Verifiable Blind Pseudonymization via Polymorphic Encryption Using Zero-Knowledge Proofs

Job Doesburg
Radboud University
Nijmegen, The Netherlands
job.doesburg@ru.nl

Bernard van Gastel
Radboud University
Nijmegen, The Netherlands
bernard.vangastel@ru.nl

Erik Poll
Radboud University
Nijmegen, The Netherlands
erik.poll@ru.nl

Abstract

Pseudonymization is a basic technique for privacy protection, limiting identifiability and linkability of data. Data can be pseudonymized blindly, i.e., while being end-to-end encrypted, using the Polymorphic Encryption and Pseudonymization (PEP) framework. This means that the party performing pseudonymization does not see the values of pseudonyms and cannot determine if two (encrypted) pseudonyms belong to each other, ensuring *confidentiality* and *unlinkability*. The pseudonymizing party can, however, unnoticeably corrupt any data being exchanged. This violates *integrity*. We present an extension of the PEP framework that does guarantee integrity without breaking its unlinkability features. With this, we bring the verifiability technique of two-party verifiable oblivious pseudorandom functions (OPRFs, RFC 9497) to the more advanced three-party setting of PEP. Where related systems use bilinear pairings to verify plaintext pseudonyms, our approach proves correct transformation of ciphertexts using Schnorr-style non-interactive zero-knowledge proofs that require no pairings and run on standard, fast curves such as Curve25519, making it significantly more efficient and practical to implement while offering similar privacy guarantees. We also show that the interplay between verifiability and unlinkability requires careful analysis: making rerandomization verifiable defeats the unlinkability it provides, and preserving pseudonym confidentiality requires reshuffling and rekeying to be proved jointly while rerandomization is proved separately.

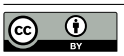
CCS Concepts

• **Security and privacy** → **Pseudonymity, anonymity and untraceability**; **Privacy-preserving protocols**; *Public key (asymmetric) techniques*; • **Theory of computation** → *Cryptographic protocols.*

Keywords

pseudonymization; homomorphic encryption; zero-knowledge proofs; verifiable computation

ACM Reference Format:
Job Doesburg, Bernard van Gastel, and Erik Poll. 2026. Verifiable Blind Pseudonymization via Polymorphic Encryption Using Zero-Knowledge Proofs. In *25th Workshop on Privacy in the Electronic Society (WPES '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3847192.3847358



WPES '26, The Hague, Netherlands

ACM ISBN 979-8-4007-3026-9/2026/11
https://doi.org/10.1145/3847192.3847358

1 Introduction

In today's digital society, where vast amounts of personal data are processed by numerous parties, privacy protection is an important challenge. One approach to address this challenge is pseudonymization, limiting the identifiability and linkability of data by replacing identifiers such as names, email addresses, or social security numbers with unlinkable pseudonyms. There are various ways to compute these pseudonyms, ranging from randomized mapping tables to cryptographic hash functions or encryption.

In 2017, Verheul and Jacobs proposed the PEP (Polymorphic Encryption and Pseudonymization) framework [26]: a novel way to use the homomorphic properties of the ElGamal public key encryption system for blind pseudonymization, i.e., pseudonymization while the identifiers remain encrypted. In PEP, users exchange data *about* data subjects, with each user knowing those subjects under their own pseudonyms that are unlinkable across users (*pseudonym unlinkability*). The party performing pseudonymization does not see the values of pseudonyms (or any other data that is being exchanged), nor do users see pseudonyms for other users, ensuring *confidentiality*. Moreover, because encryption is polymorphic, ciphertexts of the same pseudonym are also unlinkable, ensuring *ciphertext unlinkability*. This enables pseudonymization without creating a privacy hotspot.

More generally, such blind pseudonymization can be realized by *oblivious evaluation* of a pseudorandom function (PRF) such as the Diffie–Hellman PRF (DH-PRF), where the evaluator (i.e., the party performing pseudonymization) applies the pseudonymization key without learning the input or output. In a *blinding-based* (two-party) protocol, a *requester* masks the input with fresh randomness, sends it to the *evaluator* who applies the pseudonymization key, and the same requester removes the mask from the result [10]. In an *encryption-based* (three-party) protocol, the input is encrypted asymmetrically by the sender, the evaluator (called a *converter*, *transformer*, or, as we do, *transcryptor*) applies the pseudonymization key homomorphically to the ciphertext, and the receiver decrypts the result, without the sender needing to be involved. The PEP framework follows this more advanced encryption-based approach, with separate sender, transcryptor, and receiver roles. PEP has existing applications in identity management, namely in the Dutch national eID scheme *DigiD Hoog* [20, 21], and for storage and sharing of medical research data [25], in the 'PEP Responsible Data Sharing Repository' (referred to as PEP-repo).

Specifically, the PEP framework uses three homomorphic operations on ElGamal ciphertexts: to *rekey* (RK) ciphertexts from one party to another, *reshuffle* (RS) their plaintext values into different pseudonyms, and *rerandomize* (RR) these ciphertexts into different unlinkable representations. While these operations ensure

confidentiality and unlinkability, they do not guarantee integrity: nothing binds a transcryptor's output to the correct application of these operations on its input, so it may return an arbitrary ciphertext in place of the honestly computed result. Transcryptors are therefore single points of failure (SPOFs) for integrity, able to corrupt any messages that are being exchanged.

This is especially problematic for pseudonymization, because corruption cannot be detected easily: where normal data may have its own internal checksums or signatures to guard integrity, pseudonyms purposefully appear random, hence a corrupted pseudonym is indistinguishable from a valid pseudonym. Cryptographic signatures and message-authentication codes are common techniques to achieve integrity, but are incompatible with PEP as they violate the pseudonym unlinkability we aim to achieve.

In this article, we present an extension of the PEP framework that does guarantee integrity without trusting the transcryptors and without breaking its fundamental unlinkability properties, based on well-known Schnorr-style non-interactive zero-knowledge proofs (ZKPs). Such proofs are already used for verifiability in the two-party blinding-based setting, in the form of the verifiable oblivious PRF (VOPRF) [10]. In the three-party encryption-based setting, in contrast, existing verifiable systems verify the *plaintext* pseudonym, using pairing-based cryptography. We bring the ZKP approach to the three-party setting, following a *ciphertext-level* strategy: rather than verifying the plaintext pseudonym, we prove that each *ciphertext transformation* was performed correctly. Because this works directly with the key-homomorphic DH-PRF using plain elliptic-curve operations, our scheme requires no pairings and runs on standard, fast curves such as Curve25519 (via Ristretto255).

The core of our extension is a set of verifiable variants of the PEP operations, together with their integrated and transitive compositions, each producing ZKPs proving that the corresponding operation was performed correctly. Verifying a proof does not require the secret factor itself, only a commitment to it, so we require transcryptors to publish commitments to the reshuffling and rekeying factors they associate with domains and sessions, which verifiers (users or other transcryptors) record to enforce consistency. As an alternative that avoids this bookkeeping, we show how factors can instead be derived from a master key through a Carter–Wegman construction, making the commitments publicly computable. We then discuss the different modes of verification.

Beyond the constructions themselves, we make two observations. First, for integrated PEP operations, the intuitive approach of verifying each primitive individually violates pseudonym confidentiality (Section 3.4.1) and must not be used. Second, verifiable rerandomization is self-defeating when used for unlinkability (Section 3.2.1): its verification requires the original ciphertext, breaking the very unlinkability that rerandomization aims to achieve. This is a fundamental limitation of ciphertext-level verification. Together, these highlight how fragile unlinkability, the central property for protecting privacy, really is.

Rather than a single fixed protocol with one overarching security proof, we present these verifiable operations as a set of composable building blocks, each resting on the standard soundness and zero-knowledge of the underlying DLEQ proofs. The integrity, confidentiality, and unlinkability a deployment ultimately guarantees depend on how these operations are composed and verified. We leave a complete security proof for a protocol to future work.

We base our work on experience implementing the PEP framework in various applications [12], including PEP-repo, where data integrity is an important property. These experiences led to the definition of the desired security properties and highlighted the subtleties reflected in this paper. A preliminary version of our work has already been implemented in PEP-repo; an improved version of this work is implemented in the open-source `libpep` software library.[1]

2 Background

Pseudonym systems [22] are systems where entities (i.e., data subjects) are known to different parties under different, domain-specific identifiers, called pseudonyms. For simplicity, we assume a one-to-one mapping between users (i.e., parties that receive pseudonyms) and domains [14]. We write $X_{@A}$ for the pseudonym of entity X in domain A, obtained by applying an injective, domain-specific function defined by secret key material. The central security requirement is *pseudonym unlinkability*: given $X_{@A}$ and $X'_{@B}$ for domains $A \neq B$, it is infeasible to decide whether $X = X'$ without that key material, which implies that a pseudonym does not reveal the underlying identifier (*pseudonym confidentiality*).

A randomized mapping table realizes this information-theoretically, but at $O(n)$ storage in the size of the domain, and pseudonyms are irrecoverably lost with the table. Cryptographic pseudonymization instead derives pseudonyms from a key, giving constant storage and computational unlinkability. We consider one such scheme, PEP, in which these functions are, moreover, evaluated *blindly*, on encrypted identifiers.

This section provides a background on the ElGamal-PEP operations that achieve this (Section 2.1), the system that can be built from them (Section 2.2), and the zero-knowledge proofs that underlie our approach (Section 2.3).

