DeFi That Defies: Imported Off-Chain Metrics and Pseudonymous On-Chain Activity*

David W. Kravitz[0000−0001−8237−4425]
Mollie Z. Halverson[0000−0003−3990−2805]

Spring Labs, Marina Del Rey CA 90292, USA research@springlabs.com
http://www.springlabs.com

Abstract. Traditional finance quantifies risk by collecting and vetting reputation information for an individual, such as credit scores or payment history. While decentralized finance (DeFi) is an exceptionally well-suited application of permissionless blockchains, it is severely constrained in its ability to reconcile identities and quantify associated transaction risk directly on-chain. Opening the ecosystem to a broad range of use cases requires consistent pseudonymity and quantifiable reputation. This paper defies the status quo: exploring methods of assessing risk on-chain by efficiently integrating off-chain identity- and attribute- verification and on-chain transaction activity. We achieve this while preserving individual privacy within a competitive and fair environment and retaining compatibility with existing platforms such as Ethereum. Even though blockchains are inherently public, our solution gives users control over release of information that pertains to them. Consequently, our contribution focuses on customized methods that balance the degree of disclosure of provably-sourced user information against the likelihood of the user successfully gaining access to a desired resource, such as a loan under suitable terms. Our solution is consistent with the zero-trust model in that it imports explicit trust from recognized sources through relevant metrics that are subject to continuous update.

Keywords: DeFi · Zero Trust · Data Encryption · Authorization · Smart Contract · KYC · AML · Wallet · Digital Signature · Ethereum

1 Background

1.1 Centralized vs. Decentralized Finance

Decentralized finance (DeFi) differs in several major ways from centralized finance: There is transparency in the operation of financial assets and products; users exert control over their assets, including the degree to which they release information about them; the capability to create and deploy DeFi products is widely accessible, leveraging the infrastructure of a blockchain and its distributed network of consensus management [17]. The architecture is non-custodial, thus

* Supported by Spring Labs.

streamlining the cost structure. The permissionless nature eliminates blocking and censorship. Open auditability ensures checks on collateralization and overall system health. The "Lego"-like structure allows for composability of protocols constructed from a modest number of basic well-understood building blocks. This enables rules-compliant handling of capital while providing rich feature sets for rehypothecation, i.e., the reuse of assets posted as collateral [21]. The FLAX system [10] extends ERC-20, the Ethereum standard for fungible tokens, to enable composable usage of anonymous funds by other smart contracts, as instantiated using existing anonymous payment schemes. [15] proposes a model for disentangling DeFi protocols into their component building blocks, motivated, in part, by a perceived ultimate need to utilize the understanding of frequently used substructures to develop solutions for cross-chain composability.

1.2 Filling a Gap: Consolidate Off-Chain and On-Chain Reputation

Smart contract- enabled blockchain platforms, such as Ethereum, offer the power to bring financial services applications online within a more socially conscious, equitable environment that can balance maintained public visibility and the protection of individual privacy. However, mirroring the decision processes of traditional lending practices requires importing off-chain-established reputational aspects on-chain without violating aspects of users' choices regarding the extent to which they disclose their personal information and under what circumstances. If multiple users have personal information of a given type of data, i.e., dataType, in common, in traditional systems the secure communications overlay hides correlation even under identical representation of such raw attribute values. That does not apply when addressing conveyance of sensitive information on a blockchain, since smart contract execution cannot use privately held cryptographic keys without calling out to off-chain oracles. This would undermine the desired assurance of availability and auditability of transactions that decentralized ledger technology (DLT) can offer. We opt instead to post attestations to the blockchain that each carry a personalized version of a raw attribute value or of a collection of intervals containing such value. Only the specific user can unlock the information during a user-initiated query designed to provide evidence of meeting requirements set as preconditions to automatically access a resource. Within decentralized finance, an example of such a resource is a loan offered under stated terms upon a successful user transaction as determined by a Money Services Business (MSB) smart contract. MSB smart contracts invoke a Know Your Customer (KYC) smart contract which coordinator-generated transactions call to post attestations to the blockchain. KYC regulations and Anti-Money Laundering (AML) directives constitute essential elements of business conducted by an MSB. KYC is the process used to verify identity given specific information and credentials as part of a more extensive AML process in which several safeguards prevent and/or detect financial crimes.

Considerable time and effort have gone into establishing an international framework of monetary policies that the migration to blockchain should take advantage of. This paper focuses on handling migration in a cryptographically

sound manner by offering a suite of customized solutions that do not suffer from the complexity and computational burdens of generic solutions. We achieve computational efficiency of users' proof role in query. We do not consider it practical enough to efficiently manage only the size of proofs and their verification for succinctness. This theme is consistent with [8].

Assigning user-unique pseudonyms via multiparty computation of tokenized user-identifying data and associating updateable secrets that only the intended users can reproduce together establish a foundation upon which we build out our solutions. We predicate our approach on the realization that there is inefficiency in backtracking from anonymity-based methods to offer consistent pseudonymity that deters reputation laundering. Applying decentralization only to specific subprocesses can result in overall inconsistency since there is an inherent degree of centralization in off-chain verification of user credentials and import onto the blockchain of metrics such as user credit scores. Only a limited number of services deserve broad recognition as trusted to access the raw information tied directly to users and to make such assessments. Note that in the zero-knowledge context, there are potential vulnerabilities of an untrusted setup of common reference strings (CRS) [6]. GDPR [1] defines "pseudonymisation." We avoid dependency on awareness by the system infrastructure, which includes KYC processors (as issuers) and an attesting coordinator, of MSB-specific application-level considerations. We also separate attestation and re-attestation based on updated user-related data from user-involved query. Once a user has registered a blockchain wallet into the system, preparation and posting of attestations bearing on such a wallet occur without user involvement. It is only at the point of query that the user retrieves relevant attestation components.

1.3 High-Level Workflow

A KYC provider verifies a user's identity during off-chain user registration, conducts off-chain user data retrieval, and provides an attestation that the KYC provider has stored the material used to perform verification. This attestation is associated with the user's wallet on-chain but made inaccessible without interaction from the user via the wallet. Auditors can determine from KYC provider logs whether verification had indeed occurred as a prerequisite to attestation. An MSB that wants to verify that the user has gone through KYC acquires permission from the user via its MSB smart contract to access the on-chain data to verify and assess this attestation via a KYC smart contract. One way to address user registration and access to code used to execute user-initiated query is via an MSB. A client-side provisioned root of trust, such as a KYC provider public key can assure against undetected MSB-perpetrated malfeasance. Fig. 1 depicts the user onboarding, attestation, and query workflow.

2 System Overview

Section 2.1 lists the italicized terms that are hyperlinked to their definitions or salient characteristics. Registration of a user includes verification of user iden-

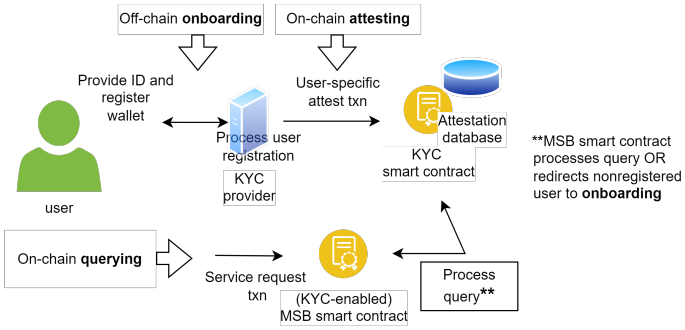

Fig. 1. High-level overview

tity through one or more identity verification services. If verification is successful, gathering of information such as KYC/AML status, credit scores and/or other user data follows, for use in attesting to the blockchain. Such an ***attestation*** ties to a *KYC processor*- and wallet- invariant pseudonym, i.e., ***KYCid***, explicitly generated for that user. The ***coordinator*** is an entity delegated by the ***KYC processors*** to form and submit *attestations* to the blockchain on their behalf. *KYC processors* utilize ***nonces*** to isolate meaningful information exposure through user-initiated ***query*** of *attestations*, as well as to ensure replay detection against unauthorized reuse without relying on compliance by downstream processes. Upfront and later-repeatable deterministic signing by the user's wallet of *attestation*-associated *nonces* enables controlled information exposure during *query* without undesirable statefulness. *nonces* discussed in this context emanate from *KYC processors* and thus are not to be confused with native blockchain nonces used to detect user wallet address-/account- specific transaction submission order and replay attempts. The *KYC processor*, acting in combination with the ***tokenization processors***, performs a tokenization process described in further detail in Appendix 1 to generate the *KYCid*, as well as *attributeValueTokens* in the method introduced in Section 3.2. The off-chain- vetted user data becomes available to an MSB smart contract in some form as *attributeValues* via user-initiated *query*. We define three distinct combinable or stand-alone methods to incorporate a representation of such user data into KYC smart contract-held queryable *attestations* as hash commitments. ***hash***() refers to application of a cryptographic hash function to the parenthetical argument. Due to the blockchain's sequence-preserving nature and signature-enforced immutability of transaction ownership, the most recent *attestation* of the given type of data, i.e., ***dataType***, with embedded ***attributeValue*** that corresponds to the querying user is unambiguously accessible. This remains true even if a user prefers to *query* against an earlier *attestation* and/or an *attestation* of a different user that includes a more favorable embedded *attributeValue*.

