Tighter Control for Distributed Key Generation: Share Refreshing and Expressive Reconstruction Policies

Sara Montanari[1], Riccardo Longo[2], and Alessio Meneghetti[3]

Abstract. The secure management of private keys is a fundamental challenge, particularly for the general public, as losing these keys can result in irreversible asset loss. Traditional custodial approaches pose security risks, while decentralized secret sharing schemes offer a more resilient alternative by distributing trust among multiple parties. In this work, we extend an existing decentralized, verifiable, and extensible cryptographic key recovery scheme based on Shamir's secret sharing. We introduce a refresh phase that ensures proactive security, preventing long-term exposure of secret shares. Our approach explores three distinct methods for refreshing shares, analyzing and comparing their security guarantees and computational complexity. Additionally, we extend the protocol to support more complex access structures, with a particular focus on threshold access trees, enabling fine-grained control over key reconstruction.

Keywords: secret sharing · decentralized key management · proactive security · threshold access trees · access policies.

1 Introduction

Decentralized systems are increasingly being seen as an attractive alternative to centralized ones, due to their benefits in data management, such as avoiding single points of failure or securely storing crypto-assets. In this context, secret sharing is used in wallet key management to improve the security and accessibility of private keys by splitting them into multiple shares, distributed among different participants or providers. No single participant holds the complete key, and access to the wallet requires a minimum number of shares to reconstruct it. An example of this application is illustrated in [14], where a decentralized wallet model is described from an engineering standpoint.

Similarly, key recovery schemes allow to reconstruct a key when some shares are lost. These schemes also exploit secret sharing and decentralized key generation, and enhancing these primitives directly translates to enabling new features or improving security in many application scenarios. For example, [4] shows how a distributed key generation scheme with some extensibility can be exploited to build a key recovery infrastructure where an emergency party may remain offline even during the enrollment of new clients, and come into play for key reconstruction only when the client loses their shares.

[1] University of Trento, Trento, Italy. Email: `sara.montanari-1@unitn.it`
[2] Fondazione Bruno Kessler, Trento, Italy. Email: `rlongo@fbk.eu`
[3] University of Trento, Trento, Italy. Email: `alessio.meneghetti@unitn.it`

Related Works The prolific field of *Secure Secret Sharing* has been pioneered by Shamir in 1979 [13], opening the door to countless applications in many fields of cryptography. In 1987, Feldman [5] extended this primitive by adding verifiability of shares, then the first step towards decentralization came in 1991, when [11] introduced a 2-round Decentralized Key Generation scheme (DKG), where each participant acts as a dealer in a Feldman VSS protocol. Quite a few years later [6] demonstrated a weakness in [11], and proposed a secure variant by adding a commitment step (3-round DKG). The application of DKGs to threshold signatures sparked new life into the field: FROST [8] uses a variant of Pedersen DKG called PedPop, where each participant proves the knowledge of their key by using a ZKP. In [3] is presented variant of PedPoP (Simpl.PedPoP) where dishonest participants trigger the abort of the protocol. Our starting point is [1], which introduces the *extensibility* property, that is new shares can be created after generation (by any threshold of parties), which are verifiably compatible with old ones.

The key reconstruction policy of all these schemes is a simple t-out-of-n threshold. In general, one may want to use more expressive policies. As far as we know, no prior work has ever integrated more complex policies into decentralized key generation.

Another important addition for various multi-party protocols is *proactive security*. An established technique to achieve it is to introduce a refresh phase [15,16]. However, to the best of our knowledge, no prior work has applied this approach to a protocol that is simultaneously decentralized, verifiable, extensible and applicable to any access policy. This gap is the motivation behind our work, that aims to ensure that security guarantees are maintained over time while preserving all the features listed above.

Our Contribution We extend the protocol by integrating more complex access structures that define the sets of participants authorized to reconstruct the secret. Among these, the most expressive and encompassing structures are *threshold access trees*. There participants are represented as leaves, and each internal node enforces a threshold condition specifying the minimum number of child nodes required for activation. With this access structures is possible to compactly express any monotone access policy, that is any policy which can be described by a boolean formula with an arbitrary combinations of `AND` and `OR`. The term *monotone* comes from the fact that if A is an authorized set of user and $A' \supseteq A$, then A' is also an authorized set of users [7].

Given their expressive power, we focus on extending each phase of the secret sharing protocol to support threshold access trees. In [9], the theory of monotone span programs is used to derive an LSSS matrix implementing the policy described by a threshold access tree. Our work bridges the gap in the literature by offering a more intuitive proof and integrating the results in the secret sharing protocol. We follow a recursive strategy for describing the generator matrix that encodes the secret vector into the final shares of the tree, extending Shamir's strategy layer-by-layer.

Moreover, we analyze the introduction of a refresh phase to the protocol, ensuring *proactive security*, i.e. the maintenance of security thresholds over time. We consider a *snapshot*, *mobile* and *adaptive* adversary and we propose three distinct methods.

The first method for refreshing the shares is ZeroSharesRefresh and involves generating and adding a polynomial with a zero constant term to the Shamir polynomial, while the second method RegenRefresh involves regenerating the Shamir polynomial while keeping the constant term fixed. These two strategies are shown to be equivalent and proactive-secure against the snapshot, mobile and adaptive adversary model.

The third method MatrixRefresh involves periodically updating the matrix that encodes the secret, exploiting the parallelism between MDS codes and secret sharing. This strategy, together with proactive security against the snapshot, mobile and adaptive adversary model, achieves also forward secrecy against an adversary that steals a sufficient number of old shares. However, unlike the first two methods, it is shown to be insecure against a *continuous-shot*, *non-mobile*, *adaptive* adversary.

Organization We start with some preliminaries in section 2, then in section 3 we present the results obtained by extending the protocol to Threshold Access Trees. Finally, in section 4 we present the integration of a refresh phase to achieve proactive security.

2 Preliminaries

In this section we introduce the notation used for our constructions, and recall all the necessary definitions and concepts.

2.1 Notation

We use a blackboard-bold font to indicate algebraic structures (i.e. sets, groups, rings, fields and elliptic curves). When speaking about a generic group $\mathbb{G}$, we use multiplicative notation unless stated otherwise. When describing data exchanged by two users we will use two indexes: the first denotes the sender, while the second denotes the receiver (i.e. the symbol $x_{i,j}$ denotes that the value was generated by party i and sent to party j). With an abuse of notation we sometimes say that a list is a subset of a set. In this context we simply mean that every element of that list is an element of the set.

In Section 3 background colors will be used to help the reader to navigate diagrams and formulas: parts highlighted with the same color are related or refer to the same things.

2.2 Link between Codes and Secret Sharing

Let α be an agreed-upon primitive element of $\mathbb{F}_q$ and let $p = \sum_{k=0}^{t-1} p_k x^k$, a polynomial of degree $t-1$ with coefficients $p_k \in \mathbb{F}_q$, and define $\beta_j = p(\alpha^j)$ with

$j \in 1, \dots, q-1$. In the context of Shamir Secret Sharing, $p(0) = p_0$ is the secret and β_j is a share.

Definition 1. *Let $J = [j_1, \dots, j_n]$ be a list of n distinct integers in the set $\{1, \dots, q-1\}$. We define the $t \times n$ matrix $G_J = \left[\alpha^{j \cdot k}\right]_{k \in \{0,\dots,t-1\},\, j \in J}$. If $n = 1$ then $J = [j]$ and we sometimes simply use G_j instead of $G_{[j]}$.*

The following proposition summarizes the properties of the matrix defined in Definition 1 and the link with Reed-Solomon codes. Moreover, since p has degree at most $t-1$, given any list $J \subseteq \{1, \dots, q-1\}$ of cardinality at least t, with the list of evaluations $[\beta_j]_{j \in J}$ it is possible to interpolate the polynomial p. This process can be executed exploiting the matrix G_J, as the proposition describes.

Proposition 1. *For any $t \le n \le q-1$ and for any $J = [j_1, \dots, j_n]$, the matrix G_J (constructed as in Definition 1) is the generator matrix of a punctured $[n, t]_q$ Reed-Solomon code. In particular:*

- *G_J has maximum rank for any $J = [j_1, \dots, j_n]$;*
- *if $n = t$ then G_J is invertible;*
- *if $J = [j_1, \dots, j_t]$ is a list of t distinct integers in $\{1, \dots, n\}$, then, $(p_0, \dots, p_{t-1}) = (\beta_{j_1}, \dots, \beta_{j_t}) \cdot G_J^{-1}$.*

For the proof see [1].

Remark 1. The vector containing all the shares $(\beta_1, \dots, \beta_n)$ can be viewed as the *codeword* obtained by encoding the secret vector $(p_0, \dots, p_{t-1})$ with the matrix $G_{[1,\dots,n]}$.

2.3 Access Structures and Linear Secret Sharing Schemes

Definition 2. *An access structure (respectively, monotone access structure) is a collection (respectively, monotone collection) $\mathbb{A}$ of nonempty subsets of $\{P_1, P_2, \dots, P_n\}$, i.e., $\mathbb{A} \subseteq 2^{\{P_1, P_2, \dots, P_n\}} \setminus \{\emptyset\}$. The sets in $\mathbb{A}$ are called the authorized sets, and the sets not in $\mathbb{A}$ are called the unauthorized sets.*

Any monotone access structure can be realized by a *linear secret sharing scheme*, as shown in [2].