2.1 ElGamal-PEP

The PEP framework [26] describes a homomorphic pseudonymization scheme based on ElGamal, performing pseudonymization operations on encrypted data. It describes three homomorphic operations on ElGamal ciphertexts:

- **Rekeying** (RK) makes a ciphertext encrypted for secret key y decryptable by $k \cdot y$.
- **Reshuffling** (RS) scales the plaintext by a factor s (i.e., M becomes $s \cdot M$) without disclosing M or $s \cdot M$, generating a pseudonym.
- **Rerandomization** (RR) changes the binary representation of a ciphertext without affecting its plaintext or recipient, making otherwise identical ciphertexts unlinkable.

2.1.1 ElGamal. Given a fixed generator point G on an elliptic curve, generating group $\mathbb{G}$, we have the following definitions for ElGamal encryption and decryption:

[1] See https://github.com/NOLAI/libpep.

Definition 1 (Encryption). *Let b be a non-zero random scalar. A plaintext message M can be encrypted for a public key Y as $\langle B, C, Y\rangle$:*

$$\begin{aligned}\mathsf{Enc}(b, M, Y) &= \langle b \cdot G, M + b \cdot Y, Y\rangle \\ &= \langle B, C, Y\rangle\end{aligned}$$

Definition 2 (Decryption). *An ElGamal ciphertext $\langle B, C, Y\rangle$ can be decrypted knowing secret key y (where $Y = y \cdot G$) as:*

$$\begin{aligned}\mathsf{Dec}(\langle B, C, Y\rangle, y) &= C - y \cdot B \\ &= M + b \cdot Y - y \cdot B \\ &= M + b \cdot y \cdot G - y \cdot b \cdot G \\ &= M\end{aligned}$$

An encrypted message $\langle B, C, Y\rangle = \mathsf{Enc}(b, M, Y)$ is thus an ElGamal ciphertext *of* plaintext message M (which is enciphered in C), encrypted *for* (a user possessing) secret key y, being encrypted *with* associated public key Y, *using* random blind b.

Notation. We use the same ElGamal notation as in the original PEP paper [26] (where ciphertexts are $\langle B, C, Y\rangle$ triples instead of $\langle B, C\rangle$ tuples), but with slightly different metavariables for convenience. This notation includes public key Y used for encryption, to make the intended recipient more explicit. $\mathsf{Enc}(b, M, Y)$ could, however, also just be represented as tuple $\langle B, C\rangle$ instead of $\langle B, C, Y\rangle$, as the definition of $\mathsf{Dec}(\langle B, C, Y\rangle, y)$ does not depend on Y. The additional ciphertext component Y can thus be removed in implementations, saving bandwidth and computational overhead. Only when using the RR operation (discussed later), encoding as triples $\langle B, C, Y\rangle$ is required or Y should be passed as a parameter. For completeness, we use $\langle B, C, Y\rangle$ notation throughout this article.

2.1.2 ElGamal-PEP primitives. The three primitive PEP operations that transform ElGamal ciphertexts are the following:

Definition 3 (Rekey – RK). *Given a scalar $k \notin \{0, 1\}$, a ciphertext $\langle B, C, Y\rangle$ for secret key y can be transformed into a new ciphertext which can be decrypted with $k \cdot y$ as*

$$\mathsf{RK}(\langle B, C, Y\rangle, k) = \langle k^{-1} \cdot B, C, k \cdot Y\rangle$$

such that $\mathsf{Dec}(\mathsf{RK}(\mathsf{Enc}(b, M, Y), k), k \cdot y) = M$.

Rekeying is a form of proxy re-encryption [2]: the transcryptor transforms a ciphertext from one encryption key to another without learning the plaintext.

Definition 4 (Reshuffle – RS). *Given a scalar $s \notin \{0, 1\}$, a ciphertext $\langle B, C, Y\rangle$ of message M encrypted with Y can be transformed into a new ciphertext which decrypts to $s \cdot M$ as*

$$\mathsf{RS}(\langle B, C, Y\rangle, s) = \langle s \cdot B, s \cdot C, Y\rangle$$

such that $\mathsf{Dec}(\mathsf{RS}(\mathsf{Enc}(b, M, Y), s), y) = s \cdot M$.

Reshuffling is the encryption-based oblivious evaluation of the Diffie–Hellman pseudorandom function (DH-PRF) $F_s(M) = s \cdot M$, which is pseudorandom under the Decisional Diffie–Hellman (DDH) assumption when s is secret: it applies the PRF to the plaintext homomorphically, without the evaluator learning M or $s \cdot M$.

Definition 5 (Rerandomize – RR). *Given a non-zero scalar r, a ciphertext $\langle B, C, Y\rangle$ encrypted with Y can be transformed into a new ciphertext of the same message and decryptable by the same key as*

$$\mathsf{RR}(\langle B, C, Y\rangle, r) = \langle r \cdot G + B, r \cdot Y + C, Y\rangle$$

such that $\mathsf{Dec}(\mathsf{RR}(\mathsf{Enc}(b, M, Y), r), y) = M$, *where the result is a fresh ElGamal ciphertext with blind $b + r$ under the same key Y.*

Rerandomization changes the binary representation of a ciphertext without changing its contents, which can be used to make otherwise identical ciphertexts unlinkable.

2.1.3 Integrated operations. Notably, an integrated RSK operation can be constructed to perform reshuffling and rekeying simultaneously.[2] This can be used to transform an identifier encrypted by one user into a pseudonym encrypted for a different user.

Definition 6 (RSK). *Given two scalars $s, k \notin \{0, 1\}$, a ciphertext $\langle B, C, Y\rangle$ of message M encrypted with Y for secret key y can be transformed into a new ciphertext that can be decrypted with $k \cdot y$ to $s \cdot M$ as*

$$\mathsf{RSK}(\langle B, C, Y\rangle, s, k) = \langle s \cdot k^{-1} \cdot B, s \cdot C, k \cdot Y\rangle$$

such that $\mathsf{Dec}(\mathsf{RSK}(\mathsf{Enc}(b, M, Y), s, k), k \cdot y) = s \cdot M$.

Rerandomization is typically also applied alongside reshuffling and rekeying to prevent a plaintext injection attack. If a malicious sender submits a ciphertext with a plaintext pseudonym M placed directly in C without proper encryption (i.e., sending $\langle B, M, Y\rangle$), the transcryptor would return the reshuffled plaintext $s \cdot M$ in the clear after performing an RSK operation. This effectively bypasses rekeying: the attacker obtains the recipient's pseudonym without needing their decryption key. Rerandomization prevents this by mixing fresh randomness and the recipient's key into the ciphertext, so that $s \cdot M$ is never exposed to anyone but the recipient. This results in an integrated RRSK operation, where r, s and k are applied simultaneously, that keeps $s \cdot M$ hidden even on a malformed input.

Definition 7 (RRSK). *Given two scalars $s, k \notin \{0, 1\}$ and a non-zero scalar r, a ciphertext $\langle B, C, Y\rangle$ of message M encrypted with Y for secret key y can be transformed into a rerandomized new ciphertext that can be decrypted with $k \cdot y$ to $s \cdot M$ as*

$$\begin{aligned}&\mathsf{RRSK}(\langle B, C, Y\rangle, r, s, k) \\ &\quad = \langle s \cdot k^{-1} \cdot B + r \cdot G,\ s \cdot C + r \cdot k \cdot Y,\ k \cdot Y\rangle\end{aligned}$$

such that $\mathsf{Dec}(\mathsf{RRSK}(\mathsf{Enc}(b, M, Y), r, s, k), k \cdot y) = s \cdot M$ *where the result is a fresh ElGamal ciphertext with blind $\frac{s \cdot b}{k} + r$ under key $k \cdot Y$.*

2.2 The PEP Pseudonym System

The ElGamal-PEP operations can be used to construct a pseudonym system. In the pseudonym system literature, the entities are typically the *users* of the system themselves. In our system model, we instead let pseudonymous communication take place *between* users

[2] In contrast to the original paper, we use RSK instead of RKS. While reshuffling and rekeying commute, the naming emphasizes that reshuffling should be applied before rekeying in sequential execution: if rekeying is performed first and reshuffling is subsequently omitted (or fails), the recipient may obtain the original value M without pseudonymization.

about entities (data subjects) known under different pseudonyms, facilitated by one or more *transcryptors*.[3]

For example, in medical research, PEP is used to share medical data between a measuring device and an analyst performing a research study, with built-in pseudonymization. This prevents the analyst from learning the patient's identity, while the transcryptors can perform access control without seeing the data. In identity management, such as in the Dutch national eID system *DigiD Hoog*, PEP is used to blindly convert a global identifier such as a social security number into multiple unlinkable identifiers (pseudonyms) for different service providers, that are then unable to combine their data. Here, the data subjects are also the end users of the system, but the pseudonymization itself is performed blindly by the transcryptors on their behalf.

2.2.1 Blind pseudonymization protocol. As introduced in Section 1, PEP follows the encryption-based (three-party) approach to blind pseudonymization, where sender, transcryptor, and receiver are distinct parties. For two users A and B to communicate about entity X using their own pseudonyms, conceptually,[4] A first encrypts their pseudonym $X_{@A}$ and then sends this encrypted pseudonym *through* the transcryptor(s) *to* the recipient. The transcryptors perform reshuffling, rekeying, and rerandomization, using specific secret reshuffling and rekeying factors for the specific users that are communicating. Reshuffling applies the DH-PRF to the encrypted pseudonym; rekeying changes the encryption key so only the intended recipient can decrypt the result. This results in a ciphertext containing $X_{@B}$ that can only be decrypted by user B. The encryption guarantees transcryptors cannot see the values of pseudonyms (i.e., pseudonym confidentiality), and because encryption is polymorphic, the transcryptors also cannot distinguish whether two ciphertexts actually encrypt the same pseudonym or not (i.e., ciphertext unlinkability).

