

Individual Cryptography from Iterated Squaring

Georg Fuchsbauer 

georg.fuchsbauer@tuwien.ac.at

TU Wien

Christoph U. Günther 

cguenthe@ista.ac.at

ISTA

Charlotte Hoffmann* 

charlotte.hoffmann@zama.ai

Zama

Krzysztof Pietrzak 

pietrzak@ista.ac.at

ISTA

Fabian Regen 

fabian.regen@tuwien.ac.at

TU Wien

October 8, 2026

Abstract

Individual cryptography (Dziembowski et al. Crypto’23) and *complete knowledge* (Kelkar et al. CCS’24) aim at constructing protocols that ensure that some individual party must know a secret value in the clear, and so the computation cannot have been “outsourced” to an MPC computation. Two central primitives are *proofs of complete (or individual) knowledge*, which enforce that some party actually knows the secret witness, and *secret sharing with snitching*, which enforces that whenever the secret is reconstructed, some party must learn a certificate of this fact.

The instantiations proposed by both papers (Crypto’23 and CCS’24) ensure that to compute a proof or reconstruct the shared secret, a large number of hash function evaluations is necessary. Thus, at least one party must compute many of these “in the clear” (and not via MPC) and from the hash function inputs one can extract the secret (in the random oracle model). In practice, they suggest using Bitcoin mining hardware.

Sequential proofs of complete knowledge have recently been proposed, which use proofs of sequential work to enforce that the hashes are performed sequentially. As a consequence, it suffices to require a much smaller number of hashes, and thus such primitives can be implemented on standard hardware.

Unfortunately, this approach does not work for primitives like secret sharing, as one would need to efficiently sample the challenge for the sequential computation together with its solution. In this work, we leverage ideas from *time-lock puzzles* to overcome this issue. We construct secret sharing with snitching and encryption with *self-incriminating proofs* where reconstruction/decryption is sequential.

*Most of this work was done while the author was at ISTA.

The security of our scheme relies on a new knowledge assumption, which informally states that whenever parties that hold secret shares of an RSA modulus $N = pq$ jointly compute $2^{2^T} \bmod N$ for some large T (too large to perform a sequential computation of length T in MPC), then at least one of them must have learned a multiple of N .

1 Introduction

A *proof of knowledge* [BG93] convinces a verifier that the party that created the proof had access to a secret witness. Formally this is argued by the existence of an extractor that, given black-box access to a prover, can learn the secret. In practice, a prover might, however, be able to compute a convincing proof without actually knowing the secret. For example, the secret might be *secret-shared* among many parties, who then run a secure multiparty computation (MPC) protocol to compute the proof of knowledge, without any single party actually knowing the witness. Another possibility is to embed the secret in some trusted hardware, which restricts the way in which that secret can be used. For example, if the secret is a signing key, it might be possible to only sign particular messages (or the secret can only be used to compute a proof of knowledge).

Being able to perform actions without actually knowing the secret, which one would intuitively assume is required, undermines the security of many use cases. For example, to prevent *vote selling* in blockchains, the signing key used for voting is also linked to collateral set up by the voter, so that knowing the key can be used to withdraw the collateral. Intuitively, this should deter a rational voter from “selling” their secret key. This strategy fails if the seller supplies the key via an MPC protocol that allows signing votes, but not withdrawing the collateral.

(Sequential) proofs of complete/individual knowledge. Two concurrent papers tackle this issue using the terms *individual cryptography* [DFL23] and *proofs of complete knowledge* [KBD⁺24] (PoCK), respectively, to refer to settings where we want to guarantee that a *single party* holds some secret “in the clear” instead of multiple parties jointly holding it via MPC. Both papers also discuss many real-world settings where such a guarantee is required.

Both works propose a similar mechanism to construct proofs (or signatures) of individual/complete knowledge. The general idea is to force a prover to make a huge number of hash queries. These hash queries require inputs, and from a small fraction of these inputs, one can efficiently extract the secret. More concretely, in [KBD⁺24] one must hash many triples (a, c, z) of a sigma protocol with the same a but different challenges c . Sigma protocols have the property (“special soundness”) that from any two tuples with the same a but different c one can extract the secret. Thus given only two hash-function inputs, one can extract the secret.

The number of required hashes is chosen sufficiently large, so that it is completely infeasible to evaluate a significant fraction of these inputs entirely within MPC. Thus at least one party must have seen some inputs in the clear, from which the secret can be extracted. In other words, since an extractor exists, the party must have “known” the secret. To implement such schemes, [DFL23] and [KBD⁺24] suggest using (outdated)

Bitcoin mining hardware, which is specially designed to evaluate a massive number of SHA256 hashes at minimal hardware and electricity cost.

Recently *sequential* proofs of complete knowledge (SPoCK) have been proposed [DHP26]. In their construction, one must compute many hashes in a sequential way. As the depth of a computation is the major limiting factor for multiparty computation, one can choose the required number of hashes in a SPoCK much lower than in a standard PoCK, thus making this primitive available on standard hardware. The idea behind the SPoCK construction in [DHP26] is to take a PoCK similar to the one from [KBD⁺24], but embed the hash queries into the computation of the proof of sequential work from [CP18].

Secret-sharing with snitching. Apart from primitives that ensure that some secret is held in the clear by one party, individual cryptography [DFL23] suggests another type of primitive. In the setting where some computation is performed by multiple parties who hold shares of a secret, we need to ensure that whenever the parties compute a value that depends on the secret, at least one of the parties learns a certificate for this fact.

The most basic such primitive is *secret sharing with snitching* (SSS) [DFL23, DFLM24, BDF⁺25]. In the simplest n -out-of- n case, there are parties $\mathcal{A}_1, \dots, \mathcal{A}_n$ that do not all mutually trust each other, in the sense that for every party $\mathcal{A}_i$, there’s at least one $\mathcal{A}_j$ where $\mathcal{A}_i$ would prefer to “punish” $\mathcal{A}_j$ if it can profit from doing so.

An SSS scheme is an n -out-of- n secret-sharing scheme, where a dealer splits the secret s into shares $s_1, \dots, s_n$, and party $\mathcal{A}_i$ receives share s_i . The additional security requirement is that whenever the n parties try to learn anything about s , at least one party will learn a certificate of this fact.

This is useful if we want to disincentivize the parties from reconstructing the secret when they should not, e.g., before some protocol-specified event happens. For example, the secret could be a testament, and the dealer only wants it reconstructed after death. To achieve this, one would set up the sharing in a way such that whenever the secret is reconstructed too early, some party $\mathcal{A}_j$ gets a “snitching proof”, which it can then use to “snitch” on another party and receive some reward, while the party that was snitched on will lose their collateral. We stress that it is not sufficient to punish reconstruction of the entire secret; the parties should also be disincentivized to use secure multiparty computation to compute a function of s or to just learn one bit of information that depends on s .

