \sigma_{register})$$

11: Client generates per-authenticator salts $\{\text{salt}_i\}_{i=1}^{n}$:

$$\text{salt}_i \leftarrow \text{GenerateRandomSalt}(), \forall i \in [1, n]$$

Algorithm 3 Authentication Initialization

Require: vid
Ensure: $\text{uid}, \{\text{salt}_i\}_{i=1}^{n}, \text{pepper}$

1: Client sends (vid) to the Verifier.
2: Verifier checks the existence of vid in its local database.
3: **if** vid **then**
4: Verifier retrieves from its LocalDB and outputs the following to the Client:

$$(\text{uid}, \{\text{salt}_i\}_{i=1}^{n}, \text{pepper})$$

5: **end if**

Algorithm 4 Registration Finalization

Require: $\text{uid}, \text{vid}, \{\text{salt}_i\}_{i=1}^n, \{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{vk}^i_{rid_i}\}_{i=1}^n, \text{vk}_{register}, \theta_{register}, \text{req}_{auth}$
Ensure: $\text{vk}_{auth}, \text{Coconut}_{Randomize}(\sigma_{auth})$

for $i = 1$ to n **do**
  Client sends $(\text{id}_i, \theta^i_{rid_i}, \text{vk}^i_{rid_i}, \text{vk}_{register}, \theta_{register}, \text{req}_{auth})$ to i-th Authenticator.
  The i-th Authenticator retrieves $\text{vk}_{register}$ from GlobalDB and verifies $\theta_{register}$:
  if $\text{Coconut}_{Verify}(\text{vk}_{register}, \theta_{register})$ **then**
    Authenticator i verifies the proof $\theta^i_{rid_i}$:
    if $\text{Coconut}_{Verify}(\text{vk}^i_{rid_i}, \theta^i_{rid_i})$ **then**
      Authenticator i stores $(\text{id}_i, \text{vk}^i_{rid_i})$ in its LocalDB
      Authenticator i blind signs authentication token:

$$(\text{vk}^i_{auth}, \sigma^i_{auth}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{auth})$$

      Authenticator i stores vk^i_{auth} to GlobalDB.
      Authenticator i sends $(\text{vk}^i_{auth}, \sigma^i_{auth})$ to Client.
    end if
  end if
end for
Client receives all $\{\text{vk}^i_{auth}\}_{i=1}^n$, $\{\sigma^i_{auth}\}_{i=1}^n$, aggregates them and creates a proof:

$$\text{vk}_{auth}, \sigma_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n, \{\sigma^i_{auth}\}_{i=1}^n)$$

$$\theta_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma_{auth})$$

Client sends $(\text{uid}, \text{vid}, \{\text{salt}_i\}_{i=1}^n, \theta_{auth}, \{\text{vk}^i_{auth}\}_{i=1}^n)$ to the Verifier.
Verifier retrieves all $\{\text{vk}^i_{auth}\}_{i=1}^n$ from GlobalDB, aggregates them and verifies θ_{auth}:

$$\text{vk}_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n)$$

if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta_{auth})$ **then**
  Verifier stores (uid, vid, vk_{auth}, $\{\text{salt}_i\}_{i=1}^n$) in its LocalDB
  Registration Successful
end if

Algorithm 5 Token Authentication

Require: $\text{uid}, \text{vid}, \text{vk}_{auth}, \sigma'_{auth}$
Ensure: $\text{Coconut}_{Randomize}(\sigma'_{auth})$

Client takes the randomized authentication token σ'_{auth}, and creates a new proof:

$$\theta'_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma'_{auth})$$

Client sends $(\text{uid}, \text{vid}, \theta'_{auth}, \text{vk}_{auth})$ to the Verifier.
if uid & vid exists in its LocalDB **then**
  Verifier retrieves vk_{auth} from its LocalDB and verifies θ'_{auth}:
  if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta'_{auth})$ **then**
    Token Authentication Successful
  end if
end if