Typically, users establish different session keys, resulting in different rekeying factors and different ciphertexts for each session, while the reshuffling factors and thus the pseudonyms are only domain (user) specific. Notably, transcryptors can perform access control and decide to allow transcryption to happen or not, based on the users that are communicating. We do not discuss this further.

We assume authenticated encrypted communication channels between all parties to prevent reordering, replaying, or tampering, which can be realized using standard protocols such as TLS.

2.2.2 Pseudonyms and attributes. While this article primarily concerns the pseudonymization of identifiers, data sharing in practice typically also involves exchanging *attributes* alongside pseudonyms. For the remainder of this article, we therefore consider data to consist of *pseudonyms*, being identifiers for data subjects, and *attributes*, being any facts about these data subjects that do not identify the data subject in any way. Attributes may be shared through a separate protocol, but PEP also natively supports this: to exchange attributes instead of pseudonyms, transcryptors only perform the rekeying operation (omitting the reshuffle). In this case, it is crucial to use different encryption keys and rekeying factors for pseudonyms and attributes to prevent pseudonyms from being rekeyed without being reshuffled, which would violate pseudonym confidentiality and hence pseudonym unlinkability.

Pseudonymization only protects against identification through identifiers. When combinations of attributes form quasi-identifiers that allow re-identification, additional measures to eliminate them are required, which we consider out of scope.

2.2.3 Asynchronous data sharing. Because the encryption-based PEP protocol is non-interactive between sender and recipient, the sender does not need to specify the recipient at encryption time. This enables asynchronous data sharing: data is first encrypted *once*, stored by an intermediate storage, and later shared *multiple times* with different recipients without the original sender being involved. The rerandomization operation can then be used to create multiple copies of such a stored ciphertext, so that recipients cannot tell whether two copies originate from the same stored ciphertext. We do not discuss these use cases further in this article.

2.2.4 Transitive and distributed transcryption. For multidirectional pseudonymization without violating pseudonym unlinkability, the PEP framework must be extended with *transitive* reshuffling operations, so that the direction of pseudonymization between users does not result in different pseudonyms. To do this, one can use reshuffling factor $s = s_{\text{from}}^{-1} \cdot s_{\text{to}}$ consisting of two reshuffling factors for both the sending and receiving user, and a transitive rekeying operation with $k = k_{\text{from}}^{-1} \cdot k_{\text{to}}$. Following the n-PEP extension of the PEP framework [13], where these transitive operations are defined, we denote them RS2, RK2, RSK2 and RRSK2.

Additionally, to remove single points of failure, the PEP primitives from Section 2.1 can be distributed over multiple transcryptors, instead of just a single one, so that $s = \prod_{i=1}^{n} s_i$ and $k = \prod_{i=1}^{n} k_i$ where each individual transcryptor only holds its own s_i and k_i.

Users can then also establish their session keys by combining n session key shares $u_i = b_i \cdot k_i$ from each transcryptor, and a blinded global secret key $y'_b = y \cdot \prod_{i=1}^{n} b_i^{-1}$, where y is a global secret key used during system setup and b_i are random key blinding factors from each transcryptor. We do not further discuss the key management details of distributed transcryption in this article; these are also addressed in n-PEP [13]. While we do present verifiable variants of transitive operations because they add complexity, our insights also apply to basic PEP transcryption in its original form.

2.3 Zero-Knowledge Proofs

Schnorr non-interactive zero-knowledge proofs [18, 23] prove knowledge of a discrete logarithm: given $A = a \cdot G$, a prover demonstrates knowledge of a without revealing it. For our purposes, we need a stronger statement: given $A = a \cdot G$ and $N = a \cdot M$, prove that the *same* secret a relates both pairs. Concretely, we use the Chaum–Pedersen Σ-protocol for equality of discrete logarithms [9], also known as a DLEQ proof, made non-interactive using the Fiat–Shamir transform [15].

Let G be a generator point on an elliptic curve, generating group $\mathbb{G}$, as in Section 2.1.

Definition 8 (DLEQ proof). *Let r be a random scalar. Assume $M \neq O$ (and typically also $A \neq O$ to avoid trivial statements), where $O = 0 \cdot G$ is the identity element. A DLEQ proof* ZKP{N=a·M} *for*

[3] The original PEP framework actually refers to this as a 'transformer'. Another common term is 'converter'.

[4] To simplify implementation, a slightly different but cryptographically equivalent model could be implemented, see the footnote of Section 3.6.

the proper construction of $N = a \cdot M$ *is defined as*

$$\mathsf{ZKP}\{N{=}a{\cdot}M\} = \langle N, C_1, C_2, s\rangle$$

where

$$\begin{aligned} A &= a \cdot G \\ C_1 &= R = r \cdot G \\ C_2 &= r \cdot M \\ s &= a \cdot e + r \\ e &= \mathsf{hash}(A, M, N, C_1, C_2) \end{aligned}$$

with hash *being a secure hash function in the random oracle model.*

Definition 9 (DLEQ verification). *A DLEQ proof* $\mathsf{ZKP}\{N{=}a{\cdot}M\}$ *can be verified given values* M *and* A*. Given the proof components* $\langle N, C_1, C_2, s\rangle$*, the verifier checks if both*

$$s \cdot G \stackrel{?}{=} e \cdot A + C_1$$

$$s \cdot M \stackrel{?}{=} e \cdot N + C_2$$

with $e = \mathsf{hash}(A, M, N, C_1, C_2)$ *as per Definition 8.*

These proofs rely on the hash function behaving as a random oracle: the prover must commit to C_1 and C_2 before the challenge e is determined, because it is computationally infeasible to find commitments that hash to a predetermined challenge. Without this ability, a prover who does not know a cannot compute s satisfying both verification equations for the challenge. Computing s thus proves they know such secret a without actually disclosing a itself. This can be used to prove that value N has properly been computed from value M and some secret a of which only A (which is equal to $a \cdot G$) is known to the verifier.

The three PEP primitives discussed in Section 2.1 consist of a series of scalar multiplications. These proofs can be used to prove that a specific scalar multiplication was performed correctly without disclosing the secret scalar. This can be used to guarantee transcryptor integrity, as we discuss in Section 3. This proof type has been standardized for verifiable oblivious PRFs in the two-party blinding-based setting [10]; we apply it to the architecturally more challenging three-party encryption-based setting.

3 Enforcing Transcryptor Integrity and Consistency

While transcryptors in the PEP framework cannot see the pseudonyms or attributes they transcrypt (Section 1), nothing forces them to transcrypt *correctly*: a transcryptor may apply a wrong reshuffling or rekeying factor, or return an arbitrary ciphertext altogether, yielding malformed pseudonyms after decryption. Just as polymorphic encryption removes the need to trust transcryptors for confidentiality and unlinkability, we use zero-knowledge proofs to remove the need to trust them for integrity, so that no single transcryptor is a point of failure.

In this setting, integrity comes down to *consistency*: a transcryptor must apply, on every transcryption, the correct operation with the same (committed) factors, namely the same reshuffling factor for a given domain and the same rekeying factor for a given session. A transcryptor that breaks this, for instance, by pseudonymizing an identifier into a domain other than the one requested, produces pseudonyms that no longer map to the same value across messages.

Factors are typically derived deterministically, from a transcryptor master secret and the relevant domain or session identifier, using a keyed hash function such as HMAC. This derivation is hidden, so it cannot be checked directly. Instead, we require each transcryptor to publish a commitment $A = a \cdot G$ the first time it uses secret factor a for a domain or session. This commitment binds the transcryptor to that factor without revealing it and is domain-specific for reshuffling and session-specific for rekeying. The commitments are recorded by verifiers (i.e., users or other transcryptors in the distributed setting), who check that a transcryptor uses the same factor for the same domain or session on every transcryption. Verifiers perform this check with the zero-knowledge proofs presented below, without learning the factors themselves, so that consistency, and hence integrity, becomes something they can enforce rather than have to trust.

We first set out our system and threat model (Section 3.1), then present verifiable variants of the basic PEP primitives (Section 3.2) and the factor commitments they require (Section 3.3), then extend these to integrated and transitive operations (Section 3.4) and session key shares (Section 3.5), and finally discuss different models for who performs verification (Section 3.6).

3.1 System and Threat Model

We work in the three-party setting of Section 2.2: a sender encrypts data, one or more transcryptors apply the PEP operations, and a receiver decrypts the result. The proofs we introduce in this section are checked by *verifiers*, which depending on the deployment are users or other transcryptors (Section 3.6). A verifier is not assumed to be trusted.

We take the confidentiality and unlinkability of the original PEP framework as given and focus on integrity. Our concern is a transcryptor that misbehaves: instead of performing the correct operations, it may return an arbitrary ciphertext or apply inconsistent factors across domains or sessions. We therefore do not assume transcryptors are honest, and we allow misbehaving parties to collude, including transcryptors with one another and with senders, receivers, or verifiers. What we do assume is standard: parties communicate over authenticated, encrypted channels such as TLS; the DDH assumption holds in $\mathbb{G}$ and the hash function used for the Fiat–Shamir transform behaves as a random oracle; and once a transcryptor publishes a factor commitment, it cannot later present a different one to a different verifier. Our verifiable operations are built entirely from DLEQ proofs (Section 2.3), whose soundness and zero-knowledge in the random-oracle model are well established.