Self-incriminating proofs. This is another powerful primitive in the same spirit, introduced by [CDF⁺24]. Here $\mathcal{A}_1, \dots, \mathcal{A}_n$ hold shares of a decryption key for some public-key encryption scheme. The primitive ensures that, given a ciphertext, whenever the parties try to learn anything about the underlying plaintext, then one party will learn a certificate of this fact. This is motivated by its application to *encrypted mempools* [RK23]: in modern blockchain consensus protocols, transactions are included in new blocks by “validators”. However, by knowing the transactions before they are included, the validators can mount “front-running attacks” by including their own transactions in the block *before* the ones received (cf. MEV attacks [DGK⁺20]), which can generate profits for the validators.

An approach to preventing this is to encrypt the broadcast transactions and only decrypt them once the block is part of the blockchain and so it is too late to mount an

MEV attack. For this, the decryption key is distributed among parties and the assumption is that a certain fraction will behave honestly. However, if they are all rational and an attack results in profit, they would simply all collude and thresholdizing the decryption process does not help. If the validators, however, do not trust each other (in particular, they fear being snitched on by others), then decryption with self-incriminating proofs disincentivizes decryption before the protocol specifies it.

In the context of blockchains, rewarding the snitcher can easily be realized via a smart contract: the validators first deposit some collateral into the contract. If a snitching proof is submitted to it together with a wallet address (before the time when decryption should take place), then the contract transfers the collateral to the given wallet.

1.1 Overview of our contribution

Sequential secret-sharing with snitching. All existing constructions of SSS [DFL23, DFLM24, BDF⁺25] rely on hash functions and use the same basic idea outlined above. The reconstruction procedure is defined so that it requires a huge number of hash evaluations, while from a small number of hash function inputs, one can learn a snitching proof.

In this paper we ask whether we can make reconstruction sequential so we get improved security in the same way SPoCK improve on PoCK. Unfortunately, the blueprint from [DHP26], which embeds the computation into a proof of sequential work (PoSW), does not transfer to our context. The reason is that during the sharing phase we need to sample a PoSW puzzle *and* its solution. Unfortunately, no PoSW supports this, so computing the solution requires an expensive sequential computation (in contrast, it is easy to sample a PoW puzzle and solution efficiently without performing a lot of work).

One might object that we could simply allow the sharing phase to perform the expensive sequential computation to find the PoSW solution, but this does not work. The reason is that in SSS the dealer itself is untrusted, so it cannot perform the sharing solely on its own (otherwise, a malicious dealer could trivially snitch on honest parties). So sharing must happen entirely inside MPC to protect against a malicious dealer (or via a more elaborate sharing protocol [BDF⁺25]). Thus, if we were using PoSW, we would arrive at a contradiction: We need the sequential computation to be long enough that it essentially cannot be computed within MPC by the shareholders, but at the same time the dealer must be able to perform it in MPC to share the secret.

Our construction using RSW time-lock puzzles. Instead of PoSW, we turn to another primitive from *timed cryptography*, namely the original time-lock puzzles from Rivest, Shamir and Wagner [RSW96]. Crucially, these allow us to sample a puzzle and solution efficiently without a long sequential computation.

In these time-lock puzzles, the challenge is an RSA modulus $N = pq$ and one must compute $2^{2^T} \bmod N$ for some given large T . As the party that sampled the challenge knows the factorization $N = pq$, they can efficiently compute the solution by first computing the group order $\varphi(N) = (p-1)(q-1)$, then $e = 2^T \bmod \varphi(N)$ and finally $2^e \bmod N = 2^{2^T} \bmod N$ with one simple exponentiation modulo N . It is conjectured that without the factorization of N , given N and T , computing $2^{2^T} \bmod N$ requires T sequential squarings modulo n , i.e., the only way is to compute $2, 2^2, 2^{2^3}, 2^{2^4}, \dots, 2^{2^T}$ modulo N .

In our SSS scheme with a sequential reconstruction phase, to share a secret, we first sample $N = pq$, and then compute $y = 2^{2^T} \bmod N$. The shared random secret is then defined as $s = \text{Hash}(N, y)$. We use a standard n -out-of- n secret-sharing scheme to compute the shares $N_1, \dots, N_n \leftarrow \text{Share}(N)$ for the n parties. The scheme will output some additional values that facilitate the snitching functionality, which we will ignore for now.

Note that sharing a secret *does not* require T sequential steps, which is important as it is not executed by the dealer, but must be run within an MPC protocol between the dealer (who learns s) and the parties (who each learn their share N_i). The dealer can then use s to encrypt the actual secret they want to share. To reconstruct s , the parties $\mathcal{A}_1(N_1), \dots, \mathcal{A}_n(N_n)$ first reconstruct N , then compute $y = 2^{2^T} \bmod N$ and finally $s = \text{Hash}(N, y)$.

The choice of T depends on the application. We want to choose T large enough, so in the time frame relevant for the application, the only way to compute $2^{2^T} \bmod N$ is for some party to learn N , or some multiple αN of it, in the clear. Note that a party having αN can compute $y' = 2^{2^T} \bmod \alpha N$. Once y' is known, the parties can compute $2^{2^T} \bmod N = y' \bmod N$ within an MPC protocol.

Our main assumption is thus that whenever the parties learn any information about the secret, some party knows a multiple of N . The main challenge is to devise a way by which we can use this value to compute a snitching proof, which we outline below.

Choosing T . For the testament application discussed above, the time-frame where we want to prevent reconstruction without a party learning a snitching proof is years, or even decades, so we need a rather large T . At the same time, in this application it is acceptable if the honest reconstruction, i.e., when given N in the clear, can take some time, hours, or even days. As a rough estimate, with $T = 2^{32}$ and a 2048-bit modulus, it takes around one hour on standard hardware to compute $2^{2^T} \bmod N$. For the mentioned application to prevent MEV attacks, the reconstruction must be much faster, but here the relevant time-window is also just seconds or minutes, not decades.

Adding a snitching proof. Since knowing (a multiple of) N allows for computing the secret, our approach is to check whether a snitching proof π is a multiple of N . To this end, we consider a small RSA subgroup of unknown order [Gro05]. In particular, we sample an RSA modulus $R := PQ$ with the following structure: P and Q are primes of the form $P := 2ap + 1$ and $Q := 2bq + 1$ where a, b are appropriately large integers, and p, q are primes. This structure ensures that the RSA group $\mathbb{Z}_R^*$ has a unique subgroup of order $N = pq$. Let g be a generator of this subgroup, and publish (R, g) . It is then possible to check whether a snitching proof is a multiple of N , as one can verify that $\pi = \alpha N$ by checking whether $g^\pi \stackrel{?}{=} 1 \bmod R$.