Algorithm 6 Authentication Finalization

Require: $\text{uid}, \text{vid}, \{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \text{req}_{auth}$
Ensure: $\text{vk}_{auth}, \text{Coconut}_{Randomize}(\sigma_{auth})$

for $i = 1$ to n **do**
  Client sends $(\text{id}_i, \theta^i_{rid_i}, \text{req}_{auth})$ to i-th Authenticator.
  if id_i exists in its LocalDB **then**
    Authenticator i retrieves $\text{vk}^i_{rid_i}$ from its LocalDB, and verifies proof $\theta^i_{rid_i}$:
    if $\text{Coconut}_{Verify}(\text{vk}^i_{rid_i}, \theta^i_{rid_i})$ **then**
      Authenticator i blind signs authentication token:

$$(\text{vk}^i_{auth}, \sigma^i_{auth}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{auth})$$

      Authenticator i stores vk^i_{auth} to GlobalDB.
      Authenticator i sends $(\text{vk}^i_{auth}, \sigma^i_{auth})$ to Client.
    end if
  end if
end for
Client receives all $\{\text{vk}^i_{auth}\}_{i=1}^n$, $\{\sigma^i_{auth}\}_{i=1}^n$, aggregates them and creates a proof:

$$\text{vk}_{auth}, \sigma_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n, \{\sigma^i_{auth}\}_{i=1}^n)$$

$$\theta_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma_{auth})$$

Client sends $(\text{uid}, \text{vid}, \theta_{auth}, \{\text{vk}^i_{auth}\}_{i=1}^n)$ to the Verifier.
Verifier retrieves all $\{\text{vk}^i_{auth}\}_{i=1}^n$ from GlobalDB, aggregates them and verifies θ_{auth}:

$$\text{vk}_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n)$$

if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta_{auth})$ **then**
  if uid & vid exists in its LocalDB **then**
    Verifier updates (vk_{auth}) in the LocalDB
    Authentication Successful
  end if
end if

Algorithm 7 Virtual Identity Reset

Require: $\text{uid}, \text{vid}, \text{nvid}, \text{vk}_{auth}, \sigma'_{auth}$
Ensure: $\text{Coconut}_{Randomize}(\sigma'_{auth})$

Client takes the randomized authentication token σ'_{auth}, and creates a new proof:

$$\theta'_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma'_{auth})$$

Client sends $(\text{uid}, \text{vid}, \text{nvid}, \theta'_{auth}, \text{vk}_{auth})$ to the Verifier.
if uid & vid exists in its LocalDB **then**
  Verifier retrieves vk_{auth} from its LocalDB and verifies θ'_{auth}:
  if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta'_{auth})$ **then**
    if nvid*isavailable* **then**
      Verifier updates (nvid) in the LocalDB
      Virtual Identity Reset Successful
    end if
  end if
end if

Algorithm 8 Real Identity Reset

Require: $\text{uid}, \text{vid}, \{\text{nsalt}_i\}_{i=1}^n$,
$\{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \{\text{vk}^i_{nrid_i}\}_{i=1}^n, \text{req}_{auth}$
Ensure: $\text{vk}_{auth}, \text{Coconut}_{Randomize}(\sigma_{auth})$

for $i = 1$ to n **do**
  Client sends $(\text{id}_i, \theta^i_{rid_i}, \text{nid}_i, \theta^i_{nrid_i}, \text{vk}^i_{nrid_i}, \text{req}_{auth})$ to i-th Authenticator.
  if id_i exists in its LocalDB **then**
    Authenticator i retrieves $\text{vk}^i_{rid_i}$ from its LocalDB, and verifies proof $\theta^i_{rid_i}$:
    if $\text{Coconut}_{Verify}(\text{vk}^i_{rid_i}, \theta^i_{rid_i})$ **then**
      Authenticator i verifies the proof $\theta^i_{nrid_i}$:
      if $\text{Coconut}_{Verify}(\text{vk}^i_{nrid_i}, \theta^i_{nrid_i})$ **then**
        Authenticator i stores $(\text{nid}_i, \text{vk}^i_{nrid_i})$ in its LocalDB.
        Authenticator i deletes $(\text{id}_i, \text{vk}^i_{rid_i})$ from its LocalDB.
        Authenticator i blind signs authentication token:

$$(\text{vk}^i_{auth}, \sigma^i_{auth}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{auth})$$