3.2 Verifiable ElGamal-PEP Primitives

Since ElGamal ciphertexts are $\langle B, C, Y\rangle$ tuples of group elements and the ElGamal-PEP primitives are essentially just scalar multiplications of these group elements with specific secret reshuffling and rekeying factors, we can prove proper transformation of the whole ciphertext by including ZKPs for each transformed group element of the tuple.

We present the verifiable rekey and reshuffle operations below:

Definition 10 (Verifiable rekey – VRK). *A ciphertext $\langle B, C, Y\rangle$ encrypted with Y can be verifiably rekeyed to $k \cdot Y$ ($k \notin \{0, 1\}$), producing*

$$\mathsf{VRK}(\langle B, C, Y\rangle, k) = \langle B', \mathsf{ZKP}\{B{=}k{\cdot}B'\}, \mathsf{ZKP}\{Y'{=}k{\cdot}Y\}\rangle$$

such that $\langle B', C, Y'\rangle = \mathsf{RK}(\langle B, C, Y\rangle, k)$, where B' is taken from the first component and Y' from the first component of the second proof, with C unchanged.

Definition 11 (Verifiable reshuffle – VRS). *A ciphertext $\langle B, C, Y\rangle$ of message M can be verifiably reshuffled to decrypt to $s \cdot M$ ($s \notin \{0, 1\}$), producing*

$$\mathsf{VRS}(\langle B, C, Y\rangle, s) = \langle \mathsf{ZKP}\{B'{=}s{\cdot}B\}, \mathsf{ZKP}\{C'{=}s{\cdot}C\}\rangle$$

such that $\langle B', C', Y\rangle = \mathsf{RS}(\langle B, C, Y\rangle, s)$, where B' and C' are taken from the first component of each ZKP, with Y unchanged.

To verify a VRK or VRS operation, a verifier compares the new ciphertext elements with the original ciphertext and verifies the ZKPs against the known factor commitments (Section 3.3). For VRS, both proofs are verified against S. For VRK, both proofs are verified against K: the B-proof verifies the inverse direction ($B = k \cdot B'$ because $B' = k^{-1} \cdot B$), while the Y'-proof verifies the forward direction ($Y' = k \cdot Y$).

3.2.1 Verifiable rerandomization. A verifiable rerandomize operation, for any random factor r, looks as follows:

Definition 12 (Verifiable rerandomize – VRR). *Given a non-zero scalar r, a ciphertext $\langle B, C, Y\rangle$ of M can be verifiably rerandomized, producing*

$$\mathsf{VRR}(\langle B, C, Y\rangle, r) = \langle r \cdot G, \mathsf{ZKP}\{Y_r{=}r{\cdot}Y\}\rangle$$

such that $\langle r \cdot G + B,\ Y_r + C,\ Y\rangle = \mathsf{RR}(\langle B, C, Y\rangle, r)$, where $R = r \cdot G$ is taken from the first component and Y_r from the ZKP's first component, with Y unchanged.

The ZKP is verified against the Y of the original ciphertext and the provided $R = r \cdot G$.

As discussed in Section 2.1, rerandomization serves two purposes: creating unlinkable copies of a ciphertext for asynchronous data sharing, and preventing plaintext injection attacks as part of regular transcryption. Verification of a VRR operation, however, just as for the other verifiable operations, requires knowledge of the original ciphertext before rerandomization. When rerandomization is used for unlinkability, this is self-defeating: the verifier must know the original ciphertext, which is precisely the link that rerandomization aims to hide. This is inherent to ciphertext-level verification: the statement that one ciphertext is a rerandomization of another, is itself the link to be hidden, and a verifier must know the statement it verifies. A deployment can therefore only choose *against whom* unlinkability is preserved, by choosing who verifies (Section 3.6).

Nevertheless, rerandomization cannot simply be omitted. To prevent the plaintext injection attack from Section 2.1, it must be applied as part of an RRSK operation (see also Section 3.4.2), and must itself be verified so that the subsequent (verifiable) reshuffle and rekey are applied to the rerandomized ciphertext rather than to the (potentially malformed) original. We must therefore use VRR regardless of whether it defeats ciphertext unlinkability. In synchronous transcryption this is harmless, as the verifier observes just a single pass through the system anyway; the problem is fundamental only in the asynchronous data sharing scenario (Section 2.2.3), where a single stored ciphertext is shared with multiple recipients.

As a solution, a dedicated auditing authority that is neither a transcryptor nor a recipient could verify rerandomization on behalf of the recipients, without the recipients themselves learning the original ciphertext. Unlinkability is then maintained from the recipients' perspective, at the cost of granting the auditor knowledge of which copies originate from the same ciphertext.

3.3 Factor Commitments

As introduced in Section 3.2, verification of the VRK and VRS operations requires knowledge of the committed group elements of the secret rekeying factors and reshuffling factors for that domain and session. For every factor a used by a transcryptor (either as reshuffling factor s or rekeying factor k), we require the factor commitment A to be published, together with the domain or session identifier they are used for. Depending on the implementation, this can be published to all users (if they are online), other transcryptors (when distributing transcryption as in Section 2.2.4), or in an (immutable) public register.

A factor commitment binds a transcryptor to *some* factor and lets verifiers enforce that it is reused consistently, but it does not constrain *which* factor was committed: the value is chosen by the transcryptor and stays opaque. The basic commitment scheme therefore guarantees *consistency* of an opaque factor, not the *correctness* of its derivation, nor that it was sampled uniformly at random. A transcryptor (possibly colluding with a recipient) could commit to a consistent yet adversarially chosen or weak factor, for instance one deliberately correlated with another domain's factor, and still pass every consistency check. In the distributed setting (Section 2.2.4), however, this does not break the system: the effective factor $s = \prod_{i=1}^{n} s_i$ is uniform and cross-domain independent as long as a single transcryptor samples honestly. Other transcryptors can additionally reject degenerate commitments, such as one equal to the identity or to a commitment already published for another domain. Whether this residual trust matters therefore depends on the deployment.

3.3.1 Alternative: verifiable factor derivation. The commitment scheme described above requires transcryptors to publish and verifiers to store factor commitments per transcryptor for every domain and session, because the derivation of factors (e.g., via HMAC) is hidden from verifiers and cannot be verified. An alternative is to derive factors from a master pseudonymization key that has a corresponding public key, so that factor commitments become publicly computable and verifiable, without requiring verifiers to store or keep track of commitments.

A naive approach is to derive reshuffling factors as $s_d = x \cdot H_s(d)$, where x is the master pseudonymization key, H_s is a hash-to-scalar function, and d is the domain identifier. The corresponding factor commitment $S_d = H_s(d) \cdot X$ (with $X = x \cdot G$) is then publicly computable from the master public key alone. However, this construction is fundamentally broken: cross-domain ratios $s_a/s_b = H_s(a)/H_s(b)$ are publicly computable because the secret x cancels out. This allows anyone to verify whether two pseudonyms

in different domains belong to the same entity, violating pseudonym unlinkability.

To prevent this, we use a two-key Carter–Wegman universal hashing construction [6]. The reshuffling factor for domain d is derived as

$$s_d = x_1 \cdot H_1(d) + x_2 \cdot H_2(d)$$

where the master pseudonymization key consists of two components x_1, x_2 with published public keys $X_1 = x_1 \cdot G$ and $X_2 = x_2 \cdot G$, and H_1, H_2 are independent hash-to-scalar functions. The factor commitment is publicly computable as $S_d = H_1(d) \cdot X_1 + H_2(d) \cdot X_2$.

Crucially, cross-domain ratios s_a/s_b now depend on both x_1 and x_2 that do not cancel out and are hence no longer publicly computable. This construction provides the information-theoretic guarantee that knowledge of one derived factor reveals no information about any other derived factor.

This construction reduces overhead. Factor commitment storage drops from $O(n)$ domains and sessions to $O(1)$. Factor derivation is now verifiable: verifiers can independently compute the expected commitment and check that the transcryptor used the correct factor for a given domain, closing the gap between factor consistency and factor correctness identified above.

However, factors are no longer independent: they are all derived from the same master key components. With two components, if two derived factors s_a and s_b for known domains a and b are leaked, an adversary can solve for x_1 and x_2, and recompute every factor in the system. The construction generalizes to n master key components, where $s_d = \sum_{j=1}^{n} x_j \cdot H_j(d)$: an adversary must then learn at least n derived factors to recover the master key. In practice, this distinction is limited, since factors only exist inside transcryptors. If a transcryptor is compromised, the adversary likely obtains the master key components directly, regardless of the scheme; the commitment scheme offers no additional protection in this scenario because the independent factors are stored by the same transcryptor. The difference only manifests if individual derived factors leak through a side channel without full transcryptor compromise.

3.4 ZKPs for Integrated PEP Operations

So far, we presented verifiable variants of the basic ElGamal-PEP primitives that guarantee the single operation was performed correctly and consistently with the correct factors. The PEP framework, however, uses integrated operations such as RSK or RRSK, combining reshuffling, rekeying, and rerandomization.

3.4.1 Verifiable RSK. For the integrated RSK operation from Definition 6, a slightly more complex computation is required. The RSK operation requires scalar multiplication with $\frac{s}{k} = s \cdot k^{-1}$, where both s and k are secret.