One might be concerned that the public information R and g reveal information about N . This is captured by the *order assumption*, which states that given R and a (random) element g it is computationally hard to find (a multiple of) its order. The order assumption is implied by the factoring assumption [Mil76, Sho97], or by the strong RSA subgroup assumption [Gro05].

Summary of our contribution. We give the first secret sharing with snitching scheme based on guarantees from *sequential computation*. Sequentiality is useful as computational depth is what makes MPC expensive. Our scheme is based on the RSW time-lock puzzle [RSW96], making it also the first individual cryptography primitive whose hardness is *not based on hash functions*.

We justify that “snitching” is possible in our scheme under a new knowledge-style assumption, while both privacy and soundness (i.e., honest participants cannot be falsely accused) hold under existing standard assumptions. Making a new knowledge-style assumption seems necessary as individual cryptography is a new model, which calls for new kinds of assumptions. We hope that our assumption receives more scrutiny and cryptanalysis, and that it finds use in other areas within individual cryptography.

1.2 Other related work

Strong secret sharing with snitching [BDF⁺25] extends SSS [DFLM24] by allowing for an *arbiter* as part of the adversary, which is able to compensate parties in case they get punished. Additionally, they strengthen the notion of punishability in order to disincentivize self-punishment.

Traceable secret sharing introduced by Goyal et al. [GSS21], later refined [BPR24] and revisited [GJP26], aims to address the problem of “selling” one’s shares using a reconstruction box (i.e., using tamper-proof hardware or an obfuscated program). Traceability guarantees that one can provably identify the parties that were involved in creating the reconstruction box, given black-box access to the box. This line of work does not consider attacks leveraging MPC, however.

Deniable secret sharing introduced by Canetti et al. [CDK⁺25] extends secret sharing in a different manner: instead of proving participation in reconstruction, the aim is to make this participation undetectable. This undetectability has to hold even with regard to the adversary that gathers the shares to reconstruct.

Rational secret sharing [HT04] considers the converse problem: it studies how shareholders can be incentivized to reconstruct a secret. In particular, they show that standard reconstruction protocols are not incentive-compatible with rational agents, as they would rather wait to receive other shares before sending their own.

2 Preliminaries

2.1 Notation

We denote the security parameter by λ . We use $y \leftarrow \text{Alg}(x)$ to denote the output of a randomized algorithm, and $y := \text{Alg}(x)$ to denote the output of a deterministic one; more generally $:=$ denotes assignment and $=$ equality. We define $[n] := \{1, \dots, n\}$ and for a set S of size $|S| = n$ denote its elements by $S_1, \dots, S_n$. To sample an element uniformly at random from S we write $s \leftarrow_{\$} S$.

We abbreviate (probabilistic) polynomial time by (P)PT, and use the shorthand $\text{negl}(\cdot)$ for a negligible function. **Hash** is a hash function, which we model as a random oracle. We abbreviate random oracle (model) by RO(M). Euler’s totient function is denoted by $\varphi(\cdot)$.

2.2 Adversarial model

We consider a distributed adversary $\mathcal{A} := (\mathcal{A}_1, \dots, \mathcal{A}_\ell)$ for $\ell \in \mathbb{N}$ that consists of multiple PPT *sub-adversaries* $\mathcal{A}_1, \dots, \mathcal{A}_\ell$. These sub-adversaries participate in an interactive protocol Π , for which by $\Pi(1^\lambda, x; y_1, \dots, y_\ell) \rightarrow o$ we denote that this protocol on shared input x , and private inputs $y_1, \dots, y_\ell$ outputs o . We put no restrictions on Π , so it may be an MPC protocol that reveals no information apart from its output. The *view* of sub-adversary $\mathcal{A}_i$ in Π is denoted by $\text{view}(\mathcal{A}_i)$. It contains $\mathcal{A}_i$'s inputs, its random coins, and all messages it has received in Π .

2.3 Secret sharing

Our construction uses an information-theoretic protocol for secret sharing, e.g., Shamir's scheme [Sha79].

Definition 1 (Secret sharing). Let $1 \leq t \leq n$. A secret sharing scheme SS_n^t is defined via a tuple of PPT algorithms ($\text{Share}, \text{Reconstruct}$) and a secret space $\mathcal{S}$:

- $\text{Share}(1^\lambda, s) \rightarrow (s_1, \dots, s_n)$ takes as input a secret s from the secret space $\mathcal{S}$, and outputs n shares.
- $\text{Reconstruct}(1^\lambda, s_1, \dots, s_t) \rightarrow s$ takes as input t shares, and outputs a secret s .

A scheme SS_n^t must satisfy the following properties:¹

- **Perfect correctness:** For any $s \in \mathcal{S}$ and $I \subseteq [n]$ with $|I| \geq t$, it holds that

$$\Pr \left[s = s' \mid \begin{array}{l} (s_1, \dots, s_n) \leftarrow \text{Share}(s) \\ s' \leftarrow \text{Reconstruct}(s_{I_1}, \dots, s_{I_{|I|}}) \end{array} \right] = 1.$$

- **Perfect privacy:** For any unbounded adversary $\mathcal{A}$ and any $I \subseteq [n]$ with $|I| < t$ it holds that

$$\Pr \left[s^* = s \mid \begin{array}{l} s \leftarrow \mathcal{S} \\ (s_1, \dots, s_n) \leftarrow \text{Share}(s) \\ s^* \leftarrow \mathcal{A}(1^\lambda, s_{I_1}, \dots, s_{I_{|I|}}) \end{array} \right] \leq \frac{1}{|\mathcal{S}|}.$$

2.4 Number-theoretic assumptions

Definition 2 (Prime sampler). For $\ell \geq 1$, a prime sampler $\text{Prime}(1^\lambda, \ell) \rightarrow (p_1, \dots, p_\ell)$ denotes an algorithm that samples ℓ random, distinct λ -bit primes.

To facilitate snitching, we will work in a small RSA subgroup of unknown order. The following defines how we generate a group $\mathbb{Z}_R^*$ together with a generator g of a small subgroup of order $N = pq$. Given g and an integer M , one can check efficiently whether $N \mid M$ since $g^M = 1 \bmod R$ if and only if $N \mid M$. To see this, consider $M = kN + d$ for an integer k and $0 \leq d < N$; then $g^M = g^d = 1 \bmod R$ if and only if $d = 0$ as it is strictly smaller than g 's order.