        Authenticator i stores vk^i_{auth} to GlobalDB
        Authenticator i sends $(\text{vk}^i_{auth}, \sigma^i_{auth})$ to Client.
      end if
    end if
  end if
end for
Client receives all $\{\text{vk}^i_{auth}\}_{i=1}^n$, $\{\sigma^i_{auth}\}_{i=1}^n$, aggregates them and creates a proof:

$$\text{vk}_{auth}, \sigma_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n, \{\sigma^i_{auth}\}_{i=1}^n)$$

$$\theta_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma_{auth})$$

Client sends $(\text{uid}, \text{vid}, \{\text{nsalt}_i\}_{i=1}^n, \theta_{auth}, \{\text{vk}^i_{auth}\}_{i=1}^n)$ to the Verifier.
Verifier retrieves all $\{\text{vk}^i_{auth}\}_{i=1}^n$ from GlobalDB, aggregates them and verifies θ_{auth}:

$$\text{vk}_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n)$$

if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta_{auth})$ **then**
  if uid & vid exists in its LocalDB **then**
    Verifier updates $(\{\text{nsalt}_i\}_{i=1}^n)$ in the LocalDB
    Real Identity Reset Successful
  end if
end if

Algorithm 9 Deletion Finalization

Require: $\text{uid}, \text{vid}, \{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \text{req}_{auth}$
Ensure: Deletion Successful

for $i = 1$ to n **do**
  Client sends $(\text{id}_i, \theta^i_{rid_i}, \text{nid}_i, \theta^i_{nrid_i}, \text{req}_{auth})$ to i-th Authenticator.
  if id_i exists in its LocalDB **then**
    Authenticator i retrieves $\text{vk}^i_{rid_i}$ from its LocalDB, and verifies proof $\theta^i_{rid_i}$:
    if $\text{Coconut}_{Verify}(\text{vk}^i_{rid_i}, \theta^i_{rid_i})$ **then**
      if nid_i exists in its LocalDB **then**
        Authenticator i retrieves $\text{vk}^i_{nrid_i}$ and verifies proof $\theta^i_{nrid_i}$:
        if $\text{Coconut}_{Verify}(\text{vk}^i_{nrid_i}, \theta^i_{nrid_i})$ **then**
          Authenticator i deletes $(\text{nid}_i, \text{vk}^i_{nrid_i})$ from its LocalDB.
          Authenticator i deletes $(\text{id}_i, \text{vk}^i_{rid_i})$ from its LocalDB.
          Authenticator i blind signs authentication token:

$$(\text{vk}^i_{auth}, \sigma^i_{auth}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{auth})$$

          Authenticator i stores vk^i_{auth} to GlobalDB
          Authenticator i sends $(\text{vk}^i_{auth}, \sigma^i_{auth})$ to Client.
        end if
      end if
    end if
  end if
end for
Client receives all $\{\text{vk}^i_{auth}\}_{i=1}^n$, $\{\sigma^i_{auth}\}_{i=1}^n$, aggregates them and creates a proof:

$$\text{vk}_{auth}, \sigma_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n, \{\sigma^i_{auth}\}_{i=1}^n)$$

$$\theta_{auth} \leftarrow \text{Coconut}_{Prove}(\text{vk}_{auth}, \sigma_{auth})$$

Client sends $(\text{uid}, \text{vid}, \theta_{auth}, \{\text{vk}^i_{auth}\}_{i=1}^n)$ to the Verifier.
Verifier retrieves all $\{\text{vk}^i_{auth}\}_{i=1}^n$ from GlobalDB, aggregates them and verifies θ_{auth}:

$$\text{vk}_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n)$$

if $\text{Coconut}_{Verify}(\text{vk}_{auth}, \theta_{auth})$ **then**
  if uid & vid exists in its LocalDB **then**
    Verifier deletes $(\text{uid}, \text{vid}, \text{vk}_{auth}, \{\text{salt}_i\}_{i=1}^n)$ from the LocalDB
    Deletion Successful
  end if
end if