Definition 3. *A secret sharing scheme $\Pi = (M, \rho)$ over a set of parties $\mathcal{P}$ is called linear (over $\mathbb{F}_q$) if:*

1. *the shares for each party form a vector over $\mathbb{F}_q$, and*
2. *there exists a matrix M called the* share-generator matrix *for Π with t rows and n columns such that the following property holds. For $i = 1, \dots, n$, the i-th column M_i of M is labeled by a party $\rho(i)$ where ρ is a function from $\{1, \dots, n\}$ to $\mathcal{P}$. Given the row vector $\mathbf{p} = (s, r_1, \dots, r_{t-1})$, where $s \in \mathbb{F}_q$ is the secret to be shared and $r_1, \dots, r_{t-1} \in \mathbb{F}_q$ are randomly chosen, $\mathbf{p} \cdot M$ gives the vector of n shares of the secret s according to Π. The share $\beta_i = (\mathbf{p}M)_i$, i.e., the inner product $\mathbf{p} \cdot M_i$, belongs to party $\rho(i)$.*

Moreover, n is the size *of Π.*

Note that the map ρ that associates each share and column of M to a party is non-necessarily injective. In this way, one participant is allowed to own more than one share. The share-generator matrix M coincides with the matrix denoted $G_{[1,\dots,n]}$ in observation 1, i.e. the matrix that encodes the secret into the shares, and it defines the access structure $\mathbb{A}$ of the scheme, specifying which subsets of participants can reconstruct the secret.

2.4 Adversary Models

In this subsection we collect some adversary models definition that are used later on. Each definition distinguishes between two opposite models.

Definition 4. *We distinguish between:*

- ***Static Adversary**: An adversary whose strategy is fixed and determined before the protocol begins.*
- ***Adaptive Adversary**: An adversary that can adapt their strategy based on the information gathered during the protocol's execution.*

Definition 5. *We distinguish between:*

- ***Mobile Adversary**: An adversary that can move between different players over time, as long as the total number of corrupt players is less than or equal to t. A mobile adversary is adaptive, but an adaptive adversary is not necessarily mobile.*
- ***Non-Mobile Adversary**: An adversary that does not change the set of controlled players during the execution of the protocol.*

Definition 6. *We distinguish between:*

- ***Snapshot Adversary**: An adversary that captures and analyzes the system's state at a single point in time, without the ability to monitor or interfere with the protocol after that point.*
- ***Continuous Shot Adversary**: An adversary that continuously observes the system over an extended period. The adversary controls the corrupt players continuously in time.*

2.5 Commitments

A commitment scheme is composed by two algorithms: $\mathrm{Com}(m, r)$ (given the message m to commit and some random value r outputs the commitment $\mathtt{KGC}$ and the decommitment $\mathtt{KGD}$) and $\mathrm{Ver}(\mathtt{KGC}, \mathtt{KGD})$ (given a commitment and its decommitment outputs the committed message m if the verification succeeds, $\perp$ otherwise). It must have the following two properties:

- **Binding:** given $\mathtt{KGC}$, it is infeasible to find values $m' \neq m$ and $\mathtt{KGD}$, $\mathtt{KGD}'$ such that $\mathrm{Ver}(\mathtt{KGC}, \mathtt{KGD}) = m$ and $\mathrm{Ver}(\mathtt{KGC}, \mathtt{KGD}') = m'$. We say that the commitment is *perfectly binding* if the binding property holds even if the adversary has unbounded computational power.

– **Hiding:** Let $[\mathtt{KGC}_1, \mathtt{KGD}_1] = \mathrm{Com}(m_1, r_1)$ and $[\mathtt{KGC}_2, \mathtt{KGD}_2] = \mathrm{Com}(m_2, r_2)$ with $m_1 \neq m_2$, then it is infeasible for an attacker having only $\mathtt{KGC}_1$, $\mathtt{KGC}_2$, m_1 and m_1 to identify which $\mathtt{KGC}_i$ corresponds to which m_i with more than negligible advantage. We say that the commitment is *perfectly hiding* if the hiding property holds even if the adversary has unbounded computational power.

Notice that perfect hiding and perfect binding are mutually exclusive properties, in fact in a perfectly binding commitment $\mathtt{KGC}$ can be decommitted in at most one way, so a computationally unbounded adversary can violate the hiding property via a brute-force search.

In this work we need a homomorphic commitment, that is a commitment HCom for which the following properties hold for all $m_0, m_1, z_0, z_1, \gamma \in \mathbb{F}_q$:

1. $\mathrm{HCom}(m_0; z_0) \cdot \mathrm{HCom}(m_1; z_1) = \mathrm{HCom}(m_0 + m_1; z_0 + z_1)$
2. $\mathrm{HCom}(m_0; z_0)^\gamma = \mathrm{HCom}(\gamma \cdot m_0; \gamma \cdot z_0)$

The Pedersen commitment [12], based on the difficulty of the discrete logarithm, is a perfectly hiding homomorphic commitment scheme which works as follows:

Setup let $\mathbb{G}$ be a group of prime order q where the DLOG problem is hard (for the binding property to hold), and g, h be random generators of $\mathbb{G}$, then the message space of the commitment scheme is $\mathbb{Z}_q$, the randomizer space is $\mathbb{Z}_q$ and the commitment space is $\mathbb{G}$;

Commitment to commit to $m \in \mathbb{Z}_q$ using the randomizer $z \in \mathbb{Z}_q$, the committer computes $C = \mathrm{HCom}(m, z) = g^m \cdot h^z$;

Verification the decommitment is the pair (m, z), and $\mathrm{Ver}(C, m, z)$ simply outputs m if $C = g^m \cdot h^z$, $\perp$ otherwise.

2.6 Extensible DKG

In this subsection, we mention a decentralized variant of the Verifiable Secret Sharing Scheme (VSSS) by Pedersen [12] presented in section 3 of paper [1]. It will be the starting point of our original contributions. It has three important features, illustrated in figure 1: it is *extensible* (possibility to add new parties), *decentralized* (no presence of a dealer) and *verifiable* (possibility to check received values). We refer to [1] for a complete description of the protocol and the secu-

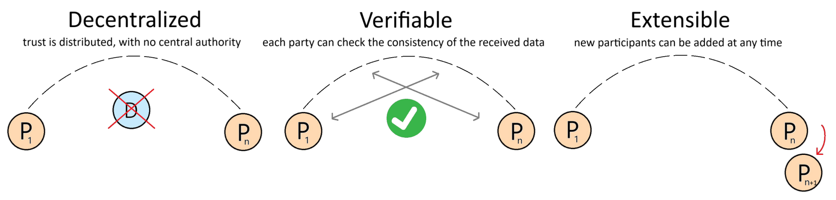
Decentralized
trust is distributed, with no central authority
Verifiable
each party can check the consistency of the received data
Extensible
new participants can be added at any time
D
P_1
P_n
P_1
P_n
P_1
P_n
P_{n+1}

Fig. 1. Properties of the protocol.

rity analysis of the three main phases: SecGen (secret and share generation), SecRec (secret reconstruction), AddNew (addition of new participants).

Complexity Analysis Here we present the results of a detailed complexity analysis that we performed. We take into consideration the following factors:

- *Number of Communications:* The total number of communications between participants, divided into: *private communications* (secret values between two participants), *public communications* (published values, i.e. commitments).
- *Data Storage:* The amount of data (values in $\mathbb{F}_q$ or in $\mathbb{G}$) that needs to be stored divided into: *Initial Data Storage* (the amount of data that needs to be stored for starting the phase), *Runtime Data Storage* (the amount of data that needs to be stored during the execution of the phase), *Final Data Storage* (the amount of data that needs to be stored at the end of the phase). All these three categories are furthermore divided into *Private data* (secret private values of a single participant, i.e. the amount of data that *each* participant must store) and Public data (public global parameters). Hence the amount of data that each participant needs to store is the sum of the two previous categories. It is always specified whether the values to be stored are in $\mathbb{F}_q$ or in $\mathbb{G}$, as elements of $\mathbb{G}$ will occupy more space than those in $\mathbb{F}_q$.
- *Number of Random Values to Generate:* The total number of random values that must be generated during the protocol.
- *Computational Cost:* The computational cost is measured in terms of the most expensive operation in the phase. In particular, the relevant operations require the following average time: exponentiation in $\mathbb{G} \to 0.135$ ms, multiplication in $\mathbb{F}_q \to 0.00334$ ms. Consistent with what can be expected from a theoretical standpoint, exponentiation in $\mathbb{G}$ is much more expensive than multiplication in $\mathbb{F}_q$. [1]

Table 1 summarizes the complexity analysis for SecGen, SecRec, AddNew. Table 2 shows the total data storage requested comprehensively for a single run (SecGen + AddNew + SecRec). Finally, we tested a proof-of-concept implementation to verify the theoretical results obtained. Moreover, we compare the time cost changing the elliptic curve on which the commitment is based: we found that Ed25519 is always better in terms of time, especially when n or τ increases; however, as expected, the choice of the curve does not influence the computational complexity in terms of its asymptotic behavior.