SubgroupGen(1^λ) :

```

01 // Sample RSA modulus
02 While  $P$  not prime or  $Q$  not prime:
03    $(p, q) \leftarrow \text{Prime}(1^\lambda, 2)$ 
04    $(a, b) \leftarrow \text{Prime}(1^{2\lambda}, 2)$ 
05    $P := 2ap + 1, Q := 2bq + 1$ 
06    $R := PQ$ 
07
08 // Find generator of subgroup
09 While  $g^p = 1 \bmod R$  or  $g^q = 1 \bmod R$ :
10    $z \leftarrow \mathbb{Z}_R^*$ 
11    $g := z^{2ab \bmod \phi(R)} \bmod R$ 
12 return  $(p, q, P, Q, g)$ 

```

Figure 1: Generation of a (small) RSA subgroup.

Definition 3 (Subgroup sampler). SubgroupGen outputs p, q, P, Q and g such that g generates a subgroup of $\mathbb{Z}_{PQ}^*$ of order pq . It is defined in Fig. 1.

Note that for $g := z^{2ab}$ we have that $g^{pq} = z^{2abpq} = z^{(P-1) bq} = 1 \bmod P$ and similarly $g^{pq} = z^{2abpq} = z^{(Q-1) ap} = 1 \bmod Q$ and by the Chinese remainder theorem the order is a divisor of pq . Since p and q are distinct primes, the check $g^p = 1 \bmod R$ ensures that the order is not p or 1, while the check $g^q = 1 \bmod R$ ensures the order is not q . Hence g has order N , as desired.

The following assumption implies privacy and punishability of our scheme. It is implied by the strong RSA subgroup assumption [Gro05], as any M that's a non-zero multiple of the order of g gives the nontrivial root relation $g^{M+1} = g$, with root g and exponent $M + 1 > 1$. It is also implied by the factoring assumption [Mil76, Sho97], as finding an even order M of a random group element g implies that $\gcd(g^{M/2} - 1, R)$ yields a non-trivial factor of R with good probability.

Assumption 1 (Order finding). We assume that for any PPT adversary $\mathcal{A}$ the following holds:

$$\Pr \left[M > 0 \wedge g^M = 1 \bmod R \mid \begin{array}{l} (\cdot, \cdot, P, Q, g) \leftarrow \text{SubgroupGen}(1^\lambda) \\ M \leftarrow \mathcal{A}(1^\lambda, R := PQ, g) \end{array} \right] \leq \text{negl}(\lambda).$$

The sequentiality of our construction is based on the iterated squaring assumption introduced in [RSW96]. In the algebraic group model [FKL18], the iterated squaring assumption is implied by factoring [KLX20, Rot22].

¹We often omit the input 1^λ for brevity.

Assumption 2 (Iterated squaring). The *iterated squaring assumption* holds for **Prime** (cf. Def. 2) if, for any PPT algorithm $\mathcal{A}$ with $\text{poly}(\lambda)$ parallelism running in time less than T , it holds that

$$\Pr \left[\mathcal{A}(1^\lambda, N) \rightarrow 2^{2^T} \in \mathbb{Z}_N^* \mid \begin{array}{l} (p, q) \leftarrow \text{Prime}(1^\lambda, 2) \\ N := pq \end{array} \right] \leq \text{negl}(\lambda).$$

Remark 1. Strictly speaking, the iterated squaring assumption as in Ass. 2 does not hold. Using at least T^2 parallelism, it is possible to compute 2^{2^T} in time $T/\log \log T$ [BS07]. In practice, where, e.g., $T = 2^{32}$, this leads to a small $6\times$ speed-up while requiring 2^{64} parallelism, which is impractically large. Hence, for simplicity, we ignore the $\log \log T$ factor.

3 Random secret sharing with snitching

3.1 Definition

A random secret sharing with snitching scheme allows a dealer to share a random secret with a set of parties. If the parties reconstruct the secret, at least one party can produce a proof of this fact. This is the “snitching proof”. We model the existence of such a proof by virtue of an extractor: Whenever the parties learn at least one bit of information about the secret, we can extract a snitching proof from the view of one party (cf. “Punishability” in Def. 4).

We require these proofs to be *sound*, that is, it is hard to accuse (i.e., “frame”) a party even though they did not participate in the reconstruction (i.e., finding a valid “snitching proof” for a party is hard unless the party participates). Apart from Punishability, we want our scheme to be correct and private like a regular secret sharing scheme. Our definition is inspired by the one given by Dziembowski et al. [DFLM24]. However, our presentation deviates from theirs (for details, see § A).

Definition 4 (Random secret sharing with snitching). A t -out-of- n random secret sharing with snitching scheme **rSSS** is defined via a tuple of (P)PT algorithms (**Share**, **Reconstruct**, **VerifySnitch**) and a secret space $\mathcal{S}$.

- **Share** $(1^\lambda) \rightarrow (s_1, \dots, s_n, \text{vsni})$ outputs n shares $s_1, \dots, s_n$ of a random secret $s \in \mathcal{S}$ as well as some additional information **vsni** (used to verify snitching proofs).²
- **Reconstruct** $(1^\lambda, s_{i_1}, \dots, s_{i_t}) \rightarrow (s, \text{sni})$ takes as input t distinct shares, and outputs a secret $s \in \mathcal{S}$ together with a snitching proof **sni**.
- **VerifySnitch** $(1^\lambda, \text{vsni}, I, \text{sni}) =: b$ takes as input the additional information **vsni**, a set of indices $I \subseteq [n]$, and a snitching proof **sni**, and deterministically outputs a bit $b \in \{0, 1\}$.

An **rSSS** scheme has to satisfy the following properties:

²**Share** could also directly output s to the dealer; we omit it for readability. In this case, **Hash** should be an MPC-friendly hash function if **Share** must be run in MPC due to the dealer potentially being malicious.