2.1 Glossary of Terminology

attestation; *attestationNotFound*; *attestationPointer*; *attestDn*; *attestUp*; *attributeValue*; *attributeValueToken*; *blindedPersonalizationInfo*; *blindedPersonalizedAttributeValueToken*; *coordinator*; *dataType*; *fill*; *hash()*; *intervalLength*; *KYCid*; *KYC processor*; *method 1*; *method 2*; *method 3*; *nonce*; *nonceSig*; *nonceSigKD*; *nonceSig**; *personalizationInfo*; *personalizedAttributeValueToken*; *preimg*; *preimgDn*; *preimgUp*; *preKYCid*; *proxy*; *proxySig*; *query*; *sandwich*; *tokenization processor*; *uberProxy*; *uberSig*; *unterProxy*; *unterSig*; *updateIndex*; *userTxnNonce*; *walletAddress*; *walletSig*; *windowDn*; *windowing*; *windowUp*.

2.2 Detailed Workflow

walletSig is the user wallet- applied signature of ***walletAddress***, as an address derived from the user's wallet public key. ***nonceSig*** is the user wallet- applied signature of *nonce*. ***updateIndex*** is a counter that tracks the updates on the same *KYCid*, *nonce* and *dataType*. The *coordinator*-submitted collection comprised of *dataType*, attestation, *nonce*, and *KYCid* is retrievable from KYC smart contract stateful storage by using as lookup, where || denotes concatenation:

$$\textbf{\textit{attestationPointer}} = hash(hash(walletSig) \,||\, walletAddress) \tag{1}$$

The KYC smart contract code includes the *coordinator*'s whitelisted wallet address used to verify *coordinator*-submitted transactions that each include *attestationPointer*, *dataType*, attestation, *nonce*, and *KYCid*. The outer hash in (1) hides *walletAddress* until the first *query* against the *attestationPointer*. *hash*(*walletSig*) is a condensed form of *walletSig* used during *query* transmission without sacrificing the relationship to its antecedent, *walletSig*, as verified along with *nonceSig* upon user registration.

$$\textbf{\textit{nonceSigKD}} = hash(dataType \,||\, updateIndex \,||\, hash(nonceSig)) \tag{2}$$

nonceSigKD, as a key derived from *nonceSig*, is computable by the user during *query* along with *hash*(*walletSig*) used to rederive *attestationPointer*. (1) is relevant to all three methods presented below, while (2) is relevant only to the first two. Further clarifying Section 1.3: a KYC provider is a composite of a *KYC processor*, identity verification service, third-party data service, *tokenization processors* and *coordinator*. *attestations* include hash commitments of (potentially personalized) *attributeValues*. To mirror *attestations*, personalization occurs live during *query* by a ***proxy*** with access to (non-personalized) *attributeValueTokens* or directly by the user's client. *attestations* are retrievable from the blockchain via rederivation of *attestationPointers* as part of *query*. Fig. 2, 3 and 4 collectively depict a workflow in greater detail than that of Fig. 1.

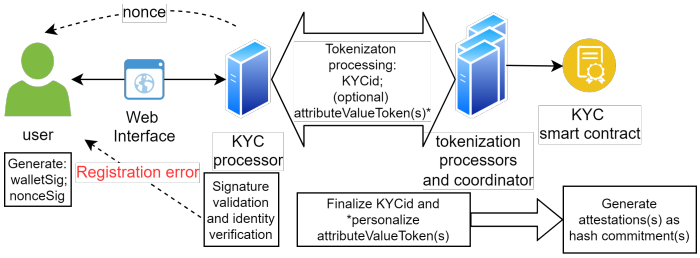

Fig. 2. User registration; tokenization; attestation

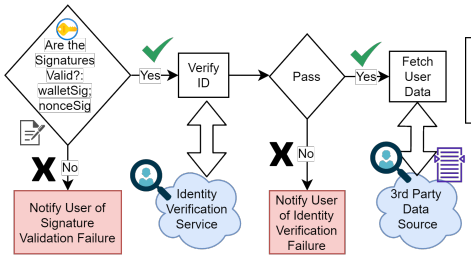

Fig. 3. User registration details

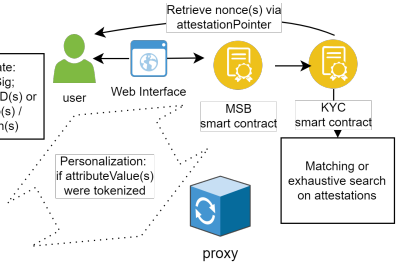

Fig. 4. On-chain query of attested-to attributes

3 Introduction to Suite of Solutions

We now introduce our *attestation* & *query* suite comprised of three customized methods. These all blind access to users' raw attribute values until user-initiated *query*. *query* results in full disclosure, *proxy*-assisted blinded matching of *query* against *attestation*, or user-selectable level of disclosure, respectively.

3.1 Method 1: Release of Raw Attribute Value

In ***method 1*** and *method 2*, *attributeValue*, as received by the *coordinator* via a *KYC processor*, is an argument of the resultant hash commitment:

$$attestation = hash(nonceSigKD \mid\mid attributeValue) \tag{3}$$

A *query* discloses a raw attribute value as an *attributeValue* under *method 1*.

3.2 Method 2: Release of Personalized Attribute Value Token

In ***method 2***, information underlying the *attributeValue* becomes available only to the degree that the outcome of a *query* transaction via an MSB smart contract leaks information. Here the *attributeValue* is a *personalizedAttributeValueToken* that the *coordinator* initially generates via access to an *attributeValueToken*, and that a *proxy* acting as middleware between the user and appropriate MSB smart

contract regenerates during *query*. The ***attributeValueToken*** corresponds to a raw attribute interval that contains the user-associated raw attribute value. The *proxy* assigns a raw attribute interval based on input by the user and the *proxy*'s knowledge of MSB-specific policy governing success criteria. To reduce the probability of a spurious unsuccessful outcome, a user may take advantage of the same third-party monitoring service(s) that supply user data at the time of onboarding and/or during subsequent user-independent updating. Appendix 2 includes details of *method 2*. A need to enable enforcement of dynamic and/or private MSB-specific requirements levied on users for a successful outcome would warrant the relative communication complexity of *method 2*. The corresponding MSB smart contract does not include coding of such requirements.

3.3 Method 3: Windows Around User's Raw Attribute Interval

method 3 aims to balance resource-access success against public disclosure. If armed with awareness of MSB-specific transaction success criteria, the user can selectively release via *query* the maximally-sized window compatible with a successful outcome if such a window that includes the user's raw attribute value exists. For example, the MSB may require credit scores at or above a specified threshold, or an MSB may require the number of past loan application rejections to be at or below a specified threshold. As another example, the MSB may provide service only to users who reside within a specified range of zip codes. Release of one or more consecutive interval labels occurs, where users may or may not be aware of the mapping of interval labels to raw attribute values bounded by interval endpoints. A user can determine from their blockchain-posted *attestations* what specific single interval label of m possibilities was associated with the user for that *dataType*. We consider this interval label, denoted by k, to be an *attributeValue* although the user releases only a disguised form of k.