[1] Tests are executed using PC and softwares with the following characteristics:

Processor	Intel(R) Core(TM) i5-10210U CPU 1.60-2.11 GHz
RAM	8,00 GB (7,72 GB usable)
Operating system	Windows 11 Home, version 23H2, 64 bit
Python	3.10.12
Sagemath	9.5

		SecGen	SecRec	AddNew
Communications	**Private**	τn	-	$t^2 + t$
	Public	τt	-	t^2
Initial Data Storage	**Private**	-	1 in $\mathbb{F}_q$	2 in $\mathbb{F}_q$
	Public	2 in $\mathbb{G}$ 1 in $\mathbb{F}_q$	t^2 in $\mathbb{F}_q$	$(t^2 + t + 1)$ in $\mathbb{F}_q$ $(2 + t\tau)$ in $\mathbb{G}$
Runtime Data Storage	**Private**	$(2t + 2\tau)$ in $\mathbb{F}_q$	-	$(2t + 2)$ in $\mathbb{F}_q$
	Public	-	1 in $\mathbb{F}_q$	t^2 in $\mathbb{G}$
Final Data Storage	**Private**	2 in $\mathbb{F}_q$	-	2 in $\mathbb{F}_q$
	Public	$t\tau$ in $\mathbb{G}$	-	-
Random Generations		$2\tau t$	-	$2t(t - 1)$
Computational Cost		$\tau(nt + 2n + 2t)$ (exp in $\mathbb{G}$)	$\mathcal{O}(t^3)$ (mul in $\mathbb{F}_q$)	$(t^3 + 7t^2 + 4t)$ (exp in $\mathbb{G}$)

Table 1. Complexity Analysis of SecGen, SecRec, AddNew. $\mathbb{G}$ is the group used for the commitment, $\mathbb{F}_q$ is the base field for the secret sharing (as introduced in subsection 2.2), n is the total number of parties, t is the threshold and τ is the number of parties that execute SecGen.

Total Data Storage	**Private**	2 in $\mathbb{F}_q$
	Public	1 in $\mathbb{F}_q$ $(t\tau + 2)$ in $\mathbb{G}$

Table 2. Total data storage for a run.

3 DKG for general access policies

Up until now, the set of participants has been considered as a set of n equally powerful members, such that any t distinct participants can collaborate to recover the secret, while fewer than t participants cannot obtain any information about the secret. In this section, we will demonstrate how it is possible to extend a Shamir threshold secret sharing scheme to more general access policies. For example, with this technique it is possible to have some groups of participants more powerful than others or force specific collaborations.

To achieve this, we will first analyze the theory of access structures following [9], to discover that the most general and efficient structure, encompassing all others, is the *threshold access tree*. For this reason, in the following subsections, we will aim to extend the sharing scheme to any threshold access tree of participants. The results obtained can also be applied in the context of attribute-based encryption [7], where the role of parties is defined by attributes.

Besides the theoretical way of describing access policies using access structures, there are several more practical and efficient ways for describing policies. Let us see these structures.

Minimal Form Access Structures Due to the monotonicity, a monotone access structure $\mathbb{A}$ can be efficiently described by a set $\mathbb{A}^-$, which consists of the minimal elements (sets) in $\mathbb{A}$, i.e., the elements in $\mathbb{A}$ for which no proper subset is also in $\mathbb{A}$. For any party set $S \subseteq \mathcal{P}$, if and only if there exists some set $A \in \mathbb{A}^-$ such that $S \supseteq A$, S is an *authorized set* (i.e., $S \in \mathbb{A}$, or S satisfies $\mathbb{A}$). We refer to $\mathbb{A}^-$ as the *minimal form* of $\mathbb{A}$. Obviously, it is more efficient to use the minimal form to describe an access policy rather than the access structure itself which may contain many redundancies.

Monotone Boolean Formulas A *monotone boolean formula* is a boolean formula without NOT. For any minimal form monotone access structure, say $\mathbb{A}^- = \{A_1, \ldots, A_{n_\mathbb{A}}\}$, an equivalent monotone boolean formula, say $F_\mathbb{A} = \bigvee_{1 \le i \le n_\mathbb{A}} \left(\bigwedge_{x \in A_i} x \right)$, can be compressed and simplified further to obtain a version with smaller size, e.g. $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$. On the other hand, given a monotone boolean, to transform it to an equivalent minimal form access structure, we need to first convert the boolean formula to an equivalent Disjunctive Normal Form (DNF: dis-junction (OR) of conjunctions (AND) of literals, where each literal consists of variables or their negations), whose size may be larger than the original boolean formula.

Monotone Access Trees On a monotone access tree, each leaf node corresponds to a party, and each non-leaf node represents an internal gate, which is described by its children and its label. If internal nodes are threshold-gate nodes, the tree is generally termed as a *Threshold-gate access tree*. Whether a party set S satisfies a monotone Threshold-gate access tree is determined as follows. For a leaf node, if the corresponding party appears in S, the leaf node is said to be *satisfied*. For a (t, n)-threshold gate (where n is the number of its children and $1 \le t \le n$ is the threshold value), iff at least t out of n child nodes are satisfied, the (t, n)-threshold node is satisfied. Finally, if and only if the root node of the access tree is satisfied, the access tree is said to be *satisfied by S*.

An *AND-OR-gate access tree* is a special case of the Threshold-gate access tree in which the label of each internal node is either an AND (equivalent to $t = n$) or OR (equivalent to $t = 1$) gate. As a result, given an AND-OR-gate access tree, we can obtain a Threshold-gate access tree with the same number of leaf nodes. Conversely, to express a general (t, n)-threshold gate, multiple AND and OR gates have to be used and therefore, an equivalent AND-OR-gate access tree will have more leaf nodes than the original Threshold-gate access tree. Also note that an AND-OR-gate access tree is the same as an equivalent monotone boolean formula in terms of generality and efficiency, where AND gate corresponds to $\wedge$ and OR gate to $\vee$.

The expressivity of these structures are equivalent. If we use $A \succ B$ (resp. $A \sim B$) to represent that form A is more efficient than form B (resp. form A is equally efficient to form B), we have the following relationships: monotone access structure $\prec$ minimal form access structure $\prec$ monotone boolean formula $\sim$ AND-OR-gate access tree $\prec$ Threshold-gate access tree. An example is shown in figure 2, where several options are listed from top to bottom in order of efficiency. Therefore, Threshold-gate access tree is the most general and efficient

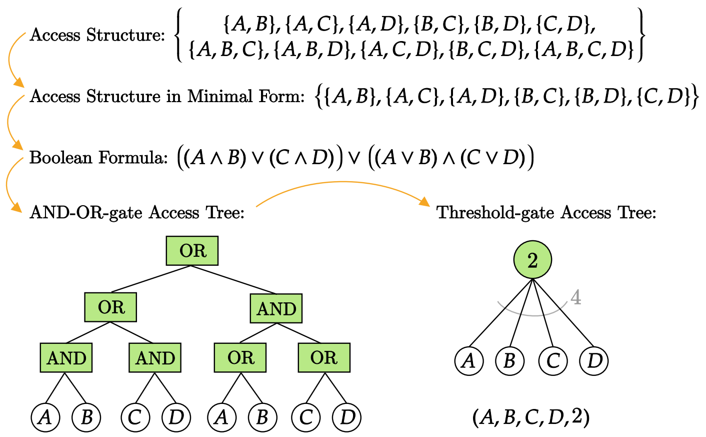
Access Structure: {A,B},{A,C},{A,D},{B,C},{B,D},{C,D}, {A,B,C},{A,B,D},{A,C,D},{B,C,D},{A,B,C,D}
Access Structure in Minimal Form: {{A,B},{A,C},{A,D},{B,C},{B,D},{C,D}}
Boolean Formula: ((A ∧ B) ∨ (C ∧ D)) ∨ ((A ∨ B) ∧ (C ∨ D))
AND-OR-gate Access Tree:
Threshold-gate Access Tree:
OR
AND
A
B
C
D
2
4
(A,B,C,D,2)

Fig. 2. Different forms for describing a $(2,4)$-threshold access policy over the universe $\{A,B,C,D\}$.

form.

Recall that any monotone access structure can be realized by a linear secret sharing scheme, regardless of the method used to express it, as shown in [2]. Starting from a general Threshold-gate access tree, we aim to realize a LSSS matrix that describes its access policy in the context of our Shamir secret sharing scheme. In this section, we explore this path starting from our Secret Sharing protocol defined on a simple threshold policy, and we extend it to a general Threshold-gate access tree. Remember the link between the secret s and the shares β_i in a (t,n) Secret Sharing protocol:

1. The secret s is equal to the constant term of the polynomial $p(x)$ of degree $t-1$, i.e. $s = p_0$.
2. The other $(t-1)$ coefficients of $p(x)$ are random values: $(p_1,\dots,p_{t-1}) = (r_1,\dots,r_{t-1})$.
3. n shares $(\beta_1,\dots,\beta_n)$ are obtained from the vector $(p_0,p_1,\dots,p_{t-1})$ through the multiplication with a generator matrix G that encodes the secret: $(p_0,p_1,\dots,p_{t-1}) \cdot G = (\beta_1,\dots,\beta_n)$.
4. The matrix G integrates the access policy through the independence relationships among its columns. Each column is associated to a single participant. In a (t,n)-Shamir Secret sharing protocol, G is a generator matrix for an $[n,t,n-t+1]$ MDS code (any t columns of G are linearly independent) with the following structure:

$$G = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \alpha & \alpha^2 & \dots & \alpha^n \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{t-1} & \alpha^{2(t-1)} & \dots & \alpha^{n(t-1)} \end{pmatrix}.$$

5. Whenever t parties (with $J = \{j_1, \ldots, j_t\}$ set of party-indexes) collaborate to recover the secret, a new square-matrix of size $(t \times t)$ named $\overline{G}$ (in previous chapters it was named G_J) is obtained from G, selecting all the columns of G with index in J.
6. $\overline{G}$ is invertible and allows to recover the secret: $(p_0, \ldots, p_{t-1}) = (\beta_{j_1}, \ldots, \beta_{j_t}) \cdot \overline{G}^{-1}$.