- **Correctness:** For any $I, I' \subseteq [n]$ with $|I| = |I'| = t$ it holds that

$$\Pr \left[s = s' \mid \begin{array}{l} (s_1, \dots, s_n, \cdot) \leftarrow \text{Share}(1^\lambda) \\ (s, \cdot) \leftarrow \text{Reconstruct}(s_{I_1}, \dots, s_{I_t}) \\ (s', \cdot) \leftarrow \text{Reconstruct}(s_{I'_1}, \dots, s_{I'_t}) \end{array} \right] = 1.$$

- **Privacy:** Let $\mathcal{A}$ be a PPT adversary. For any $\ell < t$ and any subset $I \subseteq [n]$ of size $|I| = \ell$, we have

$$\Pr \left[\mathcal{A} \left(1^\lambda, s_{I_1}, \dots, s_{I_\ell}, \text{vsni}, s^{(b)} \right) \rightarrow b \mid \begin{array}{l} b \leftarrow_{\$} \{0, 1\} \\ (s_1, \dots, s_n, \text{vsni}) \leftarrow \text{Share}(1^\lambda) \\ s^{(1)} \leftarrow \text{Reconstruct}(s_1, \dots, s_t) \\ s^{(2)} \leftarrow_{\$} \mathcal{S} \end{array} \right] \leq \frac{1}{2} + \text{negl}(\lambda).$$

- **Punishability:** Consider a distributed adversary (§ 2.2) $\mathcal{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$. If

$$\Pr \left[b^* = b \mid \begin{array}{l} b \leftarrow_{\$} \{0, 1\} \\ (s_1, \dots, s_n, \text{vsni}) \leftarrow \text{Share}(1^\lambda) \\ s^{(1)} \leftarrow \text{Reconstruct}(s_1, \dots, s_t) \\ s^{(2)} \leftarrow_{\$} \mathcal{S} \\ b^* \leftarrow \Pi(1^\lambda, s^{(b)}, \text{vsni}; s_1, \dots, s_n) \end{array} \right] := \frac{1}{2} (1 + \eta(\lambda))$$

for $\eta(\lambda) > 0$, then there exists a PPT extractor Ext such that

$$\Pr \left[\begin{array}{l} \exists i \in [k] \\ I \subseteq [n] \text{ s.t. } |I| \leq t, \\ \text{Ext}(\text{view}(A_i)) \rightarrow \text{sni}, \text{ and} \\ \text{VerifySnitch}(\text{vsni}, I, \text{sni}) = 1 \end{array} \mid \begin{array}{l} b \leftarrow_{\$} \{0, 1\} \\ (s_1, \dots, s_n, \text{vsni}) \leftarrow \text{Share}(1^\lambda) \\ s^{(1)} \leftarrow \text{Reconstruct}(s_1, \dots, s_t) \\ s^{(2)} \leftarrow_{\$} \mathcal{S} \\ b^* \leftarrow \Pi(1^\lambda, s^{(b)}, \text{vsni}; s_1, \dots, s_n) \end{array} \right] \geq \eta(\lambda) - \text{negl}(\lambda).$$

- **Monotonicity of VerifySnitch:** If $\text{VerifySnitch}(\text{vsni}, I, \text{sni}) = 1$, then we also have $\text{VerifySnitch}(\text{vsni}, I', \text{sni}) = 1$ for any $I' \subseteq I$.
- **Unframability:** Let $\mathcal{A}$ be a PPT adversary. For any $i \in [n]$, we have

$$\Pr \left[\text{VerifySnitch}(\text{vsni}, \{i\}, \text{sni}^*) = 1 \mid \begin{array}{l} (s_1, \dots, s_n, \text{vsni}) \leftarrow \text{Share}() \\ \text{sni}^* \leftarrow \mathcal{A} \left(s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n, \text{vsni} \right) \end{array} \right] \leq \text{negl}(\lambda).$$

3.2 n -out-of- n construction

We construct an n -out-of- n random secret sharing scheme **rSSS** in Fig. 2 with secret space $\mathcal{S} := \{0, 1\}^\ell$.

rSSS.Share uses **SubgroupGen** to sample two λ -bit primes p, q such that $P = 2ap + 1$ and $Q = 2aq + 1$ are 3λ -bit primes (for appropriate integers a, b), and g is a generator of a subgroup of $\mathbb{Z}_{PQ}^*$ of order pq . Let $N := pq$; then the modulus N is secret-shared using an information-theoretic n -out-of- n secret sharing scheme, such as Shamir's scheme. The random secret is defined as

$$s := \text{Hash}(N, 2^{2^T} \bmod N). \quad (1)$$

Note that **Share** can compute the order $\varphi(N) = (p-1)(q-1)$, so the exponentiation can be efficiently carried out by reducing the exponent $2^T \bmod \varphi(N)$. The snitching information output by **Share** is $\text{vsni} := (R := PQ, g)$.

rSSS.Reconstruct is the honest reconstruction algorithm. It reconstructs N given the parties' shares, computes $2^{2^T} \bmod N$ by iterated squaring and reducing modulo N in sequential time T , and recovers the secret s as in (1). It returns s together with the snitching proof $\text{sni} := N$.

rSSS.VerifySnitch takes as input the snitching proof sni together with the snitching information $\text{vsni} = (R, g)$ and checks if sni is a multiple of N by checking whether $g^{\text{sni}} = 1 \bmod R$.

3.3 Handling malicious dealers

Sharing inside MPC. A malicious dealer may frame any party, since it knows the snitching value N . One way to avoid this is by running **rSSS.Share** as an MPC protocol between the dealer and the other parties. For this to work, computing **rSSS.Share** cannot perform too much sequential computation. In our scheme, this is possible since **Share**'s runtime is independent of the parameter T .

Independence of T enables running **Share** within MPC, but using off-the-shelf MPC frameworks will be expensive in practice. While modern MPC protocols generate RSA keys efficiently (e.g., [BDF⁺23]), they do not generate semiprimes with our required structure (cf. Def. 3). While one could use the approach of [ACS02], it is unfortunately less efficient than modern counterparts.

We suggest using offline precomputation for MPC, and also using shorter primes if the application allows it; the only requirement is that factoring must take longer than the punishability period needed by the application. Note that in this case, we advise using the information-theoretic privacy technique mentioned below. We leave it to future work to design an improved MPC protocol.

Alternative to sharing inside MPC. There is an alternative to running **Share** inside MPC by the following protocol [BDF⁺25]: The dealer generates κ many SSS instances, the shareholders privately fetch half of them (using MPC), and an overall snitching proof is a sufficiently large set of instance snitching proofs for a subset of this half. A malicious dealer cannot guess such a subset with high probability if κ is large.

```

01 // Parameters
02  $\mathcal{S} = \{0, 1\}^\ell$  is the secret space
03  $\text{SS} := \text{SS}_n^n$  is an  $n$ -out-of- $n$  secret sharing scheme (cf. Def. 1)
04  $\text{Hash} : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  is a hash function modelled as a random oracle
05  $T$  controls the hardness of the sequential computation
06
07  $\text{rSSS.Share}() \rightarrow ((N_1, \dots, N_n), \text{vsni} = (R, g))$ :
08    $p, q, P, Q, g \leftarrow \text{SubgroupGen}(1^\lambda)$ 
09    $N := pq, R := PQ$ 
10   // Secret sharing
11    $N_1, \dots, N_n \leftarrow \text{SS.Share}(N)$ 
12    $y := 2^{2^T \bmod \varphi(N)} \bmod N$  // this is efficient
13   // The secret is  $s := \text{Hash}(N, y)$ 
14   Output  $(N_1, \dots, N_n)$ , and  $\text{vsni} = (R, g)$ 
15
16  $\text{rSSS.Reconstruct}(N_1, \dots, N_n) \rightarrow (s, \text{sni})$ :
17    $N := \text{SS.Reconstruct}(N_1, \dots, N_n)$ 
18   Compute  $y := 2^{2^T} \bmod N$  using iterated squaring // this is slow
19   Output  $s := \text{Hash}(N, y)$  and  $\text{sni} := N$ 
20
21  $\text{rSSS.VerifySnitch}(\text{vsni} = (R, g), I, \text{sni}) \rightarrow \{0, 1\}$ :
22   If  $I \not\subseteq [n]$ , or  $g^{\text{sni}} \neq 1 \bmod R$ , output 0
23   Else, output 1

```

Figure 2: n -out-of- n random secret sharing scheme.