Algorithm 10 2nd Factor Registration Finalization

Require: $\text{uid}, \text{vid}, \{\text{id}_i\}_{i=1}^n, \{\theta^i_{rid_i}\}_{i=1}^n, \{\text{nid}_i\}_{i=1}^n, \{\theta^i_{nrid_i}\}_{i=1}^n, \{\text{vk}^i_{nrid_i}\}_{i=1}^n, \text{req}_{auth}$
Ensure: $\text{vk}_{auth}, \text{Coconut}_{Randomize}(\sigma_{auth})$

for $i = 1$ to n **do**
  Client sends $(\text{id}_i, \theta^i_{rid_i}, \text{nid}_i, \theta^i_{nrid_i}, \text{vk}^i_{nrid_i}, \text{req}_{auth})$ to i-th Authenticator.
  if id_i exists in its LocalDB **then**
    Authenticator i retrieves $\text{vk}^i_{rid_i}$ from its LocalDB, and verifies proof $\theta^i_{rid_i}$:
    if $\text{Coconut}_{Verify}(\text{vk}^i_{rid_i}, \theta^i_{rid_i})$ **then**
      Authenticator i verifies the proof $\theta^i_{nrid_i}$:
      if $\text{Coconut}_{Verify}(\text{vk}^i_{nrid_i}, \theta^i_{nrid_i})$ **then**
        Authenticator i stores $(\text{nid}_i, \text{vk}^i_{nrid_i})$ in its LocalDB.
        Authenticator i blind signs authentication token:

$$(\text{vk}^i_{auth}, \sigma^i_{auth}) \leftarrow \text{Coconut}_{BlindSign}(\text{req}_{auth})$$

        Authenticator i stores vk^i_{auth} to GlobalDB.
        Authenticator i sends $(\text{vk}^i_{auth}, \sigma^i_{auth})$ to Client.
      end if
    end if
  end if
end for
Client receives all $\{\text{vk}^i_{auth}\}_{i=1}^n$, $\{\sigma^i_{auth}\}_{i=1}^n$, and aggregates them:

$$\text{vk}_{auth}, \sigma_{auth} \leftarrow \text{Coconut}_{Aggr}(\{\text{vk}^i_{auth}\}_{i=1}^n, \{\sigma^i_{auth}\}_{i=1}^n)$$

Algorithm 11 DeriveKey: Generates signing and verification keys using a deterministic seed, and public parameters.

Require: parameters: $\{G, o, g_1, g_2, h_s, h_{\text{blind}}\}$, seed: input_bytes.
Ensure: Secret key sk, Verification key vk.

Let q be the number of attributes, determined by h_s.
Ensure len(input_bytes) ≥ 32 (for sufficient randomness in scalar derivation).
Parse x as the first scalar:
$x = \text{ScalarFromBytes}(\text{input\_bytes}[0:32])$
Parse $y = [y_1, y_2, \ldots, y_q]$, where:
$y_i = \text{ScalarFromBytes}(\text{input\_bytes}[32i : 32(i+1)])$ for $i \in [1, q]$
If $32(i+1) > \text{len}(\text{input\_bytes})$, fallback to:
$y_i = \text{ScalarFromBytes}(\text{input\_bytes}[0:32])$
Compute the verification key components:
$\alpha = x \cdot g_2$,
$\beta = [y_1 \cdot g_2, y_2 \cdot g_2, \ldots, y_q \cdot g_2]$,
$\gamma = [y_1 \cdot h_{\text{blind}}, y_2 \cdot h_{\text{blind}}, \ldots, y_q \cdot h_{\text{blind}}]$

Return: $sk = (x, [y_1, y_2, \ldots, y_q]), vk = (g_2, \alpha, \beta, \gamma)$