4 Query: User-to-MSB smart contract Communications

The user accesses their wallet of interest via a client to generate *walletSig* as the signature of the *walletAddress*, which when hashed and combined with *walletAddress* produces *attestationPointer* as in (1). This, along with List(*dataType*) serves as a reference to *attestations* on the blockchain. The body of the transaction includes, in part, *hash*(*walletSig*) and List(*dataType*). *walletAddress* is derivable from the wallet public key, which, in turn, is derivable from the message, signature and a recovery identifier used to disambiguate. In the case of *method 1* and *method 2*, the transaction also includes List(*nonceSigKD*), where *nonceSigKD* is expressed in (2) for *nonceSig* values generated over *nonce* by the user's wallet private key for each *nonce* retrieved from the blockchain as corresponding to an *attestation* of interest. In *method 3*, the transaction includes *preimgUp* and/or *preimgDn* as subsequently defined in (6) and (8), and (7) and (9), respectively. The user generates these based on recomputation of *nonceSig* from retrieved *nonces*, and solving for k of the retrieved *attestations*.

The user may already be otherwise aware of the expected value of k, such as via a third-party source that monitors the user data.

5 Query: KYC Smart Contract Code Execution

The KYC smart contract spans across invocation by all MSB smart contracts. The KYC smart contract attempts retrieval of *KYCid* and *dataType* by using the rederived *attestationPointer* in (1) based on the MSB smart contract-supplied *hash*(*walletSig*) and *walletAddress*. If the *attestationPointer* is absent, the KYC smart contract returns an ***attestationNotFound*** error. The KYC smart contract chooses the most recently blockchain-posted *attestation* of each *dataType* within List(*dataType*) for recomputation of the *attestation* to check for a match. If there is no match on a *dataType*, the KYC smart contract returns an error for that *dataType*. If an MSB smart contract includes *attributeValues* or *preimgUp*/*preimgDn* values within its invocation of the KYC smart contract, the KYC smart contract forms the *attestation* using the given values and checks for a match. In *method 1*, if the MSB smart contract does not supply *attributeValues* for the given *dataType*, the KYC smart contract performs exhaustive search over possible *attributeValues* of *dataType* until it finds a first match, if any.

6 Method 3 Constructions

We utilize one-way hash function chains, wherein anyone can iteratively apply a one-way hash function but cannot feasibly reverse the process to find hash preimages of values they were not involved in generating in the forward direction. We denote applying *hash*() a total of t times by $hash^t$(bitstring), where $hash^0$(bitstring) = bitstring. An *attestation* authorizes the user to initiate a *query* by creating an open or closed "***sandwich***" to selectively reveal a window around their current raw attribute value of a given *dataType*. An open *sandwich* involves the user introducing a single bound (either lower or upper), while a closed *sandwich* involves the user including both bounds. In the open-on-top *sandwich* type, the user chooses the disclosed lower bound interval label $i \leq k$, where k is the potentially hidden interval label corresponding to the interval in which the user's raw attribute value lies. The *dataType*-specific value m denotes the number of raw attribute intervals labeled consecutively from 1 to m. In the open-on-bottom *sandwich* type, the user chooses the disclosed upper bound interval label $m+1-j \geq k$. A closed *sandwich* is comprised of the conjunction of open *sandwich*es of both types. The minimal disclosure open-on-top *sandwich* sets $i = 1$, and the maximal disclosure open-on-top *sandwich* sets $i = k$. Analogously, the minimal disclosure open-on-bottom *sandwich* sets $m+1-j = m$, i.e., $j = 1$, and the maximal disclosure open-on-bottom *sandwich* sets $m+1-j = k$, i.e., $j = m+1-k$. Regarding minimal disclosure, proving that $k \geq 1$ or $k \leq m$ might appear to be vacuous. But this will not be the case when we subsequently introduce the concept of raw attribute sub-intervals. Fig. 5 depicts both *attestation* and *query method 3*- processing that we now explain in mathematical

detail. An *attestation* is comprised of an *attestUp* and *attestDn* pair:

$$\boldsymbol{attestUp} = hash^{k}(windowUp) \tag{4}$$

$$\boldsymbol{attestDn} = hash^{m+1-k}(windowDn) \tag{5}$$

where

$$\boldsymbol{windowUp} = hash(0||\ dataType\ ||\ updateIndex\ ||\ KYCid\ ||\ hash(nonceSig)) \tag{6}$$

$$\boldsymbol{windowDn} = hash(1||\ dataType\ ||\ updateIndex\ ||\ KYCid\ ||\ hash(nonceSig)) \tag{7}$$

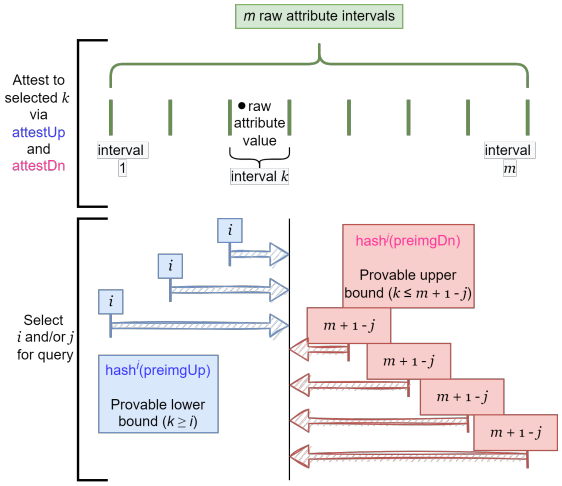

Fig. 5. Sandwich value selection

Unlike the similar-looking expression for *nonceSigKD* in (2), we include *KYCid* here because, unlike *method 2* detailed in Appendix 2, *method 3* uses a consolidated *attestation* generation method that does not deploy two-phase tokenization. For certain *dataTypes*, the system configuration may issue only either *attestUp* or *attestDn*. Additional granularity of *dataTypes* is possible, such as subdivision of the credit score *dataType* corresponding to credit bureau- specific credit scores. Neither the *tokenization processors* nor the *coordinator* accesses information pertaining to k (to the extent that users do not divulge later via selective disclosure) if *KYC processors* generate *attestUp* and *attestDn* values.

Let:

$$\boldsymbol{preimgUp} = hash^{k-i}(windowUp) \tag{8}$$

and

$$\boldsymbol{preimgDn} = hash^{m+1-k-j}(windowDn) \tag{9}$$

To provably narrow the window around k, during *query* the user transacts via the appropriate MSB smart contract by computing and submitting ***preimgs***,

i.e., a *preimgUp* and/or a *preimgDn*, for each targeted attestation. The decision of which or both to submit and the choice of the values of i and/or j depend on the intended level of ***windowing***. The outer-applied hash function within (6) and (7) avoids disclosure of $hash(nonceSig)$, by the user legitimately setting i to k or j to $m + 1 - k$, which would jeopardize its future use. An MSB smart contract provides List(*preimg*) when invoking the KYC smart contract. Note that if we replaced k by $k - 1$ in (4) and (8), or $m + 1 - k$ by $m - k$ in (5) and (9), there would be no way to prove that k is bounded below by 1 or bounded above by m, respectively, since setting $i = 0$ in (8) or $j = 0$ in (9) would not prove anything. Heuristically, *attestUp* and *attestDn* provide only the "haystack," whereas *preimg* values narrow the window where one should look to find the "needle." Users can set their values of i and/or j based on their tolerance for the level of exposure surrounding the value of their k for the given *dataType* rather than prioritizing a successful outcome when transacting with an MSB smart contract. Fig. 6 depicts *query* using a closed *sandwich*. Fig. 7 exemplifies the *attestation* "widening" extension of the basic *sandwich* method, as explained below.

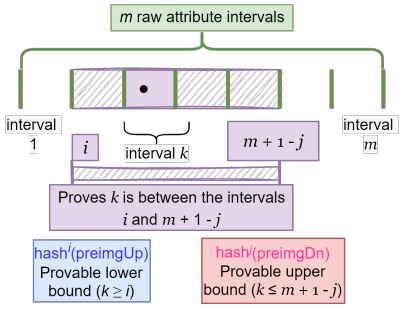

Fig. 6. Sandwich closed example

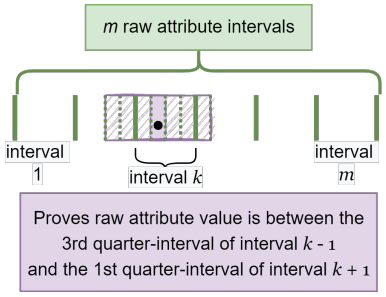

Fig. 7. Closed example of sandwich method using attestation widening

The MSB smart contract provides the i and/or j values to the KYC smart contract, or: The KYC smart contract hashes a received *preimgUp* as in (8) until the result equals *attestUp* as in (4) or the number of hashes performed would exceed an available upper bound. If the result equals *attestUp*, the KYC smart contract determines i such that the user's raw attribute value occurs within raw attribute interval i or greater:

$$hash^i(preimgUp) = hash^i(hash^{k-i}(windowUp)) = hash^k(windowUp) = attestUp$$

It is important to note that, in general, there is exposure of only i and not k. Similarly, the KYC smart contract hashes a received *preimgDn* as in (9) until the result equals *attestDn* as in (5) or the number of hashes performed would exceed an available upper bound. If the result equals *attestDn*, the KYC smart contract determines j such that the user's raw attribute value is located within

raw attribute interval $m + 1 - j$ or lesser:

$$hash^j(preimgDn) = hash^j(hash^{m+1-k-j}(windowDn)) = attestDn$$

It is important to note that, in general, there is exposure of only j and not k.