Suppose that we have a general Threshold-gate access tree. To recover the secret, an authorized set of participants (leaves) must satisfy all the Threshold-gates from the bottom up to the root node. We can imagine the tree as a layered repetition of individual cascading secret sharing thresholds. The root node encodes the secret vector in a set of shares. These shares become secrets for the second layer, that are encodes in sets of shares for the next layer and so on, until the final shares of the last layer. Figure 3 explains this idea. Ideally, shares

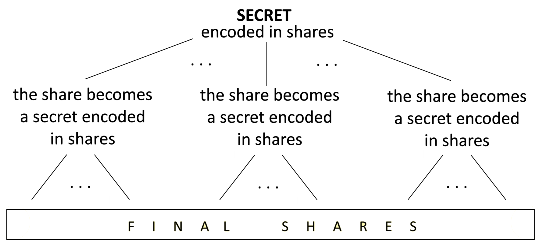
SECRET
encoded in shares
the share becomes a secret encoded in shares
the share becomes a secret encoded in shares
the share becomes a secret encoded in shares
FINAL SHARES

Fig. 3. Key idea for extending Secret Sharing to Threshold-gate access trees.

are computed top-down from the secret, while the secret is recovered bottom-up from an authorized set of shares. More formally, each (t_i, n_i) Threshold-node is represented by a MDS matrix with parameters $[n_i, t_i, n_i - t_i + 1]$. Remembering these observations, now we focus on a simple example of Threshold-gate access tree, showed in figure 4. First of all, let us see how the secret reconstruction

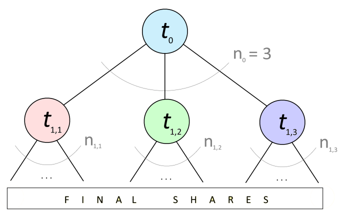

Fig. 4. Example of Threshold-gate access tree. Colors will be used in the explanation for better clarity.

works in this precise tree:

1. $t_{1,1}$ valid distinct shares $\{\beta_j^{(1,1)}\}_{j \in J_{1,1}}$ among the $n_{1,1}$ of the pink node are selected and the corresponding columns of $G_{1,1}$ are used to build $\overline{G}_{1,1}$. In

this way, the coefficients of the polynomial related to this node are recovered: $\left(\beta_j^{(1,1)}\right)_{j\in J_{1,1}} \cdot \overline{G}_{1,1}^{-1} = (p_0^{(1,1)}, \dots, p_{t_{1,1}-1}^{(1,1)})$.

2. Similarly for the green and blue nodes: $\left(\beta_j^{(1,2)}\right)_{j\in J_{1,2}} \cdot \overline{G}_{1,2}^{-1} = (p_0^{(1,2)}, \dots, p_{t_{1,2}-1}^{(1,2)})$, $\left(\beta_j^{(1,3)}\right)_{j\in J_{1,3}} \cdot \overline{G}_{1,3}^{-1} = (p_0^{(1,3)}, \dots, p_{t_{1,3}-1}^{(1,3)})$.
3. From the floor level (the one with final shares), we obtained the coefficients of the polynomials of all the nodes in the middle layer. Among these coefficients, we select only the constant terms: these values are "secrets" for the middle level, but also shares for the top level. Now, t_0 constant terms $\left\{p_0^{(1,j)}\right\}_{j\in J_0} = \left\{\left(\beta_i^{(1,j)}\right)_{i\in J_{1,j}} \cdot \overline{G}_{1,j}^{-1} \cdot e_1^T\right\}_{j\in J_0}$ among the n_0 of the middle nodes are selected and the corresponding columns of G_0 are used to build $\overline{G}_0$. In this way, the coefficients of the polynomial related to the root node are recovered: $\left(p_0^{(1,j)}\right)_{j\in J_0} \cdot \overline{G}_0^{-1} = (p_0^{(0)}, \dots, p_{t_0-1}^{(0)})$. In particular, the first component is the secret: $s = \left(p_0^{(1,j)}\right)_{j\in J_0} \cdot \overline{G}_0^{-1} \cdot e_1^T$.

Our goal is to build a single matrix that encodes the secret directly in the final shares. In particular, the values that must be encoded are the secret together with all the random elements (in a single Threshold-gate the random elements are just $(t-1)$). Let us see what are the encoded elements considering a Threshold-gate access tree:

– The secret s together with $(t_0 - 1)$ random values form the vector $(p_0^{(0)}, \dots, p_{t_0-1}^{(0)})$ of coefficients of the root polynomial. This vector is encoded by G_0 in a vector of shares. This vector of shares contains all constant terms of the polynomials of lower level nodes.
– For each of these nodes, other $(t_{(1,j)} - 1)$ random values are generated and the vector containing the constant term plus these random values describes the polynomial of the node $(p_0^{(1,j)}, \dots, p_{t_{1,j}-1}^{(1,j)})$. Finally, each of these vectors is encoded by $G_{1,j}$ in final shares.
– Therefore, the random (except for the secret) values from which we start and that we need to encode with the single matrix we are looking for, are all the coefficients of the root polynomial plus the coefficients of the other polynomials except for their constant terms:

$$\begin{array}{llll} p_0^{(0)}, & p_1^{(0)}, & \dots, & p_{t_0-1}^{(0)} \\ & p_1^{(1,1)}, & \dots, & p_{t_{1,1}-1}^{(1,1)} \\ & p_1^{(1,2)}, & \dots, & p_{t_{1,2}-1}^{(1,2)} \\ & p_1^{(1,3)}, & \dots, & p_{t_{1,3}-1}^{(1,3)}. \end{array}$$

With these observations, we can start building our matrix. Focusing on the middle level, we can observe that the encoding process can be executed for all the three nodes together with the following matrix:

$$\left(p_0^{(1,1)},\dots,p_{t_{1,1}-1}^{(1,1)},\ p_0^{(1,2)},\dots,p_{t_{1,2}-1}^{(1,2)},\ p_0^{(1,3)},\dots,p_{t_{1,3}-1}^{(1,3)}\right)\cdot\begin{pmatrix} G_{1,1} & & \\ & G_{1,2} & \\ & & G_{1,3}\end{pmatrix} =$$

$$=\left(\beta_1^{(1,1)},\dots,\beta_{n_{1,1}}^{(1,1)},\beta_1^{(1,2)},\dots,\beta_{n_{1,2}}^{(1,2)},\beta_1^{(1,3)},\dots,\beta_{n_{1,3}}^{(1,3)}\right).$$

Then we have to integrate the matrix G_0. Since each column of it corresponds to a lower node, we highlight them with the same color of the nodes:

$$G_0=\begin{pmatrix} g_{1,1} & g_{1,2} & g_{1,3}\\ g_{2,1} & g_{2,2} & g_{2,3}\\ g_{3,1} & g_{3,2} & g_{3,3}\end{pmatrix}_{t_0\times n_0}.$$

These columns encode the coefficients of the root polynomial into the constant terms of the lower polynomials, therefore we build the following matrix (named A) for the encoding process of the root level:

$$\left(p_0^{(0)},p_1^{(0)}\dots,p_{t_0-1}^{(0)},\ p_1^{(1,1)},\dots,p_{t_{1,1}-1}^{(1,1)},\ p_1^{(1,2)},\dots,p_{t_{1,2}-1}^{(1,2)},\ p_1^{(1,3)},\dots,p_{t_{1,3}-1}^{(1,3)}\right)\cdot$$

$$\cdot\left(\begin{array}{ccc|ccc|ccc} g_{1,1} & & & g_{1,2} & & & g_{1,3} & & \\ g_{2,1} & & & g_{2,2} & & & g_{2,3} & & \\ g_{3,1} & & & g_{3,2} & & & g_{3,3} & & \\ \hline & 1 & & & & & & & \\ & & \ddots & & & & & & \\ & & 1 & & & & & & \\ \hline & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & 1 & & & \\ \hline & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & 1 \end{array}\right)_{(t_0+\sum_{i=1}^3(t_{1,i}-1))\times(\sum_{i=1}^3 t_{1,i})} = \qquad (1)$$

$$=\left(p_0^{(1,1)},p_1^{(1,1)},\dots,p_{t_{1,1}-1}^{(1,1)},\ p_0^{(1,2)},p_1^{(1,2)},\dots,p_{t_{1,2}-1}^{(1,2)},\ p_0^{(1,3)},p_1^{(1,3)},\dots,p_{t_{1,3}-1}^{(1,3)}\right).$$

Remark 2. The matrix A in the previous equation can be obtained as a column permutation of a simpler matrix:

$$A=\left(\begin{array}{c|c} G_0 & \\ \hline & I_{\sum_{i=1}^3(t_{1,i}-1)}\end{array}\right)\cdot\left(\begin{array}{c|c|c|c|c|c} 1 & & & & & \\ \hline & & 1 & & & \\ \hline & & & & 1 & \\ \hline & I_{t_{1,1}-1} & & & & \\ \hline & & & I_{t_{1,2}-1} & & \\ \hline & & & & & I_{t_{1,3}-1}\end{array}\right).$$

Finally, we have to put together the two layers obtaining a unique encoding process that encodes the starting values into the final shares.

$$\left(\boxed{p_0^{(0)}, p_1^{(0)} \dots, p_{t_0-1}^{(0)}}\,, \boxed{p_1^{(1,1)}, \dots, p_{t_{1,1}-1}^{(1,1)}}\,, \boxed{p_1^{(1,2)}, \dots, p_{t_{1,2}-1}^{(1,2)}}\,, \boxed{p_1^{(1,3)}, \dots, p_{t_{1,3}-1}^{(1,3)}}\right) \cdot$$

$$\cdot A \cdot \begin{pmatrix} \boxed{G_{1,1}} & & \\ & \boxed{G_{1,2}} & \\ & & \boxed{G_{1,3}} \end{pmatrix} = \tag{2}$$

$$= \left(\beta_1^{(1,1)}, \dots, \beta_{n_{1,1}}^{(1,1)}, \beta_1^{(1,2)}, \dots, \beta_{n_{1,2}}^{(1,2)}, \beta_1^{(1,3)}, \dots, \beta_{n_{1,3}}^{(1,3)}\right).$$