Definition 13 (Verifiable RSK – VRSK). *A ciphertext $\langle B, C, Y\rangle$ of message M can be verifiably reshuffled and rekeyed to decrypt to $s \cdot M$ under key $k \cdot Y$ ($s, k \notin \{0, 1\}$), producing*

$$\begin{aligned}\mathsf{VRSK}(\langle B, C, Y\rangle, s, k) = \langle T,\ & \\ & \mathsf{ZKP}\{S{=}k{\cdot}T\},\\ & \mathsf{ZKP}\{B'{=}(s \cdot k^{-1}){\cdot}B\},\\ & \mathsf{ZKP}\{C'{=}s{\cdot}C\},\\ & \mathsf{ZKP}\{Y'{=}k{\cdot}Y\}\rangle\end{aligned}$$

where $T = (s \cdot k^{-1}) \cdot G$, such that $\langle B', C', Y'\rangle = \mathsf{RSK}(\langle B, C, Y\rangle, s, k)$, where B', C' and Y' are taken from the first component of each ciphertext-specific ZKP.

The transcryptor delivers the intermediate commitment T alongside the proofs. The first proof is verified against K, confirming that T corresponds to the combined factor $s \cdot k^{-1}$ (since $k \cdot T = S$). The B'-proof is then verified against T, while the C'- and Y'-proofs are verified against S and K respectively.

The intuitive approach for integrated PEP operations, namely performing the individual verifiable primitives sequentially, should in fact not be used. For the VRSK operation, if we were to instead perform a VRK and then VRS in sequence, they would need to be verified separately and hence the intermediate value of both operations would be disclosed to verifiers. Specifically, the rekeyed but not yet reshuffled ciphertext would be disclosed to the verifier. This intermediate ciphertext still encrypts the sender's original pseudonym $X_{@A}$, which has not yet been reshuffled into the recipient's pseudonym $X_{@B}$, but it is now encrypted under the recipient's key $k \cdot Y$. If this verifier is or colludes with the recipient, they can decrypt this intermediate ciphertext using the recipient's secret key $k \cdot y$, recovering the sender's pseudonym $X_{@A}$ directly, violating pseudonym confidentiality.

Similarly, if we switch the order of operations and first perform VRS and then VRK, this would still violate pseudonym confidentiality, though in a slightly more complex manner. Verifiers would receive a reshuffled but not yet rekeyed ciphertext. A verifier colluding with the sender could decrypt this and see the recipient's value for their pseudonym, violating pseudonym confidentiality.

It is thus crucial to first prove the composition of individual factors through a chain of ZKPs on the factor commitments (as in the VRSK construction of Definition 13, which establishes the combined factor $T = s \cdot k^{-1}$ from S and K before applying it to any ciphertext component), rather than applying and proving each operation on the ciphertext separately. Performing the verifiable operations sequentially violates pseudonym confidentiality.

3.4.2 Verifiable RRSK. Making the integrated RRSK operation verifiable requires some care. Recall from Section 2.1 that RRSK keeps the reshuffled pseudonym hidden even when the input ciphertext is malformed. A naive single integrated proof would lose this: the ZKP for the reshuffling of C necessarily exposes $C' = s \cdot C$ as a proof component, and if the input ciphertext is malformed, C' directly reveals the reshuffled pseudonym to any verifier, regardless of B. A verifiable RRSK can therefore not be constructed as a single integrated proof from the DLEQ proofs we use.

In contrast to reshuffling and rekeying, where sequential composition of verifiable primitives is unsafe (Section 3.4.1), here the correct approach is precisely the sequential one: first perform a

VRR, then a VRSK on the rerandomized ciphertext. This is safe because the intermediate rerandomized ciphertext encrypts the same plaintext under the same key, so exposing it to verifiers reveals nothing new. Moreover, after rerandomization C contains the term $r \cdot Y$, so the subsequent ZKP component $C' = s \cdot (C + r \cdot Y)$ no longer exposes the plaintext even if the original input was malformed. The sequential construction therefore keeps the reshuffled pseudonym hidden on malformed inputs, just as the non-verifiable RRSK does.

3.4.3 Verifiable transitive operations. A similar construction is required for the extended reshuffling and rekeying for transitive pseudonymization, as discussed in Section 2.2. Those operations take $s = s_{\text{from}}^{-1} \cdot s_{\text{to}}$ and $k = k_{\text{from}}^{-1} \cdot k_{\text{to}}$, combining factors for the sending and receiving session, to which transcryptors have committed earlier. For reshuffling, we will therefore first prove that some S is consistent with the known factor commitments S_{from} and S_{to}, and similarly for rekeying with factor k. Ultimately, this results in a chain of ZKPs for intermediate products, that prove the integrated operation is correctly performed.

Definition 14 (Verifiable RS2 – VRS2). *A ciphertext $\langle B, C, Y\rangle$ of message M can be verifiably transformed into a new ciphertext that decrypts to $(s_{\text{from}}^{-1} \cdot s_{\text{to}}) \cdot M$, with $s_{\text{from}}, s_{\text{to}} \notin \{0, 1\}$, as*

$$\begin{aligned}\text{VRS2}(\langle B, C, Y\rangle, s_{\text{from}}, s_{\text{to}}) = \langle S,\ & \\ & \text{ZKP}\{S_{to}{=}s_{\text{from}}{\cdot}S\}, \\ & \text{ZKP}\{B'{=}(s_{\text{from}}^{-1} \cdot s_{\text{to}}){\cdot}B\}, \\ & \text{ZKP}\{C'{=}(s_{\text{from}}^{-1} \cdot s_{\text{to}}){\cdot}C\}\rangle\end{aligned}$$

where $S = (s_{\text{from}}^{-1} \cdot s_{\text{to}}) \cdot G$, such that $\langle B', C', Y\rangle = \text{RS}(\langle B, C, Y\rangle, s_{\text{from}}^{-1} \cdot s_{\text{to}})$, where B' and C' are taken from the first component of each ciphertext-specific ZKP, with Y unchanged.

Definition 15 (Verifiable RK2 – VRK2). *A ciphertext $\langle B, C, Y\rangle$ encrypted with Y can be verifiably transformed into a new ciphertext encrypted with $(k_{\text{from}}^{-1} \cdot k_{\text{to}}) \cdot Y$, with $k_{\text{from}}, k_{\text{to}} \notin \{0, 1\}$, as*

$$\begin{aligned}\text{VRK2}(\langle B, C, Y\rangle, k_{\text{from}}, k_{\text{to}}) = \langle K,\ B',\ & \\ & \text{ZKP}\{K_{to}{=}k_{\text{from}}{\cdot}K\}, \\ & \text{ZKP}\{B{=}(k_{\text{from}}^{-1} \cdot k_{\text{to}}){\cdot}B'\}, \\ & \text{ZKP}\{Y'{=}(k_{\text{from}}^{-1} \cdot k_{\text{to}}){\cdot}Y\}\rangle\end{aligned}$$

where $K = (k_{\text{from}}^{-1} \cdot k_{\text{to}}) \cdot G$ and $B' = (k_{\text{from}} \cdot k_{\text{to}}^{-1}) \cdot B$, such that $\langle B', C, Y'\rangle = \text{RK}(\langle B, C, Y\rangle, k_{\text{from}}^{-1} \cdot k_{\text{to}})$, where Y' is taken from the first component of the Y'-proof, with C unchanged.

Definition 16 (Verifiable RSK2 – VRSK2). *A ciphertext $\langle B, C, Y\rangle$ of message M can be verifiably transformed into a new ciphertext that decrypts to $(s_{\text{from}}^{-1} \cdot s_{\text{to}}) \cdot M$ under key $(k_{\text{from}}^{-1} \cdot k_{\text{to}}) \cdot Y$, with $s_{\text{from}}, s_{\text{to}}, k_{\text{from}}, k_{\text{to}} \notin \{0, 1\}$, as*

$$\begin{aligned}\text{VRSK2}(\langle B, C, Y\rangle, s_{\text{from}}, s_{\text{to}}, k_{\text{from}}, k_{\text{to}}) & \\ = \langle S,\ K,\ T, & \\ & \text{ZKP}\{S_{to}{=}s_{\text{from}}{\cdot}S\}, \\ & \text{ZKP}\{K_{to}{=}k_{\text{from}}{\cdot}K\}, \\ & \text{ZKP}\{S{=}(k_{\text{from}}^{-1} \cdot k_{\text{to}}){\cdot}T\}, \\ & \text{ZKP}\{B'{=}(s_{\text{from}}^{-1} \cdot s_{\text{to}} \cdot k_{\text{from}} \cdot k_{\text{to}}^{-1}){\cdot}B\}, \\ & \text{ZKP}\{C'{=}(s_{\text{from}}^{-1} \cdot s_{\text{to}}){\cdot}C\}, \\ & \text{ZKP}\{Y'{=}(k_{\text{from}}^{-1} \cdot k_{\text{to}}){\cdot}Y\}\rangle\end{aligned}$$

where $S = (s_{\text{from}}^{-1} \cdot s_{\text{to}}) \cdot G$, $K = (k_{\text{from}}^{-1} \cdot k_{\text{to}}) \cdot G$, and $T = (s_{\text{from}}^{-1} \cdot s_{\text{to}} \cdot k_{\text{from}} \cdot k_{\text{to}}^{-1}) \cdot G$, such that $\langle B', C', Y'\rangle = \text{RSK}(\langle B, C, Y\rangle, s_{\text{from}}^{-1} \cdot s_{\text{to}}, k_{\text{from}}^{-1} \cdot k_{\text{to}})$, where B', C' and Y' are taken from the first component of each ciphertext-specific ZKP.