Regarding an available upper bound, *attestation* transactions can incorporate *dataType*-specific m values, thus making these values available to the KYC smart contract as well as users without MSB smart contract involvement.

The MSB smart contract or the KYC smart contract should check that the user-generated received values are of hash-word length (or truncate them to hash-word length) to prevent a cheating user from successfully using their knowledge of a hash preimage of *windowUp* to "prove" their raw attribute value is located within raw attribute interval $k + 1$. Such a check also ensures that the user's knowledge of a hash preimage of *windowDn* is insufficient to "prove" their raw attribute value is located within raw attribute interval $k-1$. The system assures that an MSB smart contract that invokes the KYC smart contract is authorized to do so if the KYC smart contract includes code that processes an originator signature. The user, as originator of a transaction that results in an MSB smart contract invoking the KYC smart contract, includes within that signed transaction the identifier of the MSB smart contract and the intended epoch. The KYC Smart contract verifies the signature using the wallet public key that matches the *walletAddress* of the *attestationPointer*, and checks that msg.sender matches the address of the invoking MSB smart contract. It also checks that the signed "epoch" value is consistent with the current blockchain epoch.

We now extend the capabilities of the *sandwich* method to give the user increased flexibility in their choice of user-selectable *windowing* via *query*. *windowing* can manage arbitrary *dataTypes* with or without implied order of raw attribute values across labeled intervals. However, users cannot prove their raw attribute values lie within intervals more finely granulated than those set up through *KYC processor*- administered assignment of interval labels on an MSB-agnostic basis. There are two ways to divide raw attribute intervals. Suppose we want to divide the intervals into four equal pieces each. The first way establishes the *attestUp* and/or *attestDn* using a k value between 1 and $4m$. As an example, the first interval originally labeled $k = 1$ is attestable using (4) by hashing between one and four times depending on the specific sub-interval the raw attribute value lies in. We can call this a "deepening" solution, as the size of that *attestUp* remains unchanged at the expense of additional consecutive hashing. Under an attestation-"widening" solution, m stays constant throughout the potentially iterative subdividing of intervals. We can thus maintain the original amount of hashing, at least during a *query*. This second way to divide raw attribute intervals establishes the attestation, i.e., *attestUp* and *attestDn*, by distinguishing four types of raw attribute sub-intervals. If we do not wish to reveal via the *attestation* which sub-interval type the user's raw attribute value lies in, then we must widen the *attestation* to cover all four types.

The inclusion of dummy values as *attestUp* and *attestDn* placeholders corresponding to those quarter-interval types (among 1st, 2nd, 3rd, and 4th) that

do not include the user's raw attribute value would serve the purpose of hiding within the *attestation* identification of the populated quarter-interval type without sacrificing provable conveyance during *query* of the non-dummy quarter-interval type. Under this dummy value paradigm, the user necessarily discloses their populated raw attribute sub-interval type during a successful *query*, and there is a limitation in the user releasing bounded windows using only the non-dummy raw attribute quarter-interval type. We explore next an attestation-widening solution that distinguishes raw attribute sub-interval types without causing this unnecessary degree of information leakage. The goal is to ascertain a security profile that is identical to that of the deepening solution. We exemplify the refined attestation-widening method by using raw attribute quarter-intervals:

Using "***fill***" to denote $dataType \mid\mid updateIndex \mid\mid KYCid \mid\mid hash(nonceSig)$:

$$windowUp_0 = hash(000 \mid\mid fill);\ windowUp_1 = hash(001 \mid\mid fill);$$

$$windowUp_2 = hash(010 \mid\mid fill);\ windowUp_3 = hash(011 \mid\mid fill);$$

$$windowDn_0 = hash(100 \mid\mid fill);\ windowDn_1 = hash(101 \mid\mid fill);$$

$$windowDn_2 = hash(110 \mid\mid fill);\ windowDn_3 = hash(111 \mid\mid fill)$$

Suppose a user's raw attribute value occurs in the 2nd of four raw attribute quarter-intervals of the kth raw attribute interval:

$$attestUp_2 = hash^{k-1}(windowUp_2);\ attestUp_3 = hash^{k-1}(windowUp_3);$$

$$attestUp_0 = hash^{k}(windowUp_0);\ attestUp_1 = hash^{k}(windowUp_1);$$

$$attestDn_1 = hash^{m+1-k}(windowDn_1);\ attestDn_2 = hash^{m+1-k}(windowDn_2);$$

$$attestDn_3 = hash^{m+1-k}(windowDn_3);\ attestDn_0 = hash^{m+1-(k+1)}(windowDn_0)$$

Note that these values can be computed in parallel. The user performs a *query* by computing one of $attestUp_0$, $attestUp_1$, $attestUp_2$ and $attestUp_3$ and/or one of $attestDn_0$, $attestDn_1$, $attestDn_2$ and $attestDn_3$.

Using credit score as an example *dataType*, suppose that a user's actual credit score is 735, thus within the second (i.e., 725-749) raw attribute quarter-interval of the previously configured 700-799 raw attribute interval. For credit scores beginning at 300 and raw attribute intervals of length 100, we would have $k = 5$. Further suppose a particular MSB requires a credit score that exceeds 570 to qualify for a loan: The user reveals their credit score is at least as high as in the 575-599 raw attribute quarter-interval by utilizing $attestUp_3$ that (blindly) corresponds to the 675-699 raw attribute quarter-interval. The user sets $i = 3$, so that $preimgUp_3 = hash^{k-i}(windowUp_3) = hash^2(windowUp_3)$, which the user computes as $hash^3(011 \mid\mid fill)$. There would be greater information leakage in the presence of dummy values for $attestUp_0$, $attestUp_2$ and $attestUp_3$, since, to qualify for a successful outcome, the user would have to reveal their credit score is at least as high as 625 by setting $i = 4$ and utilizing $attestUp_1$ blindly corresponding to the 725-749 raw attribute quarter-interval.

If the user chooses to provably reveal their credit score is in the 725-749 raw attribute quarter-interval, they will use both $attestUp_1$ and $attestDn_1$, setting $i = 5 = k$ and $m + 1 - j = 5 = k$ (i.e., $j = 2$ where $m = 6$ to accommodate credit scores from 300 through 850). This would expose both $windowUp_1$ and $windowDn_1$, but not $hash(nonceSig)$.

A natural question to address now is whether *windowing* is configurable to enable the user to provably narrow within the actual raw attribute interval in which their raw attribute value occurs without the *attestation* process enduring the communications or computational cost of quantizing to that level throughout. The answer is affirmative: exploiting a fractal viewpoint for the specific raw attribute interval that contains the user's raw attribute value, we specify generation of an *attestation* that is comprised of specialized values $attestUp_{\text{Final}}$ and $attestDn_{\text{Final}}$. In this case, we replace m in (5) and (9) by ***intervalLength*** that conveys the number of discrete subdivisions of the kth raw attribute interval (or of the raw attribute sub-interval within the kth raw attribute interval). The granularity of subdivision can be down to the level of all possible raw attribute values within the kth raw attribute interval (or the appropriate raw attribute sub-interval) for discretely populated *dataTypes*. In our example, we would have (for $intervalLength = 25$): $attestUp_{\text{Final}} = hash^{11}(windowUp_{\text{Final}})$ (since $735 = 725 + 10$) and $attestDn_{\text{Final}} = hash^{25+1-11}(windowDn_{\text{Final}})$ for, say differentiated $windowUp_{\text{Final}} = hash(0111 \,||\, fill)$ and $windowDn_{\text{Final}} = hash(1111 \,||\, fill)$. Then the user can provably further narrow within the raw attribute quarter-interval 725-749, potentially down to 735. *intervalLength* is not necessarily the same for all raw attribute intervals/sub-intervals of a given *dataType*. Logically, each MSB smart contract is coded to ignore output of KYC smart contract execution of inputs purported to be specialized *preimgs* (i.e., $preimgUp_{\text{Final}}$ and/or $preimgDn_{\text{Final}}$) unless the necessary conditions are met. Namely: the *query* successfully results in narrowing to a single raw attribute interval (or single raw attribute sub-interval), i.e., $i = m + 1 - j$; the raw attribute sub-interval type, if any, of the matched *attestUp* is the same as that of the matched *attestDn*.