The matrix obtained from the product of the two matrices in the previous equation is the generator matrix we were looking for. If the Threshold-gate access tree has more levels, it is sufficient to iterate the procedure up to the root node. Consider for example the tree in figure 5. Firstly, we analyze orange and violet

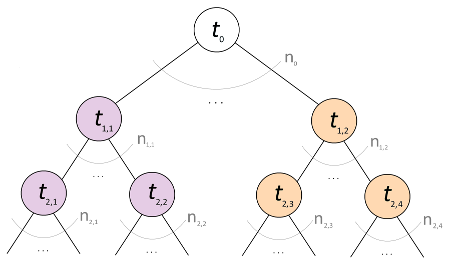

Fig. 5. Example of a 3 levels Threshold-gate access tree. Colors will be used in the explanation for better clarity.

sub-trees independently, following the procedure done until now and obtaining the two matrices:

$$\boxed{A_{1,1} \cdot \begin{pmatrix} G_{2,1} & \\ & G_{2,2} \end{pmatrix}}\,, \quad \boxed{A_{1,2} \cdot \begin{pmatrix} G_{2,3} & \\ & G_{2,4} \end{pmatrix}}\,,$$

where $G_{1,1}$ is encoded in $A_{1,1}$ and $G_{1,2}$ in $A_{1,2}$, as showed in equation 1. Then we integrate the root node with the same strategy, obtaining the final matrix:

$$A_0 \cdot \begin{pmatrix} \boxed{A_{1,1} \cdot \begin{pmatrix} G_{2,1} & \\ & G_{2,2} \end{pmatrix}} & \\ & \boxed{A_{1,2} \cdot \begin{pmatrix} G_{2,3} & \\ & G_{2,4} \end{pmatrix}} \end{pmatrix}, \tag{3}$$

where G_0 is encoded in A_0 as showed in equation 1.

Remark 3. Observe that the starting vector is obtained collecting the starting polynomial coefficients of each node, following the same node-order of a DEPTH-FIRST-SEARCH of the tree-graph.

3.1 Computing the Final Matrix

We finally found the structure of the generator matrix we were looking for, but until now we are able to write it only as a multiplication of matrices. In this section we try to find its final structure. Since the procedure to obtain the matrix is iterative, it is sufficient to describe how the matrix change whenever a new node is inserted. Then, starting from the simple generator matrix that describes only the threshold root node, it is possible to obtain the final structure of the matrix related to the whole tree following the instructions step by step (adding one node at a time).

Suppose that we start from a generator matrix M_1 related to a Threshold-gate access tree with root label t_0 and we want to describe the new matrix M that describes the tree after the insertion of a new threshold-node labeled t_1 described by the generator matrix M_2. The situation is described in figure 6. In

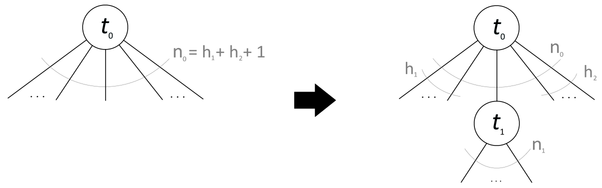

Fig. 6. Insertion of a threshold-node.

M_1 we denote with $\boldsymbol{v}$ the column vector that corresponds to the party instead of which the new node is inserted, with $M_{1,1}$ the sub-matrix containing the set of h_1 columns that precedes v and with $M_{1,2}$ the sub-matrix containing the set of h_2 columns that succeeds $\boldsymbol{v}$: $M_1 = \left(M_{1,1} \middle| \boldsymbol{v} \middle| M_{1,2} \right)_{t_0 \times n_0}$. In M_2 we denote with $\boldsymbol{u}$ the first row and with $M_{2,1}$ the sub-matrix containing the other rows of M_2: $M_2 = \left(\begin{array}{c} \boldsymbol{u} \\ \hline M_{2,1} \end{array} \right)_{t_1 \times n_1}$. Following the strategy explained in equations 1, 2, the matrix that describes the final tree is the product of the two following matrices:

$$M = \left(\begin{array}{c|c|c|c} M_{1,1} & \boldsymbol{v} & & M_{1,2} \\ \hline & & I_{t_1 - 1} & \end{array} \right) \cdot \left(\begin{array}{c|c|c} I_{h_1} & & \\ \hline & M_2 & \\ \hline & & I_{h_2} \end{array} \right).$$

The sizes of these two matrices are $(t_0 + t_1 - 1) \times (h_1 + t_1 + h_2)$ and $(h_1 + t_1 + h_2) \times (h_1 + n_1 + h_2)$, respectively. By carrying out the multiplication and paying attention to the dimensions of sub-matrices, we obtain the final matrix:

$$M = \left(\begin{array}{c|c|c} M_{1,1} & \boldsymbol{v} \otimes \boldsymbol{u} & M_{1,2} \\ \hline & M_{2,1} & \end{array} \right)_{(t_0 + t_1 - 1) \times (h_1 + n_1 + h_2)}, \tag{4}$$

where $\boldsymbol{v} \otimes \boldsymbol{u}$ is the matrix obtained from the tensor product: $(\boldsymbol{v} \otimes \boldsymbol{u})_{i,j} := v_i \cdot u_j$. We finally found a way to describe the LSSS matrix related to a general Threshold-gate access tree and we did it extending the strategy of Shamir Secret sharing layer-by-layer. The result found in this subsection matches the algorithm proposed in paper [9], in which the authors exploit the theory of *Monotone Span Programs (MSP)* to generate a LSSS matrix related to a general Threshold-gate access tree. In section 3.2 we describe carefully the algorithm proposed in [9], adapting it to our more convenient notation and mentioning also its relevant computational efficiency.

3.2 Efficient Algorithm

In this subsection we present the algorithm proposed in [9] that coincides with the result we obtained in section 3.1, adapted with our notation. It is based on the following strategy:

1. Write the LSSS matrix for a single (t, n)-threshold gate.
2. Repeatedly perform the insertion operation according to the structure of the input Threshold-gate access tree.
3. Use a formatted string to describe the input tree and design an algorithm which can output (M, ρ) based on the formatted string alone.

Firstly, let us describe how to associate a string to a tree.

1. Leaves (participants or attributes) are described with letters: $A, B, C, \ldots$ or $P_1, P_2, \ldots$
2. "()" will be used to define non-leaf nodes and "," will work as a separator to separate the child nodes of each non-leaf node and the threshold value. For example, a Threshold-gate with threshold 2 and child leaves A, B, C, D corresponds to the string $(A, B, C, D, 2)$.
3. A Threshold-gate access tree $\mathbb{A}$ is described by a string that describes its root node. In particular, suppose that the root node is a (t, n)-threshold gate. The access tree can then be described by a string $(F_1, F_2, \ldots, F_n, t)$ where F_i $(1 \leq i \leq n)$ are strings that represent the children of the root node.
4. F_i can be a letter corresponding to a leaf node, or a non-leaf node described by its children and a threshold value, i.e. $F_i = (F_{i,1}, F_{i,2}, \ldots, F_{i,n_i}, t_i)$, where $F_{i,1}, F_{i,2}, \ldots, F_{i,n_i}$ represent the n_i children of F_i and t_i is a threshold value with $1 \leq t_i \leq n_i$.

We refer to such a recursive-form string as a *threshold-tree-string*. Furthermore, given a threshold-tree-string F_A, when we say "i^{th} attribute of F_A" we mean the i^{th} attribute, indexed from left to right, ignoring the symbols "(", ")" and ",".

Remember that, for a (t, n)-threshold access structure described by the threshold-tree-string $(P_1, \ldots, P_n, t)$, we can construct the corresponding LSSS as in definition 1:

$$M = \left[\alpha^{j \cdot k}\right]_{k \in \{0,\ldots,t-1\}, j \in \{1,\ldots,n\}}; \qquad \rho(i) = P_i, \quad \forall i \in \{1, \ldots, n\}. \tag{5}$$

Note that M is completely determined by the values n and t, and (M, ρ) is completely determined by the string $(P_1, P_2, \ldots, P_n, t)$, as $\rho(i) = P_i$. Using an

LSSS as in Equation 5 for each Threshold-gate, we can follow the threshold-tree-string's structure to repeatedly execute the one-column-insertion, and eventually output an LSSS (M, ρ) for $\mathbb{A}$, where ρ is determined by $F_{\mathbb{A}}$ as the i^{th} row of M is labeled by the i^{th} attribute of $F_{\mathbb{A}}$. In particular, each node of the access tree is regarded as a participant of a threshold access structure specified by its parent node. For this reason, we start with a $(1, 1)$-threshold LSSS and consider the root node of the tree as its participant. Let us see precisely the algorithm.