In each transitive definition, the first proofs establish the combined commitments against the published factor commitments of s_{from} and k_{from}. The S-proof is verified against S_{from}, showing $s_{\text{from}} \cdot S = S_{\text{to}}$, which establishes S as the commitment to $s_{\text{from}}^{-1} \cdot s_{\text{to}}$. Analogously, the K-proof is verified against K_{from}, showing $k_{\text{from}} \cdot K = K_{\text{to}}$, establishing K as the commitment to $k_{\text{from}}^{-1} \cdot k_{\text{to}}$. For VRSK2, a third proof then combines these two: it establishes T against K by showing $(k_{\text{from}}^{-1} \cdot k_{\text{to}}) \cdot T = S$, so that T commits to the full combined factor $s_{\text{from}}^{-1} \cdot s_{\text{to}} \cdot k_{\text{from}} \cdot k_{\text{to}}^{-1}$. The ciphertext-specific proofs are then verified against these combined commitments, following the same pattern as their non-transitive counterparts. As in VRK (Definition 10), the B-proof in VRK2 and VRSK2 verifies the inverse direction.

3.4.4 Batch optimization. Notice that when exchanging multiple (encrypted) pseudonyms and attributes, the (intermediate) group elements S, K and T and their ZKPs are not specific to the actual ciphertext elements being transformed and thus only need to be provided and verified once per session; the same holds for the public key Y when it is carried as an explicit ciphertext element (Section 2.1.1). This can be optimized by working in batches. Effectively, this reduces a VRS2, VRK2 and VRSK2 to a VRS, VRK and VRSK with extra proofs for the factor commitments, similar to how a RS2, RK2 and RSK2 can be implemented with just a RS, RK and RSK with a specific factor.

Verifiable transcryption then requires only two ZKPs per ciphertext, and three ZKPs plus three group elements per batch. When using the sequential VRRSK = VRSK ∘ VRR approach for a verifiable RRSK operation (Section 3.4.2), the rerandomization step requires one additional ZKP and one additional group element per ciphertext, bringing the total to at most three ZKPs and one group element per ciphertext, and three ZKPs plus three group elements per batch.

3.5 ZKPs for Session Key Shares

The ZKPs described so far prove that transcryptors correctly transformed ciphertexts. However, users must also obtain a session-specific decryption key to decrypt the transcrypted ciphertexts they receive. When using distributed transcryption as described in Section 2.2.4, this session key is not held by any single party; instead, each transcryptor contributes (after authenticating the user)

a session key share $u_i = b_i \cdot k_i$, where k_i is the transcryptor's rekeying factor for that session and b_i is a random blinding factor. The user combines these shares with a blinded global secret key y'_b to reconstruct the session decryption key.

Since the session key share u_i depends on the same rekeying factor k_i used for transcryption, a dishonest transcryptor could provide a session key share derived from a different factor than the one used for actual transcryption, causing decryption to silently produce incorrect pseudonyms. To prevent this, we also require ZKPs on the construction of each session key share. For this, we create a $\mathsf{ZKP}\{U_i{=}b_i{\cdot}K_i\}$ and let users additionally check if $U_i \stackrel{?}{=} u_i \cdot G$, as well as verifying the ZKP using preconfigured $B_i = b_i \cdot G$ commitments for blinding values and K_i from the stored factor commitments. This proves that the session key share was constructed using the same committed rekeying factor as used for transcryption, without revealing that factor.

Note that only the user should receive (and verify) this ZKP, as u_i must remain secret, whereas the other ZKPs may be shared publicly. Since session key retrieval does not need to happen asynchronously, an alternative is to implement a different (possibly interactive) ZKP protocol and possibly incorporate it into the authentication protocol, as long as it proves consistency between the session key share and the committed rekeying factor.

3.6 Verification Models

Using verifiable PEP operations, transcryptors can prove that a specific ciphertext was properly transformed into another ciphertext. Who verifies these transformations is a deployment choice that depends largely on whether data sharing is synchronous or asynchronous.

In a real-time setting, where the recipient is online and receives fresh encrypted and transcrypted data without delay, the recipient can verify the proofs itself. PEP, however, is (also) designed for *asynchronous* data sharing (Section 2.2.3): data may be encrypted by a sender once and transcrypted multiple times (while being rerandomized for ciphertext unlinkability) for different recipients much later.[5] Leaving verification to the recipient is then problematic for two reasons. First, recording the published factor commitments (Section 3.3) requires recipients to be constantly online, which they typically are not. Second, as seen in Section 3.2.1, verification of rerandomization necessarily breaks ciphertext unlinkability against the verifier. Letting the transcryptors verify one another in the distributed setting (Section 2.2.4) avoids both: verification then happens at transcryption time by parties that, unlike users, are always online, and any unlinkability broken by checking a rerandomization is broken only against a transcryptor, not against a recipient. The recipient can still verify the proofs as an end-to-end check, but the integrity of the system no longer hinges on its doing so. There are different modes to implement this verification among the transcryptors.

3.6.1 All transcryptors verify each other. Before a transcryptor decides to perform transcryption of a ciphertext, it first verifies all ZKPs from all previous transcryptors that resulted in this input ciphertext (i.e., the second party verifies the first, the third party verifies the first and second, etc.) The result of their transcryption is not only sent to all subsequent transcryptors but also back to all previous transcryptors, for verification (i.e., the first party also verifies the second and third, etc.). When proofs do not verify (or no proofs are received at all), the transcryptors can, apart from logging and triggering a warning, refuse to create their session key share for the receiving user.

While the most expensive ($n \cdot (n-1) = n^2 - n$ verifications per session), this design is the most secure: corrupting a ciphertext unnoticed requires all n transcryptors to collude. It is truly symmetric in the sense that each party is verified by all $n-1$ other parties.

3.6.2 Each transcryptor verifies all previous parties. This is similar to Section 3.6.1, but more efficient (only $\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$ verifications per session). However, increasing trust is placed on the later transcryptors in the network, with the last transcryptor becoming a SPOF for integrity, because no other transcryptors verify this party anymore. Since distributed transcryption is a commutative operation, the order of transcryptors could be changed every session to limit their power.

Alternatively, we could additionally require the first transcryptor to verify the proofs of the last one. This would, however, make these two (instead of all n) transcryptors together a *point of failure* for integrity, because together they are able to corrupt ciphertexts without the rest of the network noticing.

3.6.3 Each transcryptor verifies the $m < n-1$ previous transcryptors. This is an even more efficient version of Section 3.6.2 ($n \cdot m$ verifications per session) that is also more symmetrical. It does effectively change the level of distributed trust from n to $m+1$, because any $m+1$ colluding parties are able to corrupt a ciphertext without the rest of the network noticing.

4 Evaluation

We evaluate the cost of verifiability by comparing the verifiable PEP operations of Section 3 against their non-verifiable counterparts in our open-source implementation. We do not benchmark competing systems; Section 5 situates our approach against related work.

Setup. All operations run over Ristretto255, the curve on which PEP is typically deployed. We benchmark both ElGamal encodings the implementation supports: the three-element ciphertext encoding common for PEP that additionally carries the recipient public key Y (as discussed in Section 2.1.1), and the more efficient standard two-element ElGamal ciphertext $\langle B, C\rangle$ that does not carry Y (which requires Y to be provided as a parameter for specific operations).

Measurements were taken on an Apple M2 Max, single-threaded, compiled in release mode with rustc 1.95.0 against curve25519-dalek 5.0.0 (which on `aarch64` uses its portable 64-bit serial arithmetic backend) as the mean of 100 samples collected with the `criterion` framework.

[5] This assumes a protocol where a sender A encrypts using its own key (a, A) and sends the encrypted data to the recipient B who possesses key (b, B) and cannot decrypt the data directly, but who will request the transcryptors to transform the ciphertext into something that can be decrypted with its key b. Alternatively, the sender could also request transcryption for the recipient itself. This, however, makes the protocol interactive w.r.t. the sender and recipient because the sender needs to provide the transcryptors with the session from the recipient to transcrypt to. The first protocol does not require such interaction from recipient to sender and is therefore preferred.

The dominant cost is elliptic-curve scalar multiplication; the remaining point additions, hashing (SHA-512), and scalar arithmetic add only a small, roughly fixed overhead. We therefore report, alongside the timings, the exact number of scalar multiplications each operation and its verification require. We likewise report message sizes: on Ristretto255 a group element or scalar occupies 32 bytes, and a zero-knowledge proof is 128 bytes.

Results. Table 1 reports the cost of a single verifiable transcryption, for the base operations and their transitive variants, under both ElGamal encodings.

Proof generation and verification range from roughly 130 µs for a VRR to roughly 1 ms for a VRRSK2, so even the most expensive operation runs at about a thousand per second on a single core and the lighter ones at several thousand. Verifiability multiplies the cost of the underlying plain operation by between roughly two-fold (for RR) and nine-fold (for RSK2).

Across the verifiable operations the running time is highly correlated with the number of scalar multiplications. This confirms that this count is a faithful, implementation-independent proxy for cost, which we use for comparison with related work in Section 5.

Optimizations. The timings in Table 1 are conservative, because several standard optimizations can be implemented. On the prover, multiplications by the fixed generator G can use a precomputed basepoint table, several times faster than a variable-base multiplication. On the verifier, each proof's two verification checks are linear combinations of points and can be evaluated as multiscalar multiplications, roughly halving the verification cost. Moreover, the figures in Table 1 are those of a single transcryption in isolation, which is a worst-case metric: when a transcryptor processes many ciphertexts in a batch (with the same domain and session and hence the same reshuffling and rekeying factors), much of the proof material is independent of the individual ciphertext and can be generated and verified once per batch (Section 3.4.4). Table 1 accordingly reports the scalar-multiplication count split per message and per batch.