7 Related Work

In this section, we examine related work as alternatives to our contributions that are less private or more generic in their approach to the blockchain-compatible release of user information relevant to successful access to a resource. Unlike our methods, there is no customized splitting of functionality and of access to secrets across components per the least privilege principle.

7.1 NFT-Based Import of KYC/AML Status

ERC-1155 [18] enables a scaled-back version of KYC import [2] with the limitation that direct on-chain *attestations* of user attributes are immediately visible without a user-initiated *query*. Without a *coordinator*, each issuer's public

key is whitelisted within the KYC smart contract that the issuer calls to register/update user KYC/AML status as associated with a user-specific *KYCid*. Across issuers, there is uniform generation of *KYCid* values without outsourcing to *tokenization processors*. Irrespective of the reliance on (non-transferable) NFTs as representative of users' status, only authorized issuers can generate function parameters. Relative to the pseudonymous but visible form of *query*, when a user interacts with an MSB smart contract such transaction results in a user status check via the wallet address associated with the signed transaction. Although an off-chain issuer-invariant database is lacking in this *coordinator*-less model, legitimate updates to a user's *KYCid*, as warranted, e.g., via a verifiable change of address, is accommodatable by an on-chain association of a previous *KYCid* to the replacement *KYCid*. Without reliance on a cross-issuer trust paradigm, the issuance of the replacement *KYCid* depends on off-chain proof by the user of ownership of the information underlying the replaced *KYCid*. As an alternative to the outsourced tokenization and *attestation* processing deployed by our contribution, the key management could potentially use Shamir secret sharing and on-premise or cloud-based enclaves hosted by the issuers.

7.2 Zero Knowledge Meets Smart Contract

Semaphore [12,13] is "a zero-knowledge signaling framework highly inspired by Zcash, providing a method for users who are part of a group to broadcast an arbitrary string without exposing their identity." It is a generic privacy-preserving base layer on top of which Ethereum applications are buildable. It uses zk Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) to prove that the user registered via the Semaphore smart contract an identity commitment within a Merkle tree as a hash of their public key and randomly generated secrets. It also assures that the signal was only broadcasted once, and by the user who generated the proof. Our reputation visibility and pseudonym consistency requirements are inconsistent with Semaphore's "anonymous authentication, where members of a group can login to a service without revealing which member of the group they are and in the process hide their transaction history."

As [9] points out, "Intuitively, one can imagine a naive implementation of Zcash that copies Bitcoin's unspent transaction output (UTXO) model, but where each UTXO's data is kept off-chain. In its place, a hash is kept on-chain, and a zero-knowledge proof is used to prove that some address has the right to consume the UTXO." Cryptocurrency-only transactions benefit from zk-based anonymity, such as that offered by Zcash. However, for the De-Fi applications on which we focus, we want the user to establish a pseudonymous (*KYCid*- linked) on-chain reputation. Such reputation should not be subject to disavowal except under certain stipulated authorized circumstances. This should remain true even if a user changes their wallet and/or issuer (such as a *KYC processor*) through which they onboard or re-onboard.

7.3 On-Chain Attestations via Decentralized Identity Management

[5] is predicated on a premise that centralized ID federation and management along with services that utilize traditional verification systems for identity authorization do not provide sufficient trust for applications dependent on KYC. Their platform is based on merchants and e-commerce service providers attesting their users and storing the reusable-verification records of the *attestations* on a blockchain network. Although the trust assumptions of this work differ from ours, elements in common include deployment on a permissionless blockchain and the avoidance of storage of user ID information on the blockchain and thus the inherent complexities of regulatory compliance such storage would imply. A major distinction from our work is that [5] does not operate within an existing blockchain such as Ethereum. For example, its Delegated Proof of Stake (DPOS) consensus mechanism distributes time slots randomly to active participants to assure against monopolization of block creation. Their blockchain executes two types of non-utility transactions for consensus management (i.e., seed-part and evidence) and three types of utility transactions that result in gain or loss of tokens (i.e., payments, identity verification requests and *attestations*).

8 Conclusions and Future Work

Our contribution to DeFi provides lenders with the capability to make automated decisions on prospective borrower requests that account for user choice regarding the degree to which users are willing to disclose provably-sourced relevant data. Users can compare publicly advertised terms across lenders, while lenders benefit from visibility into transactions that a given user initiates with other lenders and into such lenders' responses. We achieve these goals using three methods that we customize for maximal efficiency and privacy preservation specific to their operational setting. During query, the user signs using a wallet that earlier submission of the attestation now identified by the query had specified. Our sandwich method enables the desired pseudonymous control by the user over the selective release of aspects of their attribute value without the encumbrance of a zero-knowledge-based setup procedure.

In contrast to stateless reproducibility of deterministic signatures generated by a single user wallet, one area of future work would extend our model to accommodate: multiple wallets corresponding to a single user pseudonym in a k-out-of-n quorum environment; a business for which different members or groups have (potentially hierarchical) signing authority to ensure resilient and secure operation. This should be achievable through a judiciously chosen combination of multisignature, threshold signature and ring signature techniques. However, the requirement for stateless generation by wallets of a per-message uncorrelated random/pseudorandom component within elliptic curve-/discrete log- based signatures presents other security and performance challenges warranting additional research [11].

Ideally, there should be a low probability of false negatives in extracting data from user-presented forms of identification corresponding to the same user as uti-

lized to populate fields used as precursors for identity tokenization. Conversely, identity tokenization should be set up to gather sufficient information to avoid false positives across different users. A construct such as pairwise-pseudonymous decentralized identifiers (DIDs) [20] can run counter to the goals of a reputation-based system, as it would allow an individual to have multiple context-dependent identities. Techniques such as locality-sensitive/fuzzy hashing [3] are applicable, with the caveat that there are constraints in practice based on privacy concerns around retention for secondary screening of raw data/images collected by identity verification services. Preferably, irreversible feature-extraction (deep learning) representations of such raw data are potentially usable instead [19].

1 Appendix: Tokenization Schema

In this section, we present our multi-party token generation methodology that is designed to address the pseudonymous presence of the user on the blockchain, sanitization of raw attribute intervals and personalization of sanitized raw attribute intervals, respectively.

As mentioned in Section 7.1, issuers as *KYC processors* could directly tokenize such that the requirement of uniform results across them is met. However, outsourcing to independently operated *tokenization processors* that act sequentially or in parallel renders the process decentralizable and proactively refreshable without compromising the inputs. Sensitive information is mapped to an elliptic curve point that is transiently blinded via scalar multiplication by a random value. The secret scalars held uniquely by each *tokenization processor* multiply or add together to a constant, although each scalar can be updated as an element of a resplitting operation to thwart successful overall compromise. This remains true unless compromise is so tightly orchestrated across all participating *tokenization processors* as to succeed. Because of the commutativity of scalar multiplication, sequential or parallel multiplication by secret scalars does not impair the ability to unblind final or intermediate results via multiplication by the inverse of the random value previously applied as blinding factor. Combinations of the parallel additive and sequential multiplicative application of secret scalars comprehensively retain the resplit capability that safely enables business continuity of tokenized values. This is especially relevant in an immutable blockchain deployment. Following up with the application of HMAC keys that can be differentiated based on token type renders the resultant tokens non-invertible even if these (non-resplittable) keys are later compromised. Beyond the use of multiple *tokenization processors*, each *tokenization processor* can be partitioned, as can its partitions (reminiscent of the fractal makeup of Merkle trees). In the case of tokenization of the user's identifying information, the resultant ***preKYCid*** can be used as the ultimate *KYCid*, or the *coordinator* can maintain in its database an associated randomly generated value to be used as the on-chain *KYCid*. The latter approach enables authorized severing of the association of the user to a past-utilized *KYCid*, where such association can potentially be reestablished via the user's redeployment of a wallet that was

associated with such previously utilized *KYCid*. Provided that the wallet has not been compromised, signing by such wallet private key of a freshly generated *nonce* proves the user's association.