Input A threshold-tree-string $F_{\mathbb{A}}$ for a Threshold-gate access tree $\mathbb{A}$.
Output A matrix M and a function ρ, which maps the i^{th} row of M to the i^{th} attribute in $F_{\mathbb{A}}$, (M, ρ) is the LSSS realizing $\mathbb{A}$.
Algorithm description Steps:

1. Let $M \coloneqq (1)_{1\times 1}$ matrix and $L \coloneqq (F_{\mathbb{A}})$ vector of strings.
2. Repeat the following steps until all coordinates of L are single letters (attributes or participants):
   (a) Scan the coordinates of $L = (L_1, \ldots, L_m)$ to find the first coordinate that is a threshold-tree-string rather than a single letter. Suppose the index of this coordinate is i and $L_i = (F_{i,1}, \ldots, F_{i,n_i}, t_i)$.
   (b) Let M_2 be the LSSS matrix that describes the Threshold-node associated to this coordinate, computed as in equation 5.
   (c) Execute one-column insertion (equation 4) of M_2 on the i^{th} column of M to obtain a new M.
   (d) Set $L = (L_1, \ldots, L_{i-1}, F_{i,1}, \ldots, F_{i,n_i}, L_{i+1}, \ldots, L_m)$
3. Return the matrix M.

Remark 4. If we denote with n the number of leaves (participants or attributes) of $\mathbb{A}$ and with $t_0, t_1, t_2, \ldots, t_w$ the thresholds of the root node and all the internal nodes, respectively, then the size of the output matrix M is: $(t_0+\sum_{i=1}^{w}(t_i-1))\times n$.

The time complexity of this algorithm is $O(n^2)$, in particular $10n^2$. We refer to [9] for the complexity analysis.

3.3 Extension of the Phases

In this subsection we show how the phases of our protocol can be adapted to the case of a Threshold Access Tree. We carefully describe the new versions of SecGen and SecRec, while for AddNew and Refresh we just give the guideline to adapt them, since the strategy is very similar and trivial.

SecGen The matrix M that describes the Threshold-access tree is supposed to be known and public. The algorithm described in 3.2 allows to compute it starting from the Threshold-gate access tree. Recall that the size of M is $(t_0 + \sum_{i=1}^{w}(t_i - 1)) \times n$, where n is the number of leaves, t_0 is the threshold of the root node, w is the number of internal nodes, t_i is the threshold of an internal node.
To adapt SecGen we have to substitute:

- $t_0 + \sum_{i=1}^{w}(t_i - 1)$ instead of t.
- A secret vector $p^{(i)} = (p_0^{(i)}, \dots, p^{(i)}_{(t_0+\sum_{i=1}^{w}(t_i-1))-1})$ instead of the secret polynomial $p^{(i)}(x)$, a random vector $z^{(i)} = (z_0^{(i)}, \dots, z^{(i)}_{(t_0+\sum_{i=1}^{w}(t_i-1))-1})$ instead of the random polynomial $z^{(i)}(x)$.
- The secret is the sum of the first components $p_0 = \sum_{i=1}^{\tau} p_0^{(i)}$.
- The values $\beta_{i,j} \coloneqq p^{(i)} \cdot M_j$,$\gamma_{i,j} \coloneqq z^{(i)} \cdot M_j$, (where M_j is the column of M associated to P_j), instead of the evaluations $\beta_{i,j} \coloneqq p^{(i)}(\alpha^j)$, $\gamma_{i,j} \coloneqq z^{(i)}(\alpha^j)$.
- $(M)_{k,j}$ instead of $\alpha^{i \cdot k}$, in checks.

SecRec If J is a list of distinct indexes corresponding to an authorized set of participants, then with the vector of shares $(\beta_j)_{j\in J}$ it is possible to reconstruct the secret p_0 as follows: $p_0 = \sum_{j\in J} \omega_j \beta_j$, where ${\omega_j}_{j\in J}$ are the solutions of the linear system $\sum_{j\in J} \omega_j M_j = (1, 0, \dots, 0)^T$, where M_j is the column of M corresponding to P_j. The system has $(t_0 + \sum_{i=1}^{w}(t_i - 1))$ equations and $|J|$ $(\leq (t_0 + \sum_{i=1}^{w}(t_i - 1)))$ unknowns.

AddNew An authorized set J of participants with a more restricted request (see Definition 7 below) must collaborate to generate a new share and add a new column to the matrix M. In particular, suppose that we start from a Threshold-gate access tree described by the matrix M_1 and we want to add h new participants obtaining a Threshold-gate access tree described by the matrix M_2.

Remark 5. M_2 must be an "extension" of M_1, i.e. M_1 with new columns. This is needed because the column of M_1 must remain immutable in M_2, in this way the shares of all the participants before the addition remain valid.

In order to add h new participants, we execute an adaptation of AddNew h times: at each time a column is added to M_1 and a new share is generated for the new leaf. To add a new share, we have to recover the entire secret vector $\boldsymbol{p}$ that has the secret p_0 as first component (and other random elements). Knowing this vector and the new column $\boldsymbol{v}$ that has to be added, the new share will be equal to $\boldsymbol{p} \cdot \boldsymbol{v}$. The length of $\boldsymbol{p}$ is $(t_0 + \sum_{i=1}^{w}(t_i - 1))$, therefore, we need $(t_0 + \sum_{i=1}^{w}(t_i - 1))$ number of shares, i.e. this is the number of participants that have to collaborate to add a new share. In particular, it is needed that EVERY threshold-node is *fulfilled*, according to the following notion:

Definition 7. *The root node is fulfilled if its threshold t_0 is achieved. An internal node is fulfilled if its threshold minus one $(t_i - 1)$ is achieved.*

Collecting together a set of shares that fulfills all nodes, it is possible to recover $\boldsymbol{p}$ and compute a new share. The sub-matrix M_J with the columns of M corresponding to the parties in J is a square invertible matrix of size $(t_0 + \sum_{i=1}^{w}(t_i - 1)) \times (t_0 + \sum_{i=1}^{w}(t_i - 1))$ that plays the role of G_J in the original protocol. To adapt AddNew we have to substitute:

- M_J instead of G_J.
- The column of M_2 that we are adding (named $\boldsymbol{v}$) instead of G_{n+1}.
- $(M_1)_{k,j_l}$ instead of $\alpha^{j_l \cdot k}$.
- The k^{th} element of $\boldsymbol{v}$ instead of $\alpha^{(n+1)\cdot k}$.

3.4 Complexity

Finally, we report the complexity of the phases adapted to a Threshold-gate access tree.

- SecGen: same of section 2.6, with $(t_0 + \sum_{i=1}^{w}(t_i - 1))$ instead of t.
- SecRec: The computational cost is that of solving the linear system of size $(t_0 + \sum_{i=1}^{w}(t_i - 1)) \times |J|$, hence it depends (generally $\mathcal{O}((t_0 + \sum_{i=1}^{w}(t_i - 1))^2\,|J|))$.
- AddNew: h times the cost of adding one share that is the same of section 2.6 with $(t_0 + \sum_{i=1}^{w}(t_i - 1))$ instead of t.
- Refresh: same of section 4.3 with $(t_0 + \sum_{i=1}^{w}(t_i - 1))$ instead of t.

4 Share Refreshing

If the information stored by participants to share a secret remains unchanged throughout the system's lifetime, an adversary could eventually breach enough participants to recover the secret. To counteract this risk, proactive security introduces the concept of dividing time into periods known as *epochs*. At the start of each epoch, the shares held by participants are updated, although the shared secret itself remains constant. This approach enhances protection for long-lived secrets, especially against a *mobile adversary*, which is an adversary that can move among players over time but can only control a limited subset of players at any given moment (as explained in [10]). By periodically refreshing the shares, proactive security ensures that the adversary does not have enough time to break into the necessary number of participants. Moreover, the information they get at a certain time, becomes obsolete. This means that what they learned during one epoch is useless in subsequent epochs, forcing the adversary to start its attack afresh with each new period. In practice, the system's lifetime is divided into epochs according to a global clock. As shown in Figure 7, at the end of each epoch, participants engage in an interactive update protocol named *Refresh*, which ensures the secret's value is not revealed nor changed. Consequently, at the beginning of each new epoch, participants hold updated shares of the secret, reinforcing the security of the system.

Proactive security does not make sense if we consider a Non-Mobile (definition 5) or Static (definition 4) adversary. In these cases, the adversary does not change strategy or corrupted participants over time; the set of corrupted players remains the same from the start. This set has a cardinality less than t. A refresh phase will not add security since the adversary will not discover new shares, whether old or new. On the other hand, if we consider a Continuous Shot Adversary (definition 6), proactive security will be useless since the adversary

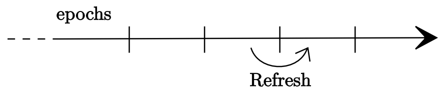

Fig. 7. Refresh general idea: periodically and proactively restore the security of the system.

has full control of the corrupted players. This means they will possess all the new shares of the corrupted players, and therefore, no additional security is provided. In conclusion, for our purposes, we consider a Snapshot, Mobile and Adaptive adversary. More precisely, suppose that the adversary has full control of x participants while possesses y lost or stolen shares. It holds $x+y \leq t-1$, otherwise the adversary is able to recover the secret. Proactive security aims to reset the threshold to $t-x$, rending useless the y lost or stolen shares. Moreover, in case we consider Snapshot Adversary, the threshold is reset to t. Finally, we can consider also active adversaries thank to the use of commitment. In particular, we use Pedersen commitment that is perfectly hiding but only computationally binding. Therefore we consider computationally bounded adversaries, even if the hiding property is secure even if the adversary has unbounded computational power.

Remark 6. If instead of Pedersen commitment, a perfectly binding but only computationally hiding is used, the adversary type is still Computationally Bounded, even if the biding property is secure even if the adversary has unbounded computational power. Computationally Unbounded Adversaries can not be taken into consideration since the commitment can not be both perfectly hiding and perfectly binding (as explained in subsection 2.5).

We propose three distinct methods for the refresh phase, let us see them in details.

4.1 First Decentralized Refresh (ZeroSharesRefresh)

$\tau \leq n$ participants $P_1, \dots, P_\tau$ collaborate to refresh shares and checking values for the players $P_1, \dots, P_n$. The secret remains p_0, the threshold t.