5 Related Work

Guaranteeing integrity of blind pseudonymization requires the party performing the conversion (i.e., the transcryptor) to prove it applied the correct pseudonymization key. There are two fundamentally different strategies to achieve this.

The first operates at the *plaintext* level: the conversion protocol produces a proof cryptographically bound to the plaintext pseudonym value, and when the pseudonym is converted to a different domain, this proof is converted along with it so the recipient can verify correctness after decryption. This requires a PRF whose algebraic structure supports such transformable proofs, typically the Dodis–Yampolskiy (DY) PRF [11] $F_k(x) = g^{1/(x+k)}$ with bilinear pairings.

The second operates at the *ciphertext* level: the transcryptor proves that each transformation of the ciphertext was performed correctly, using DLEQ zero-knowledge proofs that chain together to cover composed operations. This works with the key-homomorphic DH-PRF used in our scheme and requires only standard elliptic curves, without pairings.

Both strategies can be applied in the two-party blinding-based protocol or the three-party encryption-based protocol discussed in Section 1. In the two-party setting, the requester participates in every evaluation and can verify the result directly. In the three-party setting, the transcryptor operates on ciphertexts it cannot decrypt and must prove correct transformation to parties that cannot see intermediate values. Our work addresses this more challenging three-party setting.

5.1 Verifiable OPRFs

The IETF Oblivious Pseudorandom Function protocol (RFC 9497) [10] defines a verifiable OPRF (VOPRF) based on the DH-PRF in the two-party blinding-based setting. In the VOPRF protocol, the evaluator provides a proof that it applied the correct key to the blinded input. This is the same type of DLEQ proof we use in our scheme. However, the VOPRF protocol is inherently two-party: the same party must blind the input and unblind the output, so the requester must participate in every evaluation. Our scheme operates in the three-party encryption-based setting where sender, transcryptor, and receiver are distinct parties and pseudonymization can happen asynchronously. Moreover, in the three-party setting, proving correct transcryption involves additional subtleties that do not arise in the two-party case: composed operations must be proved carefully to avoid leaking intermediate values (see Section 3.4.1), and verifying a rerandomization reveals the link it is meant to hide (see Section 3.2.1). The VOPRF can thus be seen as the two-party analogue of our ciphertext-level approach, using the same proof technique but in a simpler protocol model. For a comprehensive overview of OPRF constructions and their security properties, we refer to [7].

5.2 The CL System

The system of Camenisch and Lehmann [4] (the CL system) follows the plaintext approach in the three-party setting. It describes distributed governmental databases where servers communicate about users via a central converter that performs pseudonym conversion. Our system and the CL system both ensure that the central party (converter or transcryptor) cannot observe pseudonyms or link them across domains, providing similar privacy guarantees, but differ in their PRF construction, integrity mechanism, and efficiency.

The CL system uses the DY-PRF, whose conversion protocol requires pairing-friendly elliptic curves equipped with a bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ to enable oblivious conversion and verification of algebraic relationships between secret exponents. Because the DY-PRF is not key-homomorphic, conversion cannot be performed by simply exponentiating the ciphertext. The CL conversion protocol is therefore more involved, relying on signatures that can be both verified publicly and transformed homomorphically alongside the pseudonym. The CL system comes with formal universal composability (UC) security proofs, ensuring security is preserved under arbitrary composition with other protocols.

Our work instead uses the key-homomorphic DH-PRF with ElGamal encryption. We make integrity local to each transcryption step using lightweight Schnorr-style zero-knowledge proofs that prove correct application of factors to a committed ciphertext, in contrast to CL's pairing-based dual-mode signatures that travel

Table 1: Cost of Verifiability per Transcryption, under the Two-Component (2) and Three-Component (3) ElGamal Encodings.

	Time (µs)						Scalar multiplications								Size (B)				
							Per message						Per batch		Per message				Per batch
	Plain		Proof		Verify		Plain		Proof		Verify		Proof	Verify	Plain		Proof		Proof
Operation	2	3	2	3	2	3	2	3	2	3	2	3			2	3	2	3	
Enc	57.0	56.4	–	–	–	–	2	2	–	–	–	–	–	–	64	96	–	–	–
Dec	28.4	29.2	–	–	–	–	1	1	–	–	–	–	–	–	32	32	–	–	–
RR	56.9	56.1	128.7	128.1	130.3	128.0	2	2	4	4	4	4	0	0	64	96	160	160	0
RS	56.8	56.0	230.8	228.4	261.7	257.6	2	2	7	7	8	8	0	0	64	96	256	256	0
RK	38.5	65.3	167.4	293.0	130.3	259.7	1	2	5	9	4	8	0	0	64	96	160	288	0
RSK	66.4	93.1	429.1	547.5	395.6	515.4	2	3	8	12	8	12	5	4	64	96	256	384	160
RRSK	123.2	148.9	556.5	683.7	529.5	647.9	4	5	12	16	12	16	5	4	64	96	416	544	160
RS2	66.1	65.3	399.7	393.4	390.6	403.5	2	2	7	7	8	8	5	4	64	96	256	256	160
RK2	47.3	75.3	337.8	470.0	262.0	393.8	1	2	5	9	4	8	5	4	64	96	160	288	160
RSK2	85.1	114.0	765.1	891.2	652.6	782.6	2	3	8	12	8	12	15	12	64	96	256	384	480
RRSK2	143.2	168.3	899.9	1014.4	786.3	911.7	4	5	12	16	12	16	15	12	64	96	416	544	480

with the pseudonym. This ciphertext-anchored approach trades universal verifiability for substantially lower computational cost.

Regarding the trust model, there are subtle architectural differences. Our system considers multiple transcryptors that verify each other's operations to eliminate single points of failure, while the CL system describes a single converter whose conversions are verified by the receiver after decryption.

5.2.1 Efficiency. The two systems differ substantially in computational cost. A CL conversion requires 94 scalar multiplications and 28 bilinear pairings [4] for the full flow of encryption, conversion, verification, and decryption.[6] The same flow in our scheme requires, for a VRRSK2 (our most expensive operation), 2 scalar multiplications for encryption, 27–31 for proof generation, 24–28 for proof verification, and 1 for decryption (see also Section 4), depending on whether the Y component is included or omitted (Section 2.1.1). A single transcryption in isolation thus totals 54–62 scalar multiplications, with no pairings.

No public implementation of the CL system is available, so we are unable to benchmark it, and a measured head-to-head comparison is not possible; the comparison is necessarily at the level of operation counts. This metric is nonetheless telling: even disregarding the pairings, CL's 94 scalar multiplications per conversion exceed our 54–62 per single transcryption, and the 28 pairings, each considerably more expensive than a scalar multiplication [1], widen the gap further. The difference is also structural: the pairing-based approach is incompatible with Ristretto255, a well-known fast curve on which PEP is typically deployed. Adopting it would require migrating PEP to a pairing-friendly curve.

5.2.2 Synchronous vs. asynchronous integrity. A key difference between the two approaches is where integrity is anchored. The CL system's plaintext-level proofs travel with the pseudonym itself, remaining verifiable even after re-encryption or rerandomization. Our ciphertext-level proofs are tied to specific ciphertext values: verifying a transformation requires knowing the ciphertext before and after.

[6] In the pairing-friendly multiplicative-group setting of the CL system this operation is an exponentiation; we use *scalar multiplication* throughout, matching the additive elliptic-curve notation of PEP.

In the synchronous setting, the two approaches achieve similar security properties. They differ in asynchronous data sharing, where an untrusted storage rerandomizes a stored ciphertext into unlinkable copies. Since our proofs are tied to specific ciphertext values, verifying such a rerandomization requires the original ciphertext and therefore reveals the link between a copy and its original to the verifier (Section 3.2.1). We address this by having transcryptors, rather than recipients, verify these proofs: integrity against the storage is still enforced, and the copies remain unlinkable to the recipients, while only the verifying transcryptors learn the link. The CL system's plaintext-level proofs instead travel with the pseudonym and remain verifiable after rerandomization without the original ciphertext, so a recipient could verify them without learning this link. This is the fundamental trade-off of the ciphertext-level strategy (Section 3.2.1), and which approach fits therefore depends on the deployment. As long as verification can be left to the transcryptors, our ciphertext-level approach handles the untrusted-storage scenario on standard, non-pairing curves. Only if the recipients themselves must act as verifiers while remaining unlinkable across storage rerandomization is CL's more expensive pairing-based plaintext approach needed.

5.2.3 ScrambleDB. ScrambleDB [19] replaces the DY-PRF used by the CL system with the DH-PRF, while keeping ElGamal-based encryption and conversion, which is the same algebraic core as PEP. It introduces the oblivious and convertible PRF (coPRF) as a standalone cryptographic primitive with game-based security definitions for pseudorandomness, obliviousness, and collusion resistance. ScrambleDB, however, does not provide integrity: it assumes an honest-but-curious converter, expected to follow the protocol but treated as adversarial about what it can learn from its inputs and outputs. Our scheme is therefore closer to ScrambleDB than to the CL system cryptographically, with the addition of DLEQ-based integrity proofs that ScrambleDB does not provide.