Table 1 shows the elliptic curve computations used within Appendix 2 and demonstrates flexibility in varied circumstances, such as tokenization as requested of *tokenization processors* by a *KYC processor* vs. personalization overlay by an ***uberProxy*** of an ***unterProxy***-held *attributeValueToken*. The commutative Pohlig-Hellman encryption/decryption operations [16] use the same math as elliptic curve Diffie-Hellman ephemeral key agreement for communications security such as provided by the ECDHE-ECDSA TLS cipher. The Pohlig-Hellman cipher has also resurfaced because of its general applicability to private set intersection [14], and to COVID-19 contact tracing in particular [7].

Table 1. Pohlig-Hellman- and HMAC- based tokenization and personalization

Requestor	Processor A_1	Processor A_1	Processor B
(1) Derives P and blinds P with $e \Rightarrow eP$			
	(2(a)) Applies a_1 to $eP \Rightarrow a_1eP$	(2(b)) Applies a_2 to $eP \Rightarrow a_2eP$	
(3) Computes $a_1eP + a_2eP$ and applies $e^{-1} \Rightarrow (a_1 + a_2)P$			
			(4) Applies $b \Rightarrow b(a_1 + a_2)P$
			(5) Applies HMAC key appropriate to token type

The process outlined in Table 1 shows the requestor deriving an elliptic curve point P from data, blinding P, and communicating the blinded P to Processor A_1 and Processor A_2 simultaneously to allow them to apply their respective secrets (i.e., a_1 held by Processor A_1 and a_2 held by Processor A_2). These two processors may be consolidated into a single processor, Processor A, in which case that processor would apply its single secret a. Upon receiving both processors' results, the requestor adds them together and unblinds it (where if there is a single processor, there is no addition as only one result is received). The requestor then communicates with Processor B, enabling Processor B to apply its secret b. If there is only a Processor A, then the result of step 4 is baP. Finally, Processor B applies an HMAC key, the value of which is dependent on the token type. This processing can be utilized to generate *preKYCids* and *attributeValueTokens*. In

the case of *preKYCids*, the input P can be derived from the user's identifying information. In the case of *attributeValueTokens*, the input P can be derived from the raw attribute interval. The input to derive P for *attributeValueTokens* may also include the *dataType*. The requestor is the *KYC processor*, and the secrets a_1, a_2 and b are the static secrets securely held by Processors A_1, A_2 and B, respectively (i.e., the *tokenization processors*). The *KYC processor* should have no knowledge of the finalized *preKYCid* or the *attributeValueToken*. Therefore, all five steps should be used, necessitating the use of Processor B.

2 Appendix: Details of Method 2

The goal of *method 2*, as introduced in Section 3.2, is to utilize a 3rd-party *proxy* to minimize the amount of information about the raw attribute value that is publicly disclosed via *query*. While *method 3* gives the user more control, *method 2* makes the system less dependent on user involvement. In other words, there may be circumstances in which the user does not wish to pursue the option of participating in *windowing* It may also be essential to allow for dynamic and/or private enforcement of MSB-specific requirements levied on users for a successful outcome where the MSB smart contract code does not reflect such details. However, *method 2* does not allow the same degree of flexibility in the granularity of user control as does *method 3*. In addition, the use of a *proxy* introduces additional complexity that may not be considered a universally worthwhile tradeoff.

Bulk Generation of Attribute Value Tokens and Setup of Proxies

The *dataType*-specific *attributeValueTokens* (prior to personalization) may be precomputed in bulk by the *KYC processor* in conjunction with the *tokenization processors*, stored at the *coordinator*, and distributed on-demand to authorized *proxies* during a setup procedure described in this section.

Efficient operation requires precomputation of sets of [*dataType*, raw attribute interval, *attributeValueToken*] tuples for *proxies* to receive upon setup, for those *dataTypes* they are authorized to address. An example of such a set is a collection of contiguous non-overlapping credit score intervals that span the set of credit score values. *unterProxies* that gain such access constitute part of the core infrastructure in that, unlike *uberProxies* that are denied such access, they are not MSB-specific. In addition to blinding all raw attribute intervals for a given *dataType* to request tokenization processing of these raw attribute intervals, the *KYC processor* applies a permutation that is randomly or pseudo-randomly generated for each *dataType* (and locally stored if not otherwise reproducible) to deny the *coordinator* the capability to map raw attribute intervals to tokens. These *attributeValueTokens* are independent of any data/metadata ancillary to the raw attribute intervals (other than *dataType*, if appended to the raw attribute interval when submitted for tokenization to avoid potential collisions of *attributeValueTokens* across multiple *dataTypes*). In particular, the *attributeValueTokens* are not yet personalized.

To set up an authorized *proxy*, it receives the [*dataType*, raw attribute interval, *attributeValueToken*] tuples required to convert raw attribute values or raw attribute intervals received from users to *attributeValues* that take the form of *personalizedAttributeValueTokens*. The *KYC processor* sends to a *proxy* the *dataType*-specific permutation for each *dataType* that the *proxy* is authorized to process. The *coordinator*, in turn, sends to the *proxy* the set(s) of locally stored *attributeValueTokens* of the indicated *dataTypes*, where the *proxy* restores these permuted *attributeValueTokens* to their proper order of associated raw attribute intervals.

Roles and Access: uberProxy vs. unterProxy

proxy deployment can be adjusted to meet distributed access control requirements consistently with specified roles of each of the components, to achieve the intended overall level of security.

Deployment of *proxies* (whether of distinguished *uberProxy* vs. *unterProxy* nature or as one monolithic *proxy*) enables a user-specific tokenized form of a raw attribute interval as a *personalizedAttributeValueToken*, as was introduced in Section 3.2. This validates that one or more user-associated raw attribute values satisfy MSB-specific requirements (e.g., pertaining to credit scores or zip codes) without exposing the granular data to the publicly accessible immutable blockchain. It is preferable that the *KYC processors*, *tokenization processors*, *coordinator*, MSB smart contracts and KYC smart contract each be able to operate under the following constraints: no awareness of MSB-specific application-level policies, which can change dynamically; no sacrifice of the capability to validate (or reject) *attestations* through *query* usefully; no need to update the code of the KYC smart contract (or even that of the MSB smart contract); no need to modify the construction of raw attribute intervals. It is the responsibility of the *proxy* (or, more specifically, the *uberProxy*) to be aware of and enforce aspects of MSB-specific policy. While some such aspects may become publicly discernible based on outcomes, others may remain proprietary to the relationship between MSBs and their delegated *uberProxies*. Dependent on implementation and regulatory constraints, users may be able to opt-in or out of such a granular data-suppression system without affecting their ability to meet MSB-specific KYC requirements for service fulfillment at the application level.

Users communicate directly with an *uberProxy* designated by the MSB of interest, such as via whitelisting of the *uberProxy* public key within the corresponding MSB smart contract. The *unterProxies* have the *attributeValueTokens* used to compute the *personalizedAttributeValueTokens*, but are at least transiently blinded by the *uberProxy* from access to user-specific information that could otherwise aid in targeting specific users as victims of false-negative matching results during KYC smart contract execution of *queries*. Typically, each *unterProxy* possesses the set of *attributeValueTokens* for a given *dataType*, although *dataTypes* may be divisible into granular *dataTypes* with access by *unterProxies* potentially refined accordingly. For example, a *dataType* comprised of zip code information is divisible into subordinate *dataTypes* corresponding

to geographical regions. There may be multiple *unterProxies* in charge of the same *dataType* for reasons such as load-balancing. In this case, an MSB might be able to provide a preference for which *unterProxies* it would like a designated *uberProxy* to utilize in processing. A user may remain oblivious as to the extent that a *proxy* exists as a whole or in parts. As an example, an *uberProxy* can be split (and remain compatible with Ethereum) by utilizing ECDSA threshold signatures that are verifiable using an associated public key that is invariant of the split-control specifics [4].

Proxy-Based Personalization of Attribute Value Tokens

This subsection of the Appendix details how a *proxy* system involving an *uberProxy* and *unterProxy* (possibly more than one of each type) derives a *personalizedAttributeValueToken* to be leveraged by the user. Expounding upon Fig. 3, this process is depicted in Fig. 8 below.