While their protocol and analysis are tailored to the PoW setting, we believe that their ideas can be adapted to our scheme. Since κ needs to be large, **Share**'s runtime being independent of T is needed in order for the dealer to be practically efficient. We leave it to future work to see if the ideas are indeed applicable.

3.4 Remarks and extensions

t -out-of- n secret sharing with snitching. We focus on the n -out-of- n case, since prior work [DFLM24] has shown that any 2-out-of-2 secret sharing with snitching can be generically compiled into a t -out-of- n one.

Non-random secret sharing. As mentioned above, the scheme can be easily transformed to allow for a user-specified secret $\hat{s}$ (instead of the randomly chosen s) by using s as a one-time pad for $\hat{s}$.

Information-theoretic privacy. One can obtain information-theoretic privacy by employing an information-theoretic secret-sharing scheme as follows: use the information-theoretically secure secret-sharing scheme to share s_1 together with our construction

sharing secret s_2 , and define the secret $s := s_1 \oplus s_2$. Breaking the computational secret-sharing scheme (and thus obtaining information about s_2) gives no information about s .

Verifiable reconstruction. As written, we presume that `rSSS.Reconstruct` is run by a single party in our scheme. This means that the other parties either need to trust this party, or re-run `Reconstruct` themselves to verify the output (but this is costly due to the iterated squaring).

Our scheme can be made verifiable by using a proof of exponentiation (e.g., [Pie19, Wes20, HHK⁺22]): All parties reconstruct N on their own; then, one party computes $y = 2^{2^t} \bmod N$ together with a proof that it performed the exponentiation correctly. The other parties can then simply recover $s := \text{Hash}(N, y)$.

Threshold encryption. Our `rSSS` can also be used to construct threshold public-key encryption with self-incriminating proofs [CDF⁺24]. The high-level idea is using the secret s as a key for a symmetric encryption scheme (so *random* SSS suffices) where the party that encrypts a message acts as the `rSSS` dealer.

In more detail, we assume that the decrypters each have a key pair for a standard public-key encryption scheme. To encrypt a message, the encrypter plays the dealer in our random secret-sharing scheme with snitching: this yields shares $s_1, \dots, s_n$, a random secret s and `vsni`; the encrypter then uses s as a symmetric key to encrypt the message, yielding a ciphertext C ; it then encrypts each share s_i under the i -th decrypter's public key, yielding C_i . The ciphertext is defined as $(C, C_1, \dots, C_n, \text{vsni})$.

To decrypt a ciphertext, the decrypters first retrieve their individual shares of the secret s from the ciphertexts C_i , and reconstruct s . Using s , they decrypt C , which yields the message. Note that this scheme achieves the unframability notion defined in [CDF⁺24], which assumes that the encrypter does not collude with decrypters to frame others. (The authors also explain why this is a reasonable assumption in the intended applications to blockchain protocols.)

4 Punishability of our scheme

We will now justify why we believe that our scheme satisfies Punishability. For this, we will need to make some new assumptions, which are effectively knowledge assumptions against resource-bounded adversaries, as required by individual cryptography.

Plugging the scheme from § 3.2 into the definition of Punishability in Def. 4 (for the threshold $t = n$) leads to the following ad hoc assumption, which is thus equivalent to punishability of our scheme.

Assumption 3 (Punishability of our scheme). Let `SubgroupGen` be a subgroup sampler (cf. Def. 3) and SS_n^n be an n -out-of- n secret sharing scheme (cf. Def. 1). Consider a

distributed adversary (§ 2.2) denoted by $\mathcal{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$. If

$$\Pr \left[b = b^* \mid \begin{array}{l} b \leftarrow_{\$} \{0, 1\} \\ (p, q, P, Q, g) \leftarrow \text{SubgroupGen}(1^\lambda) \\ N := pq, R := PQ \\ (N_1, \dots, N_n) \leftarrow \text{SS}_n^n.\text{Share}(N) \\ s^{(0)} := \text{Hash}(N, 2^{2^T} \bmod N) \\ s^{(1)} \leftarrow_{\$} \{0, 1\}^\ell \\ b^* \leftarrow \Pi(1^\lambda, R, g, s^{(b)}; N_1, \dots, N_n) \end{array} \right] := \frac{1}{2}(1 + \eta(\lambda))$$

for $\eta(\lambda) > 0$, then there exists a PPT extractor Ext such that

$$\Pr \left[\begin{array}{l} \exists i \in [k] \text{ such that} \\ N^* \leftarrow \text{Ext}(\text{view}(\mathcal{A}_i)) \\ \text{and } N \mid N^* \end{array} \mid \begin{array}{l} b \leftarrow_{\$} \{0, 1\} \\ (p, q, P, Q, g) \leftarrow \text{SubgroupGen}(1^\lambda) \\ N := pq, R := PQ \\ (N_1, \dots, N_n) \leftarrow \text{SS}_n^n.\text{Share}(N) \\ s^{(0)} := \text{Hash}(N, 2^{2^T} \bmod N) \\ s^{(1)} \leftarrow_{\$} \{0, 1\}^\ell \\ b^* \leftarrow \Pi(1^\lambda, R, g, s^{(b)}; N_1, \dots, N_n) \end{array} \right] \geq \eta(\lambda) - \text{negl}(\lambda).$$

At the core of our justification of Ass. 3 lies the simpler assumption Ass. 4 stated below. The primary observation is that Ass. 3 requires $\mathcal{A}$ to do better than guessing in distinguishing whether it received a $\text{Hash}(N, 2^{2^T} \bmod N)$ or a uniformly random bit string. In the random oracle model, to do better than guessing, $\mathcal{A}$ must at some point query $(N, 2^{2^T} \bmod N)$ to Hash . This in turn means that it must have computed $2^{2^T} \bmod N$.