Player P_i, $i \in \{1, \dots, \tau\}$	
Private Input:	(β_i, γ_i)
Public Input:	α, g, h
Private Output:	$(\overline{\beta}_{n+1}, \overline{\gamma}_{n+1})$
Public Output:	-

1. Each P_i for $i \in \{1, \dots, \tau\}$ generates a secret polynomial $h^{(i)} \in \mathbb{F}_q[x]$ of degree $t-1$, by sampling the coefficients $h_k^{(i)}$ uniformly at random in $\mathbb{F}_q$, with $h_0 = 0$.
2. Each P_i samples another random polynomial $e^{(i)} \in \mathbb{F}_q[x]$ of degree $t-1$ with $e_0 = 0$, and uses its coefficients to compute and publish the commitments to the coefficients of their secret polynomial $h^{(i)}$: $C_{i,k} = \text{HCom}\left(h_k^{(i)}; e_k^{(i)}\right)$ for$k \in \{1, \dots, t-1\}$.

3. After having received every single commitment $C_{j,k}$, for $j \in \{1, \dots, \tau\}$ and $k \in \{0, \dots, t-1\}$, each P_i sends to each P_j the evaluations $\delta_{i,j} = h^{(i)}(\alpha^j)$ and $\epsilon_{i,j} = e^{(i)}(\alpha^j)$.
4. Each P_i for $i \in \{1, \dots, \tau\}$ sends the pair $(\delta_{i,j}, \epsilon_{i,j})$ also to every party P_j for $j \in \{\tau+1, \dots, n\}$.
5. By exploiting the homomorphic properties of the commitment scheme, each P_i for $i \in \{1, \dots, n\}$ checks the values received against the published commitments:
$$\mathrm{HCom}(\delta_{j,i}; \epsilon_{j,i}) \overset{?}{=} \mathrm{HCom}(0, 0) \cdot \prod_{k=1}^{t-1} (C_{j,k})^{(\alpha^i)^k}, \qquad \forall j \in \{1, \dots, \tau\}. \tag{6}$$
6. If all of these checks pass, each P_i sets its new share of the same secret as $\overline{\beta_i} = \beta_i + \sum_{j=1}^{\tau} \delta_{j,i}$, and the checking value $\overline{\gamma_i} = \gamma_i + \sum_{j=1}^{\tau} \epsilon_{j,i}$.
Finally, P_i deletes the old values β_i, γ_i and also the update value $\delta_i = \sum_{j=1}^{\tau} \delta_{i,j}$.

Observe that a new polynomial $h(x) := \sum_{i=1}^{\tau} h^{(i)}(x)$ is generated. Consequently, a new polynomial $\overline{p}(x) := p(x) + h(x)$ is implicitly created. This new polynomial has the same constant value of the old one: $\overline{p}_0 = p_0$, since $h_0 = \sum_{i=1}^{\tau} h_0^{(i)} = 0$. This constant value is the secret that is fixed. The idea is explained in figure in 8 (for simplicity, polynomials are drawn as lines).

4.2 Second Decentralized Refresh (RegenRefresh)

This method is based on the idea of regenerating the polynomials that had been created in the SecGen phase. Let us see the protocol.

The same set of $\tau \le n$ participants $P_1, \dots, P_\tau$ that had carried out the SecGen, collaborate to refresh shares and checking values for the players $P_1, \dots, P_n$. The secret remains p_0, the threshold t.

Player P_i, $i \in \{1, \dots, \tau\}$	
Private Input:	$p_0^{(i)}$
Public Input:	$\alpha, g, h, \{C_{0,i,0}\}_{1 \le i \le \tau}$
Private Output:	$(\overline{\beta}_{n+1}, \overline{\gamma}_{n+1})$
Public Output:	-

1. Each P_i for $i \in \{1, \dots, \tau\}$ generates a secret polynomial $\overline{p}^{(i)} \in \mathbb{F}_q[x]$ of degree $t-1$, by sampling the coefficients $\overline{p}_k^{(i)}$ uniformly at random in $\mathbb{F}_q$, with the restriction $\overline{p}_0^{(i)} = p_0^{(i)}$.
2. Each P_i samples another random polynomial $\overline{z}^{(i)} \in \mathbb{F}_q[x]$ of degree $t-1$ with $\overline{z}_0^{(i)} = z_0^{(i)}$, and uses its coefficients to compute and publish the commitments to the coefficients of their secret polynomial $\overline{p}^{(i)}$: $C_{i,k} = \mathrm{HCom}\left(\overline{p}_k^{(i)}; \overline{z}_k^{(i)}\right) \quad \forall k \in \{1, \dots, t-1\}$.
3. After having received every single commitment $C_{j,k}$, for $j \in \{1, \dots, \tau\}$ and $k \in \{0, \dots, t-1\}$, each P_i sends to each P_j the evaluations $\overline{\beta}_{i,j} = \overline{p}^{(i)}(\alpha^j)$ and $\overline{\gamma}_{i,j} = \overline{z}^{(i)}(\alpha^j)$.

4. Each P_i for $i \in \{1, \dots, \tau\}$ sends the pair $(\overline{\beta}_{i,j}, \overline{\gamma}_{i,j})$ also to every party P_j for $j \in \{\tau+1, \dots, n\}$.
5. By exploiting the homomorphic properties of the commitment scheme, each P_i for $i \in \{1, \dots, n\}$ checks the values received against the published commitments:
$$\mathrm{HCom}(\overline{\beta}_{j,i}; \overline{\gamma}_{j,i}) \overset{?}{=} C_{0,j,0} \cdot \prod_{k=1}^{t-1} (C_{j,k})^{(\alpha^i)^k}, \qquad \forall j \in \{1, \dots, \tau\}. \tag{7}$$
Recall that $C_{0,j,0}$ are commitments published and memorized in SecGen.
6. If all of these checks pass, each P_i sets its new share of the same secret as $\overline{\beta_i} = \sum_{j=1}^{\tau} \overline{\beta}_{i,j}$, and the checking value $\overline{\gamma_i} = \sum_{j=1}^{\tau} \overline{\gamma}_{i,j}$.
Finally, P_i deletes the old values β_i, γ_i.

Observe that a new polynomial $\overline{p}(x) \coloneqq p(x) + h(x)$ is implicitly generated. This new polynomial has the same constant value of the old one: $\overline{p}_0 = p_0$, since $\overline{p}_0^{(i)} = p_0^{(i)}$ for $i \in \{1, \dots, \tau\}$. This constant value is the secret that is fixed. The idea is explained in figure in 8 (for simplicity, polynomials are drawn as lines).

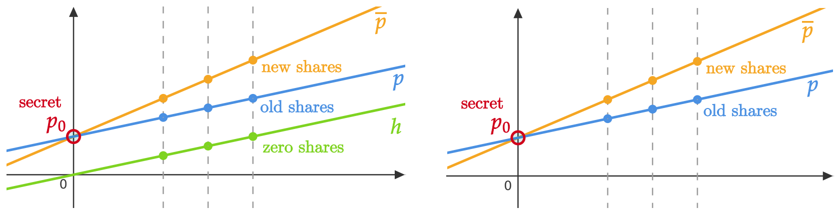

Fig. 8. Representations for ZeroSharesRefresh (on the left) and RegenRefresh strategies (on the right) in the case of polynomials of degree 1 (2-out-of-3 reconstruction policy).

Remark 7. The two methods are completely equivalent. Indeed, in both cases, a new polynomial $\overline{p}(x)$ is generated, and the new shares are evaluations of this polynomial. In the first method, the difference polynomial between $p(x)$ and $\overline{p}(x)$ (named $h(x)$) and its evaluations δ_i are explicitly used and exchange, while in the second their are only implicitly set. However, the information used for the second method are enough to recover all this "hidden" values. Indeed, each participant can recover their update value $\delta_i = h(\alpha^i)$ by computing $\overline{\beta}_i - \beta_i$. Moreover, as in the first method $t-1$ participants can collaborate to interpolate and find $h(x)$ (as explained in the next subsection), also in the second one they can compute $h(x)$:

1. Each participant computes $\delta_i = \overline{\beta}_i - \beta_i$.
2. They knows that there exists a polynomial $h(x)$ that passes through their evaluations (α^i, δ_i) plus the point $(0, 0)$.
3. They use these t information to interpolate and recover $h(x)$. This polynomial is such that $\overline{p}(x) = p(x) + h(x)$.

Security Formal security proofs for these two protocols are analogous to that of SecGen (see [1] for details). Here, we analyze security in more depth, considering two periods separated by a refresh phase and an adversary. Since the two methods are equivalent, we handle them in a unique way. Recall that $h(x)$ has degree $t-1$ and $h(0)=0$. Each P_i knows $\delta_i = h(\alpha^i)$. $t-1$ corrupted participants can collaborate to recover $h(x)$ by interpolating. Indeed, they know $t-1$ evaluations plus the info $h(0)=0$. Hence they recover also $h(\alpha^j)$ for $j \in \{1,\ldots,n\}$, all the update factors. In this way, if they find a old share β_i, they can compute the corresponding new share $\overline{\beta}_i$ by computing $\overline{\beta}_i = \beta_i + h(\alpha^i)$. Knowing this new share, they can recover the secret. Security doesn't seem to hold up anymore. However, the assumption requires that, during each period, less than t participants can not recover any info about the secret. In this case, $t-1$ participants can recover the polynomial $h(x)$, but they still need t shares to recover the secret. Therefore, the security assumption still holds. More formally, suppose I is the set of indexes that the attacker has corrupted, obtaining their shares. Then the participants execute a refresh phase and a new period starts. Now suppose that J is the set of indexes that the attacker corrupts after the refresh phase, obtaining their new shares. Trivially, suppose $|I \cap J| \leq t-1$, otherwise the attacker can already recover the secret. If, $|I \cap J| = t-1$, the attacker can compute $h(x)$, but needs a t-th share to recover the secret.