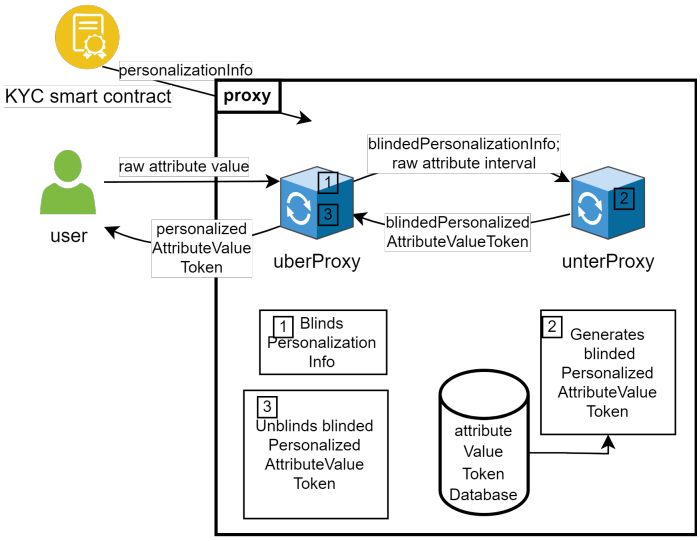

Fig. 8. Personalization of attributeValueTokens

To begin utilizing a *proxy*, a user communicates privately, via a client, with an *uberProxy* to perform a *query* against *attestations*. The user accesses their wallet to generate *walletSig* (via application of the wallet private key), which in combination with *walletAddress* produces *attestationPointer* as in (1). The *attestationPointer* and the *dataTypes* in the selected List(*dataType*) serve as a pointer to the *attestations* on the blockchain, which the MSB smart contract is instructed to *query* against. Using *attestationPointer*, the user retrieves List(*nonce*) and List(*updateIndex*), and signs each *nonce* to form List(*nonceSigKD*) as in (2).

The user provides the *uberProxy* with their wallet public key, List(*dataType*), List(*nonceSigKD*), *walletSig*, MSB smart contract ID and raw attribute value or raw attribute interval for those attributes that (based on *dataType*) are expressed in tokenized form.

The *uberProxy* begins processing by recomputing *attestationPointer*. The *uberProxy* uses the result to retrieve from the KYC smart contract the *KYCid*, List(*nonce*), List(*updateIndex*) and the latest *userTxnNonce* for each *dataType* in List(*dataType*). The ***userTxnNonce*** is a native blockchain nonce used within the blockchain infrastructure and incremented with each successive native blockchain signature generated by the user wallet. The *uberProxy* incorporates the derived value, *userTxnNonce*+1, as an argument of its signature to enable the MSB smart contract to check the freshness of the *proxy* signature as measured against the natively signed transaction received from the user. In a cryptographic context, checking freshness can guard against unwitting acceptance of a received transmission that should be rejected, e.g., because the received transmission contains replayed and/or delayed data (where delay may be measured, e.g., as elapsed time and/or degree of intervening transaction activity). The *uberProxy* checks the MSB policy for the acceptable threshold(s) for the *dataTypes* sent and any instructions concerning routing to specific *unterProxies*.

The *uberProxy* must decide which raw attribute interval to turn into an *attributeValueToken* and therefore use in the process of creating the *personalizedAttributeValueToken*. For clarity in explanation, the description here shows only the decision process for the *dataType* of credit score. However, the process will be similar for other *dataTypes* amenable to partial or total ordering. In the description here, x represents the MSB-specific policy threshold for acceptable credit scores, where the MSB sets the credit score threshold as the highest value within a raw attribute interval. Upon the input by the user of the credit score (or a raw attribute interval of scores), the *uberProxy* determines if the credit score is greater than the necessary threshold x as determined by the MSB-specific policy. If the credit score does not exceed x, then the first raw attribute interval that would exceed x is used, i.e., $[x+1, x+n_{\text{int}}]$ where n_{int} is the size of the raw attribute interval in question. If the credit score is higher than x, then the raw attribute interval of the submitted credit score is used. This description focuses on why the choice of *attributeValueToken* results in a match or not during *query* execution by the KYC smart contract and therefore why the credit score would or would not be approved by the MSB smart contract. Approval of the credit score may be for one of two reasons: the user-provided credit score or raw attribute interval matched the attested raw attribute interval and exceeded the threshold of x; or the attested raw attribute interval exceeded the threshold of x, and this raw attribute interval matched the raw attribute interval selected by the *uberProxy* as higher than that provided by the user. If the credit score was not approved, then it may be for one of three reasons: the incoming raw attribute interval did not match the attested raw attribute interval, although they both exceeded the threshold of x; or the attested raw attribute interval was too low for the MSB policy and therefore did not match

the higher raw attribute interval submitted by the *uberProxy*; or the attested score was higher than $x + n_{\text{int}}$, although the incoming raw attribute interval was lower than x and the raw attribute interval of $[x + 1, x + n_{\text{int}}]$ was submitted by the *uberProxy*. It is acceptable for the conveyance of false information by a user (due to user misrepresentation or an uninformed user) to have the effect of a false negative match but preferably not a false positive match relative to the computation of *attestations* during *query* as compared to their original counterparts attested to the KYC smart contract by the *coordinator*. To reduce the possibility of false negatives, a *proxy* could send multiple *personalizedAttributeValueTokens* for a single queried *attestation*, where dummy values are added by the *proxy* to provide uniformity as a leakage resistance measure. Note that non-uniformity in the number of *personalizedAttributeValueTokens* corresponding to a single *attestation* that are sent during *query* by a user targeting an MSB smart contract could be a potential source of leakage of information, dependent on the rationale (such as a disparity in credit scores across credit reporting agencies and/or a credit score that is at or near the edge of a raw attribute interval). In lieu of or in addition to a *query* resulting in a *proxy* sending multiple *personalizedAttributeValueTokens* per *attestation*, the *coordinator* could send multiple *attestations* to the KYC smart contract. These choices may have bearing on the lengths and number of raw attribute intervals set by the system for each *dataType* supported by the *proxy* method.

To prepare transmission to the *unterProxy*, the *uberProxy* first blinds the

$$\boldsymbol{personalizationInfo} = hash(KYCid \mid\mid nonce) \tag{10}$$

The *uberProxy* converts *personalizationInfo* into an elliptic curve point representation P and applies a random scalar value, e, to form

$$\boldsymbol{blindedPersonalizationInfo} = eP \tag{11}$$

The *uberProxy* transmits to an *unterProxy* equipped for the relevant *dataType*: (a) *blindedPersonalizationInfo*; (b) designation of the submitted raw attribute interval(s); (c) ***uberSig*** (where (c) is the *uberProxy*'s signature computed over (a) and (b)). Upon receiving a request from an *uberProxy*, the chosen *unterProxy* of the several *unterProxies* utilizes its previously provisioned accessible database of [raw attribute interval, *attributeValueToken*] tuples for retrieval of the appropriate *attributeValueTokens* as based on the raw attribute interval(s) designated within the request. The *unterProxy* determines which stored *attributeValueTokens* to use, applies a scalar representation, p, of each such *attributeValueToken* to a provided eP to form a

$$\boldsymbol{blindedPersonalizedAttributeValueToken} = peP \tag{12}$$

and transmits to the *uberProxy*: *blindedPersonalizedAttributeValueTokens*; ***unterSig*** = signed *blindedPersonalizedAttributeValueTokens*. Upon receiving the response from the *unterProxy*, after verifying *unterSig*, the *uberProxy* unblinds the *blindedPersonalizedAttributeValueToken* by applying the inverse of

the previously applied blinding factor, e, to form the

$$\textbf{\textit{personalizedAttributeValueToken}} = hash(pP) \tag{13}$$

The *uberProxy* now signs the MSB smart contract ID, $hash(walletSig)$, $userTxnNonce + 1$, List($dataType$), List($nonceSigKD$), and List($personalizedAttributeValueToken$) to produce ***proxySig***. The *uberProxy* transmits to the user: List($personalizedAttributeValueToken$); $proxySig$.