We now want to extract a snitching proof whenever $\mathcal{A}$ computes $2^{2^T} \bmod N$. Using this value as a proof itself is not possible since $\mathcal{A}$ is a distributed adversary with access to MPC, so $\mathcal{A}$ may compute $2^{2^T} \bmod N$ and then hash it in MPC. Our idea is thus not to use $2^{2^T} \bmod N$, but the modulus N itself as a proof.

This justification in the ROM leads to the form of our assumption given in Ass. 4 below, which is now a search assumption. It states that computing $2^{2^T} \bmod N$ requires one party to learn a multiple αN of the modulus (here $\alpha \in \{1, 2, \dots\}$). This means that αN can be extracted from some party.

Assumption 4 (The simplest assumption). If

$$\Pr \left[2^{2^T} \bmod N \leftarrow \Pi(1^\lambda; N_1, \dots, N_n) \mid \begin{array}{l} (p, q) \leftarrow \text{Prime}(1^\lambda, 2) \\ N := pq \\ (N_1, \dots, N_n) \leftarrow \text{SS}_n^n.\text{Share}(N) \end{array} \right] := \eta(\lambda),$$

then there exists a PPT extractor Ext such that

$$\Pr \left[\begin{array}{l} \exists i \in [k] \text{ such that} \\ N^* \leftarrow \text{Ext}(\text{view}(\mathcal{A}_i)) \\ \text{and } N \mid N^* \end{array} \middle| \begin{array}{l} (p, q) \leftarrow \text{Prime}(1^\lambda, 2) \\ N := pq \\ (N_1, \dots, N_n) \leftarrow \text{SS}_n^n.\text{Share}(N) \\ y \leftarrow \Pi(1^\lambda; N_1, \dots, N_n) \end{array} \right] \geq \eta(\lambda) - \text{negl}(\lambda).$$

Justification of the simplest assumption. By the standard iterated squaring assumption (Ass. 2), computing $2^{2^T} \bmod N$ requires T sequential computation steps. Intuitively, we think of it as T squarings modulo N . This is not strictly true in practice as an algorithm could skip, e.g., every second modular reduction. Still, it is a useful intuition as the algorithm cannot skip too many modular reductions, since the intermediate values grow quickly.

We appeal to the iterated squaring assumption (Ass. 2) to justify that the entire computation of $2^{2^T} \bmod N$ cannot be carried out within MPC for reasonably set T . For example, computing $T = 2^{32}$ squarings is practical on regular hardware, but seems infeasible in MPC (e.g., requiring $\approx T$ rounds, or close to T bootstrapping operations for MPC when based on FHE). As a consequence, most of the squarings need to be carried out by a single party, outside of MPC.

One might expect that the party performing these squarings needs to learn N , but this is not true. Computing 2^{2^T} without any modular reductions is not possible, as it is too large, but the modular reduction need not be modulo N . The parties could compute a random multiple αN via MPC, publicly output this “blinded” modulus αN , and then perform the expensive computation of $2^{2^T} \bmod \alpha N$ outside of MPC. The parties perform a final modular reduction

$$(2^{2^T} \bmod \alpha N) \bmod N = 2^{2^T} \bmod N$$

within MPC to recover the output. As a consequence of this, Ass. 4 only asks the extractor to extract a multiple αN of N .

As established earlier, αN can now be used as the snitching proof.

Proofs of privacy and unframability. We defer these proofs to § B.

Usage of Artificial Intelligence. The text was written by the authors; LLMs (Codex and Claude) were used for minor edits and literature search. Moreover, Codex suggested the RSA moduli of [Gro05] as a way to make a preliminary construction of ours viable.

Acknowledgements. This research was funded in whole or in part by the Austrian Science Fund (FWF) [10.55776/F85]. This work was funded by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG18002] and [10.47379/ICT25-075].

References

- [ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 417–432. Springer, Berlin, Heidelberg, August 2002.
- [BDF⁺23] Jakob Burkhardt, Ivan Damgård, Tore Kasper Frederiksen, Satrajit Ghosh, and Claudio Orlandi. Improved distributed RSA key generation using the miller-rabin test. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 2501–2515. ACM Press, November 2023.
- [BDF⁺25] Jan Bormet, Stefan Dziembowski, Sebastian Faust, Tomasz Lizurej, and Marcin Mielniczuk. Strong secret sharing with snitching. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part IV*, volume 16003 of *LNCS*, pages 295–327. Springer, Cham, August 2025.
- [BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *CRYPTO’92*, volume 740 of *LNCS*, pages 390–420. Springer, Berlin, Heidelberg, August 1993.
- [BPR24] Dan Boneh, Aditi Partap, and Lior Rotem. Traceable secret sharing: Strong security and efficient constructions. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*. Springer Nature Switzerland, 2024.
- [BS07] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit chinese remainder theorem. *Mathematics of Computation*, 76(257):443–454, 2007.
- [CDF⁺24] James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, Arup Mondal, and Esra Yeniaras. Detecting rogue decryption in (threshold) encryption via self-incriminating proofs. Cryptology ePrint Archive, Paper 2024/794, 2024.
- [CDK⁺25] Ran Canetti, Ivan Damgård, Sebastian Kolby, Divya Ravi, and Sophia Yakoubov. Deniable secret sharing. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025, Part II*, volume 16269 of *LNCS*, pages 399–427. Springer, Cham, December 2025.
- [CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 451–467. Springer, Cham, April / May 2018.
- [DFL23] Stefan Dziembowski, Sebastian Faust, and Tomasz Lizurej. Individual cryptography. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 547–579. Springer, Cham, August 2023.