While the CL system and ScrambleDB have only been implemented as research prototypes, the PEP framework has been deployed in production systems including the Dutch national eID

scheme *DigiD Hoog* [20, 21][7] and a medical data sharing repository [25], providing practical validation of its design choices.

5.3 Other Related Work

More broadly, as discussed in Section 2.2, our scheme is related to anonymous credential and pseudonym systems [5, 8, 22]. However, these systems focus on data subjects (users) authenticating themselves to organizations, whereas we consider organizations sharing data *about* data subjects. This fundamental difference poses different cryptographic challenges: in credential systems, data flows *via* the user (who obtains credentials from issuers and presents them to verifiers) in an unlinkable manner, enabling the user to prove properties about their own credentials. In our system, data flows directly *between* organizations, without the data subject being involved in the actual data exchange themselves.

Our work also relates to homomorphic encryption and verifiable computation. These are active research fields focused on building systems that perform arbitrary computations on encrypted data efficiently [16, 24]. Complex zero-knowledge proof frameworks exist for general-purpose verifiable computation [3, 17]. However, our scheme's operations are elementary enough that DLEQ proofs suffice. This makes our scheme efficient: our proof size is constant per ciphertext component and requires only standard elliptic curve operations already used in the PEP framework.

6 Conclusion

The PEP framework provides strong confidentiality and unlinkability guarantees for pseudonymous data sharing, based on cryptographic primitives, but every transcryptor forms a single point of failure for integrity. Since pseudonyms have to appear random by design, corruption (i.e., violation of integrity) cannot be detected through conventional approaches without violating unlinkability. We presented an extension to the PEP framework that eliminates these single points of failure by using Schnorr-style non-interactive zero-knowledge proofs. Transcryptors commit to the reshuffling and rekeying factors they use for specific users and sessions through public factor commitments, then prove correct use of these factors for each transcryption. Verifiers enforce consistency and thus integrity of pseudonymization by recording these commitments and verifying the ZKPs. As an alternative to verifiers storing published commitments, we showed that factors can be derived from a single master key using a Carter–Wegman construction, making factor commitments publicly computable.

While doing this, we made some non-obvious observations. First, verifiable rerandomization is self-defeating when used for unlinkability (Section 3.2.1): verification requires knowledge of the original ciphertext, breaking the very ciphertext unlinkability that rerandomization aims to achieve. This is a fundamental limitation of ciphertext-level verification. Second, composing verifiable operations requires careful analysis of which operations must be integrated and which must be separated. Rekeying and reshuffling must be proved as an integrated operation (Section 3.4.1): performing them sequentially exposes intermediate ciphertexts that allow a colluding verifier and recipient (or sender) to recover pseudonyms they should not see, violating pseudonym confidentiality. Rerandomization, by contrast, must *not* be folded into a single integrated DLEQ proof. This underscores that the interplay between verifiability and unlinkability requires careful case-by-case analysis for each operation.

These extensions enable the PEP framework to be applied in use cases requiring strong data integrity guarantees, without sacrificing its fundamental privacy properties. Finally, by proving integrity at the ciphertext level using Schnorr-style proofs rather than at the plaintext level using bilinear pairings as in the CL system [4], our approach is significantly more efficient and practical to implement while offering similar privacy guarantees.

[7] In the BSNk scheme underlying the Dutch eID system, ciphertexts are signed by their issuer and signed again after transformation, but the correctness of the transformation is not proved.

Acknowledgments

This research is performed in context of the Dutch National Education Lab AI (NOLAI), funded by the Dutch National Growth Fund.

Ethical Considerations

This work strengthens the integrity guarantees of an existing privacy-preserving system and introduces no attacks or techniques that could be repurposed to harm users. The research involves no human subjects so no ethical review board approval was required.

Open Science

The verifiable PEP operations and the benchmarking code have been implemented in the open-source software library `libpep`, publicly available at https://github.com/NOLAI/libpep.

AI Use Disclosure

During the preparation of this work, the authors used Claude Sonnet and Opus models (versions 4 and 5, Anthropic) for paraphrasing and rewording, improving writing style and flow and checking grammar and spelling. The same tools were also used in the development of the artifact. The authors have manually verified the accuracy, originality, and integrity of the output of all AI-based tools and take full responsibility for the content of this publication.

References

[1] Diego F. Aranha, Paulo S. L. M. Barreto, Patrick Longa, and Jefferson E. Ricardini. 2014. The Realm of the Pairings. In *Selected Areas in Cryptography – SAC 2013*. Springer, Berlin, Heidelberg, 3–25. doi:10.1007/978-3-662-43414-7_1

[2] Matt Blaze, Gerrit Bleumer, and Martin Strauss. 1998. Divertible Protocols and Atomic Proxy Cryptography. In *Advances in Cryptology – EUROCRYPT '98*. Springer, Berlin, Heidelberg, 127–144. doi:10.1007/BFb0054122

[3] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE Symposium on Security and Privacy – S&P 2018*. IEEE, Piscataway, NJ, USA, 315–334. doi:10.1109/SP.2018.00020

[4] Jan Camenisch and Anja Lehmann. 2015. (Un)linkable Pseudonyms for Governmental Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1467–1479. doi:10.1145/2810103.2813658

[5] Jan Camenisch and Anna Lysyanskaya. 2001. An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation. In *Advances in Cryptology – EUROCRYPT 2001*. Springer, Berlin, Heidelberg, 93–118. doi:10.1007/3-540-44987-6_7

[6] J. Lawrence Carter and Mark N. Wegman. 1979. Universal Classes of Hash Functions. *J. Comput. System Sci.* 18, 2 (1979), 143–154. doi:10.1016/0022-0000(79)90044-8

[7] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. 2022. SoK: Oblivious Pseudorandom Functions. In *IEEE European Symposium on Security and Privacy – EuroS&P 2022*. IEEE, Piscataway, NJ, USA, 625–646. doi:10.1109/EuroSP53844.2022.00045

[8] David Chaum. 1985. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044. doi:10.1145/4372.4373

[9] David Chaum and Torben Pryds Pedersen. 1993. Wallet Databases with Observers. In *Advances in Cryptology – CRYPTO '92*. Springer, Berlin, Heidelberg, 89–105. doi:10.1007/3-540-48071-4_7

[10] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. 2023. *Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups*. RFC 9497. RFC Editor. doi:10.17487/RFC9497

[11] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A Verifiable Random Function with Short Proofs and Keys. In *Public Key Cryptography – PKC 2005*. Springer, Berlin, Heidelberg, 416–431. doi:10.1007/978-3-540-30580-4_28

[12] Job Doesburg, Bernard van Gastel, and Erik Poll. 2025. Pseudonymization as a Service: Privacy-Preserving System Evolution through Micropseudonymization. In *Belgium-Netherlands Software Evolution Workshop – BENEVOL 2025*. CEUR-WS. https://benevol2025.github.io/pre/paper12.pdf To appear.

[13] Job Doesburg, Bernard van Gastel, and Erik Poll. 2026. *n*-PEP: Secure Data Sharing with Transitive and Distributed Blind Pseudonymization. In *Security and Trust Management – STM 2026*. Springer Nature, Cham. To appear.

[14] EDPB. 2025. Guidelines 01/2025 on Pseudonymisation. https://www.edpb.europa.eu/our-work-tools/documents/public-consultations/2025/guidelines-012025-pseudonymisation_en

[15] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology – CRYPTO '86*. Springer, Berlin, Heidelberg, 186–194. doi:10.1007/3-540-47721-7_12

[16] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph. D. Dissertation. Stanford University. https://crypto.stanford.edu/craig/

[17] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*. Springer, Berlin, Heidelberg, 305–326. doi:10.1007/978-3-662-49896-5_11

[18] Feng Hao. 2017. *Schnorr Non-interactive Zero-Knowledge Proof*. RFC 8235. RFC Editor. doi:10.17487/RFC8235

[19] Anja Lehmann. 2019. ScrambleDB: Oblivious (Chameleon) Pseudonymization-as-a-Service. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 289–309. doi:10.2478/popets-2019-0048

[20] Logius. 2017. Privacy Impact Assessment DigiD Hoog. https://www.eerstekamer.nl/overig/20190129/privacy_impact_assessments_pia

[21] Logius. 2024. BSNk PP Technische Specificaties v11.1. https://gitlab.com/logius/bsnk/bsnk-techspecs/bsnk/-/raw/main/BPTSlive.pdf

[22] Anna Lysyanskaya, Ronald L. Rivest, Amit Sahai, and Stefan Wolf. 2000. Pseudonym Systems. In *Selected Areas in Cryptography – SAC 1999*. Springer, Berlin, Heidelberg, 184–199. doi:10.1007/3-540-46513-8_14

[23] C. P. Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO '89*. Springer, Berlin, Heidelberg, 239–252. doi:10.1007/0-387-34805-0_22

[24] Justin Thaler. 2022. Proofs, Arguments, and Zero-Knowledge. *Foundations and Trends in Privacy and Security* 4, 2–4 (2022), 117–660. doi:10.1561/3300000030

[25] Bernard E. van Gastel, Bart Jacobs, and Jean Popma. 2021. Data Protection Using Polymorphic Pseudonymisation in a Large-Scale Parkinson's Disease Study. *Journal of Parkinson's Disease* 11, s1 (2021), S19–S25. doi:10.3233/JPD-202431

[26] Eric Verheul and Bart Jacobs. 2017. Polymorphic Encryption and Pseudonymisation in Identity Management and Medical Research. *Nieuw Archief voor Wiskunde* 18, 3 (2017), 168–172.