Partitioning of unterProxy

Attainment of additional security may warrant the additional off-chain communications to split access to *attributeValueTokens* and render tokenized *attributeValues* persistently opaque in the absence of collusion. In this case parallelized secure multiparty computation is utilized. This subsection describes the process by which this is done. The *uberProxy*, $unterProxy_1$ for a given *dataType*, and $unterProxy_2$ for the same *dataType* make up the *proxy*. For example, provision $unterProxy_1$ for a given *dataType* with

$$p_1 = (\text{HMAC}(\text{HMAC key}, y) \,||\, \text{HMAC}(\text{HMAC key}, y+1)) \pmod n \tag{14}$$

and provision $unterProxy_2$ for the same *dataType* with

$$p_2 = (\text{HMAC}(\text{HMAC key}, y+2) \,||\, \text{HMAC}(\text{HMAC key}, y+3)) \pmod n \tag{15}$$

where y is the result of step (4) in Table 1 during the tokenization of raw attribute values, n is the order of the elliptic curve, and the HMAC key was chosen specifically for this purpose. The *coordinator* forms the p_1 and p_2 where $(p_1 + p_2) \pmod n$ is the *attributeValueToken* as subsequently personalized by the *coordinator*. $unterProxy_1$ applies p_1 to the *blindedPersonalizationInfo* computed by the *uberProxy*, as in (11), and sent to $unterProxy_1$. Similarly, $unterProxy_2$ applies p_2 to the *blindedPersonalizationInfo* computed by the *uberProxy* and sent to $unterProxy_2$. $unterProxy_1$ sends back

$$blindedPersonalizedAttributeValueToken_1 = p_1 eP \tag{16}$$

and $unterProxy_2$ sends back

$$blindedPersonalizedAttributeValueToken_2 = p_2 eP \tag{17}$$

The *uberProxy* then sums the elliptic curve points comprised of $blindedPersonalizedAttributeValueToken_1$ and $blindedPersonalizedAttributeValueToken_2$, removes the blinding factor e, and hashes to produce

$$personalizedAttributeValueToken_{1,2} = hash((p_1 + p_2)P) \tag{18}$$

Security Considerations

This subsection outlines the *proxy* method considerations that arise when relating the material included earlier in Appendix 2 to Sections 3, 4 and 5 and Appendix 1.

Beyond the user's wallet public key, *hash*(*walletSig*), List(*dataType*) and List(*nonceSigKD*), also included in the user's transaction is *proxySig* and List(*attributeValue*) = List(*personalizedAttributeValueToken*), where the *proxy* computed each *personalizedAttributeValueToken*.

The token computation processing as presented in Table 1 is utilized during the *personalizedAttributeValueToken* generation as completed in (13) and (18). Only the first three steps should be used here, as the *uberProxy* (as requestor) is the one that should be in possession of the final value. In the scenario described in Appendix 1, Processor A_1 and A_2 are either combined as a single *unterProxy* that holds a secret p as used in (12), or the split is retained as $unterProxy_1$ and $unterProxy_2$. The secret that $unterProxy_1$ holds is p_1 as in (14) and similarly, the secret that $unterProxy_2$ holds is p_2 as in (15). The input to derive P is the *personalizationInfo* as (10). If the *unterProxy* is split, the *uberProxy* completes step (3) as in Table 1, followed by a hash to form (18). If the *unterProxy* is acting as a single processor, then the *uberProxy* does the computation resulting in (13) without the need for the addition portion of step (3) as there is only one received value.

The exhaustion process discussed in Section 5 does not apply to *attributeValues* that are in the form of *personalizedAttributeValueTokens*. If the KYC smart contract has *attributeValueTokens* available to construct *personalizedAttributeValueTokens*, it would defeat the purpose of otherwise carefully limiting access. Knowledge of *attributeValueTokens* (even without their associated raw attribute intervals) would enable correlation of *attributeValues* across users if the information required to complete the computation of *attributeValues* were publicly available via the blockchain.

Yet greater user privacy is achievable by replacing the *nonce* argument of (10) by a value that is not derivable from data/metadata that surfaces on the blockchain. Such replacement defies potential after-the-fact reconstruction of (13) or (18) by an errant *unterProxy* or errant partitioned *unterProxy* under collusive attack. Such reconstruction would reveal which *attributeValueToken* was used for a queried *attestation*. This would be possible without necessarily tracking, and even without having been involved in original construction because of load-balancing use of multiple copies of the *unterProxy*. As an example, we could replace *nonce* here by

$$\boldsymbol{nonceSig*} = hash(hash(nonceSig) \mid\mid updateIndex) \tag{19}$$

where the user's client would provide *nonceSig** only to an *uberProxy* that the client trusts to represent the intended MSB. Such substitution does not impede the generation of *attestations* because *nonceSig** is not MSB-specific and is available to the *coordinator* via its knowledge of the elements in (19).

References

1. Article 4 GDPR: Definitions (2016), general data protection regulation
2. Getting started (2022), https://docs.quadrata.com/integration/introduction/introduction-to-quadrata-web3-passport
3. Andoni, A., Indyk, P., Nguyen, H.L., Razenshteyn, I.P.: Beyond locality-sensitive hashing. CoRR **abs/1306.1547** (2013). https://doi.org/10.48550/arXiv.1306.1547
4. Aumasson, J.P., Hamelink, A., Shlomovits, O.: A survey of ECDSA threshold signing. IACR Cryptol. ePrint Arch. **2020**(1390) (2020)
5. Balahontsev, V., Tsikhilov, A., Norta, A., Udokwu, C.: A blockchain system for the attestation and authorization of digital assets. Tech. rep., Tallinn University of Technology (07 2019). https://doi.org/10.13140/RG.2.2.25027.96807/1
6. Bellare, M., Fuchsbauer, G., Scafuro, A.: NIZKs with an untrusted CRS: security in the face of parameter subversion. IACR Cryptol. ePrint Arch. p. 372 (2016). https://doi.org/10.1007/978-3-662-53890-6_26
7. Berke, A., Bakker, M.A., Vepakomma, P., Raskar, R., Larson, K., Pentland, A.S.: Assessing disease exposure risk with location histories and protecting privacy: A cryptographic approach in response to a global pandemic. CoRR **abs/2003.14412** (2020). https://doi.org/10.48550/arXiv.2003.14412
8. Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 681–710. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-84242-0_24
9. Chen, T., Lu, H., Kunpittaya, T., Luo, A.: A review of zk-SNARKs (2022). https://doi.org/10.48550/ARXIV.2202.06877, https://arxiv.org/abs/2202.06877
10. Dai, W.: Flexible anonymous transactions (FLAX): Towards privacy-preserving and composable decentralized finance. IACR Cryptol. ePrint Arch. **2021**, 1249 (2021)
11. Garillot, F., Kondi, Y., Mohassel, P., Nikolaenko, V.: Threshold schnorr with stateless deterministic signing from standard assumptions. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 127–156. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-84242-0_6
12. Gurkan, K.: Semaphore (2022), https://github.com/appliedzkp/semaphore
13. Gurkan, K., Jie, K.W., Whitehat, B.: Community proposal: Semaphore: Zero-knowledge signaling on Ethereum. White Paper (2020), https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-semaphore.pdf
14. Ion, M., Kreuter, B., Nergiz, A.E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., Yung, M.: On deploying secure computing: Private intersection-sum-with-cardinality. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 370–389 (2020). https://doi.org/10.1109/EuroSP48549.2020.00031
15. Kitzler, S., Victor, F., Saggese, P., Haslhofer, B.: Disentangling decentralized finance (DeFi) compositions. CoRR **abs/2111.11933** (2021). https://doi.org/10.48550/arXiv.2111.11933
16. Pohlig, S., Hellman, M.: An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (corresp.). IEEE Transactions on Information Theory **24**(1), 106–110 (1978). https://doi.org/10.1109/TIT.1978.1055817

17. Qin, K., Zhou, L., Afonin, Y., Lazzaretti, L., Gervais, A.: CeFi vs. DeFi - comparing centralized to decentralized finance. arXiv **abs/2106.08157** (2021). https://doi.org/10.48550/arXiv.2106.08157
18. Radomski, W., Cooke, A., Castonguay, P., Therien, J., Binet, E., Sandford, R.: Eip-1155: Multi token standard (2018), Ethereum Improvement Proposals
19. Sudharsanan, R., Gopirajan, P., Kumar, K.S.: Efficient feature extraction from multispectral images for face recognition applications: A deep learning approach. Journal of Physics: Conference Series **1767**(1), 012061 (feb 2021). https://doi.org/10.1088/1742-6596/1767/1/012061
20. (W3C), C.C.G.: A primer for decentralized identifiers (Nov 2021), https://w3c-ccg.github.io/did-primer/, draft
21. Werner, S.M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., Knottenbelt, W.J.: Sok: Decentralized finance (DeFi). CoRR **abs/2101.08778** (2021). https://doi.org/10.48550/arXiv.2101.08778