- [DFLM24] Stefan Dziembowski, Sebastian Faust, Tomasz Lizurej, and Marcin Mielniczuk. Secret sharing with snitching. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 840–853. ACM Press, October 2024.
- [DGK⁺20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy*, pages 910–927. IEEE Computer Society Press, May 2020.
- [DHP26] Antonio Giulio D’Antona, Charlotte Hoffmann, and Krzysztof Pietrzak. SPoCK: Sequential proofs of complete knowledge. Cryptology ePrint Archive, Paper 2026/738, 2026.
- [FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Cham, August 2018.
- [GJP26] Vipul Goyal, Abhishek Jain, and Aditi Partap. Traceable secret sharing revisited. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology – EUROCRYPT 2026*, 2026.
- [Gro05] Jens Groth. Cryptography in subgroups of $\mathbb{Z}_n$. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 50–65. Springer, Berlin, Heidelberg, February 2005.
- [GSS21] Vipul Goyal, Yifan Song, and Akshayaram Srinivasan. Traceable secret sharing and applications. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 718–747, Virtual Event, August 2021. Springer, Cham.
- [HHK⁺22] Charlotte Hoffmann, Pavel Hubáček, Chethan Kamath, Karen Klein, and Krzysztof Pietrzak. Practical statistically-sound proofs of exponentiation in any group. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 370–399. Springer, Cham, August 2022.
- [HT04] Joseph Y. Halpern and Vanessa Teague. Rational secret sharing and multiparty computation: Extended abstract. In László Babai, editor, *36th ACM STOC*, pages 623–632. ACM Press, June 2004.
- [KBD⁺24] Mahimna Kelkar, Kushal Babel, Philip Daian, James Austgen, Vitalik Buterin, and Ari Juels. Complete knowledge: Preventing encumbrance of cryptographic secrets. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 2415–2429. ACM Press, October 2024.
- [KLX20] Jonathan Katz, Julian Loss, and Jiayu Xu. On the security of time-lock puzzles and timed commitments. In Rafael Pass and Krzysztof Pietrzak,

- editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 390–413. Springer, Cham, November 2020.
- [Mil76] Gary L. Miller. Riemann’s hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.
 - [Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019.
 - [RK23] Antoine Rondelet and Quintus Kilbourn. Mempool privacy: An economic perspective, 2023.
 - [Rot22] Lior Rotem. Revisiting the uber assumption in the algebraic group model: Fine-grained bounds in hidden-order groups and improved reductions in bilinear groups. In Dana Dachman-Soled, editor, *ITC 2022*, volume 230 of *LIPIcs*, pages 13:1–13:13. Schloss Dagstuhl, July 2022.
 - [RSW96] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, USA, 1996.
 - [Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.
 - [Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.
 - [Wes20] Benjamin Wesolowski. Efficient verifiable delay functions. *Journal of Cryptology*, 33(4):2113–2147, October 2020.

A Definitional differences to [DFLM24]

First, the definition of [DFLM24] explicitly models their resource assumption, that is, random oracle queries that are either “fast” (and local), or “slow” (and private – via MPC). Our definition does not prescribe the resource model.

Second, our definition of Unframability implicitly assumes that the dealer executing **Share** is honest, whereas [DFLM24] allows it to be malicious. However, assuming an honest dealer is without loss of generality since **Share** may be executed via MPC. Note that, in fact, [DFLM24, BDF⁺25] rely on MPC to achieve Unframability against malicious dealers.

Third, Dziembowski et al. explicitly consider trust assumptions by using the notion of “coalitions”, i.e., parties that do not snitch on each other. They do so by having one sub-adversary $\mathcal{A}_i$ represent one coalition that controls up to $t - 1$ shares. Punishability then requires that the coalition of $\mathcal{A}_i$ is able to punish another coalition $\mathcal{A}_j$. In contrast, we only implicitly talk about trust assumptions, but our definition is essentially equivalent to theirs. In our case, one sub-adversary $\mathcal{A}_i$ controls a single share, and our Punishability notion states that at least one $\mathcal{A}_i$ can produce a snitching proof. Together with our

Privacy notion, this implies that Punishability deters reconstruction as long as coalitions are of size strictly less than t , like in the case of [DFLM24].

Lastly, we only consider a randomized secret-sharing scheme. This is easily turned into a secret sharing scheme (where a given secret is shared by the dealer) by using s as a one-time pad for the given secret.

B Proofs of other properties

So far we have argued why the scheme presented in § 3.2 fulfills Punishability. We now prove the remaining properties.

Theorem 1. *The scheme presented in Fig. 2 satisfies Correctness and Monotonicity of VerifySnitch.*

Proof. The former follows from correctness of SS_n^n . The latter holds by construction. $\square$

Theorem 2. *In the random oracle model, if order finding is hard (Ass. 1), then the scheme presented in Fig. 2 satisfies Privacy.*

Proof. Assume that $\mathcal{A}$ breaks Privacy of rSSS with probability $1/2 + \eta(\lambda)$. We construct an adversary $\mathcal{B}$ using $\mathcal{A}$ such that $\mathcal{B}$ breaks the order finding assumption with probability $\eta(\lambda)$. $\mathcal{B}$ receives a modulus R and an element g and needs to output a multiple of the order of g . It now simulates the inputs that $\mathcal{A}$ expects:

- $s_{I_1}, \dots, s_{I_\ell}$: Since $|I| = \ell < n$, the values $s_{I_1}, \dots, s_{I_\ell}$ do not reveal any information about N by the perfect privacy of SS_n^n . Thus, $\mathcal{B}$ can replace them with uniformly random values.
- (R, g) : $\mathcal{B}$ uses its input (R, g) as the verification information.
- $s^{(b)}$: $\mathcal{B}$ samples $s^{(b)} \leftarrow_{\$} \{0, 1\}^\ell$ for $b \in \{0, 1\}$.

Then $\mathcal{B}$ runs $\mathcal{A}$ and answers $\mathcal{A}$'s RO queries as follows: for any query of the form $(N^*, \cdot)$, $\mathcal{B}$ checks whether $g^{N^*} = 1 \pmod R$ and $N^* \neq 0$. If so, it halts and outputs N^* ; otherwise it lazily samples the response $r \leftarrow_{\$} \{0, 1\}^\lambda$.

Now if $\mathcal{A}$ has a winning probability greater than $1/2$, it must make a query of the form $\text{Hash}(N, \cdot)$, as otherwise s is independent of all inputs that $\mathcal{A}$ received. Therefore, if $\mathcal{A}$ breaks privacy with probability $1/2 + \eta(\lambda)$, then $\mathcal{B}$ finds the order of g with probability $\eta(\lambda)$. $\square$

Theorem 3. *If order finding is hard (Ass. 1), then the scheme presented in Fig. 2 satisfies Unframability.*

Proof. Assume that $\mathcal{A}$ breaks Unframability of rSSS with advantage $\eta(\lambda)$. We construct an adversary $\mathcal{B}$ using $\mathcal{A}$ such that $\mathcal{B}$ finds (a multiple of) the order of g with probability $\eta(\lambda)$.

$\mathcal{B}$ receives (R, g) and simulates the shares s_j for $j \in [n] \setminus \{i\}$ as in the proof of Thm. 2. Now when $\mathcal{A}$ outputs sni^* , then $\mathcal{B}$ halts and outputs sni^* .

Whenever $\mathcal{A}$ wins Unframability, $\text{sni}^* \neq 0$ and $g^{\text{sni}^*} = 1 \pmod R$, and hence $\mathcal{B}$ finds a multiple of the order of g with the same probability as $\mathcal{A}$. $\square$