Remark 8. It is important that at the end of the refresh phase, each participant deletes the old share β_i. Otherwise, once the attacker corrupts a participant P_i, they recover both the old and the new shares (β_i and $\overline{\beta}_i$ respectively). In this way, whenever $|I \cup J| \geq t$, the attacker has enough old shares to recover the secret. Moreover, in the first method, it is fundamental that each participant deletes also the update value δ_i. Indeed, whenever $|J| \geq t-1$ and there exists $j \in I \setminus J$, the attacker can use the update values of J to interpolate and compute $h(x)$, then compute δ_j and finally $\overline{\beta}_j = \beta_j + \delta_j$. At the end the attacker can use the new shares in J plus $\overline{\beta}_j$ to recover the secret.

4.3 Third Decentralized Refresh (MatrixRefresh)

The two decentralized refresh protocols that we have seen to be secure, have some limitations. These two methods are based on the idea of computing new shares from which the secret can be recovered executing the same protocol (SecRec). More precisely, the old and the new shares are linked to the secret through the fixed matrix G: Vandermonde matrix that encodes the coefficients of the polynomial p into a vector of shares. Consequently, a set of t distinct old shares can be used to recover the secret, even in a future period. In this way, forward secrecy against an adversary that stoles a sufficient number of past shares is not granted. To avoid this limitation is necessary to change the matrix G that links secret and shares. Now we propose and analyze a different approach that tries to solve this problem. At the end, we will discover that achieving this goal, we introduce a new limitation that was not present in the two methods discussed above. The idea is to periodically change both the shares and the matrix G that encodes the secret. In this way, old shares can not be used anymore to recover the secret. Past information becomes useless, even if the quantity exceeds the threshold t. In this protocol, named MatrixRefresh, new shares are produced and distributed using a new matrix, past info are not used. The new matrix is basically the one of the previous epoch premultiplied with a random invertible matrix. Figure 9 shows the idea.

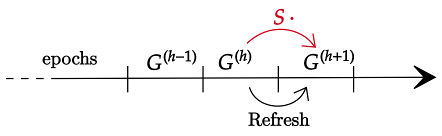

Fig. 9. MatrixRefresh general idea.

Remark 9. This can be convenient even in a context where we no longer trust a participant, so we don't give them the new share, and we are certain they won't be able to use their past information.

The new goal of security is: *In the current epoch, less than t shares provide no information about the secret and past shares provide no information about the secret even if the quantity exceeds the threshold t.*

Remark 10. We will see that this assumption holds except for the shares of the first epoch, the one that starts with SecGen. Indeed, the matrix G used initially must be stored through all the future epochs. For this reason we will suggest to execute a MatrixRefresh protocol few time immediately after SecGen.

Remark 11. Old versions of the generator matrix G *must* be forgotten and erased by everyone in order to achieve the desired security.

MatrixRefresh Protocol Time is divided into epochs, each one corresponds to a value of a counter. Between two consecutive epochs there is a refresh protocol that ends the old period and starts the new one. Each epoch has two fixed matrices that are updated through the refresh. During epoch h must be stored (see observation 13):

1) The Vandermonde matrix G used in SecGen: $G = \left[\alpha^{j\cdot k}\right]_{k\in\{0,\dots,t-1\},\, j\in\{1,\dots,n\}}$.
2) The matrix $G^{(h)}$, that is the matrix that encodes the secret (constant term of the polynomial p) into the shares of epoch h (named $\overline{\beta}_i^{(h)}$). In particular it holds: $(p_0,\dots,p_{t-1})\cdot G^{(h)} = (\overline{\beta}_1^{(h)},\dots,\overline{\beta}_n^{(h)})$.
3) The matrix S_{tot}, that is such that $G^{(h)} = S_{tot}\cdot G$.

Suppose that we are in epoch h and we want to start a new epoch $h+1$. t participants $P_{j_1},\dots,P_{j_t}$ ($J=\{j_1,\dots,j_n\}$) collaborate to refresh shares of all the participants $P_1,\dots,P_n$.

Player P_ℓ, $\ell \in \{j_1, \dots, j_t\}$	
Private Input:	$(\beta^{(h)}_{j_\ell}, \gamma^{(h)}_{j_\ell})$
Public Input:	$G, G^{(h)}, S_{tot}, \alpha, g, h, \{C_{0,i,k}\}_{\substack{1 \le i \le \tau \\ 0 \le k \le t-1}}, \{C^{(h)}_{j_\ell,J,i,k}\}_{\substack{1 \le \ell \le t \\ 1 \le i \le \tau \\ 1 \le k \le t}}$
Private Output:	$(\beta^{(h+1)}_{j_\ell}, \gamma^{(h+1)}_{j_\ell})$
Public Output:	$G^{(h+1)}, S_{tot}, \{C^{(h+1)}_{j_\ell,J,i,k}\}_{\substack{1 \le \ell \le t \\ 1 \le i \le \tau \\ 1 \le k \le t}}$

1. A matrix S with elements in $\mathbb{F}_q$ of dimensions (t, t), random, invertible is generated.
2. The two public matrices are updated: $G^{(h+1)} = S \cdot G^{(h)}$, $S_{tot} = S \cdot S_{tot}$. S is deleted.

The following steps are executed n times, one for each P_i for $i \in \{1, \dots, n\}$ that receives their share.

3. Each P_{j_ℓ} computes $\overline{f}(\beta^{(h)}_{j_\ell}, i, J, \ell) := \beta^{(h)}_{j_\ell} \cdot e_\ell \cdot {G^{(h)}}_J^{-1} \cdot G_i^{(h+1)}$ and $\overline{f}(\gamma^{(h)}_{j_\ell}, i, J, \ell) := \gamma^{(h)}_{j_\ell} \cdot e_\ell \cdot {G^{(h)}}_J^{-1} \cdot G_i^{(h+1)}$.
4. Each P_{j_ℓ} generates random values $b_{i,J,\ell,k}, z_{i,J,\ell,k} \in \mathbb{F}_q$, for $k \in \{1, \dots, t\} \setminus \{\ell\}$ and sets $b_{i,J,\ell,\ell} := \overline{f}(\beta^{(h)}_{j_\ell}, i, J, \ell) - \sum_{k=1,k\neq\ell}^{t} b_{i,J,\ell,k}$ and $z_{i,J,\ell,\ell} := \overline{f}(\gamma^{(h)}_{j_\ell}, i, J, \ell) - \sum_{k=1,k\neq\ell}^{t} z_{i,J,\ell,k}$.
5. Each P_{j_ℓ} computes and publishes the commitments $C^{(h+1)}_{i,J,\ell,k} := \mathrm{HCom}(b_{i,J,\ell,k}; z_{i,J,\ell,k}) \quad \forall k \in \{1, \dots, t\}$.
6. After having received every $C^{(h+1)}_{i,J,\ell,k}$ for $\ell, k \in \{1, \dots, t\}$, each P_{j_ℓ} checks the coherence of the values received with the previous epoch:
$$\prod_{k=1}^{t} C^{(h+1)}_{i,J,\ell,k} \stackrel{?}{=} \left(\prod_{w=1}^{t} \prod_{k=1}^{t} C^{(h)}_{j_\ell,J,w,k} \right)^{e_\ell \cdot {G_J^{(h)}}^{-1} \cdot G_i^{(h+1)}} \qquad \forall \ell \in \{1, \dots, t\}, \tag{8}$$
where $C^{(h)}_{j_\ell,J,w,k}$ are the commitments published during the previous refresh execution, with an abuse of notation since the set J could be different.
Moreover, they check the coherence of each value received with the SecGen initial phase:
$$\prod_{k=1}^{t} C^{(h+1)}_{i,J,\ell,k} \stackrel{?}{=} \left(\prod_{k=0}^{t-1} \left(\prod_{j=1}^{\tau} C_{0,j,k} \right)^{g^{(h)}_{k,j_\ell}} \right)^{e_\ell \cdot {G_J^{(h)}}^{-1} \cdot G_i^{(h+1)}} \qquad \forall \ell \in \{1, \dots, t\}, \tag{9}$$
where $C_{0,j,k}$ are the commitments published during the SecGen phase and $g^{(h)}_{k,j_\ell}$ is the element of $G^{(h)}$ in the k-th row, j_ℓ-th column.
Lastly, they check the coherence of the values received (all together) with the SecGen initial phase:
$$\prod_{k=1}^{t} \prod_{\ell=1}^{t} C^{(h+1)}_{i,J,\ell,k} \stackrel{?}{=} \prod_{k=0}^{t-1} \left(\prod_{j=1}^{\tau} C_{0,j,k} \right)^{g^{(h+1)}_{k,i}} . \tag{10}$$

7. If everything checks out, P_{j_ℓ} sends to each P_{j_k} the values $b_{i,J,\ell,k}, z_{i,J,\ell,k}$ for $\ell, k \in \{1, \dots, t\}$.
8. Each P_{j_ℓ} checks
$$\mathrm{HCom}(b_{i,J,k,\ell}, z_{i,J,k,\ell}) \overset{?}{=} C^{(h+1)}_{i,J,k,\ell} \qquad \forall k \in \{1, \dots, t\}, \tag{11}$$
and sets $b_{i,J,\ell} \coloneqq \sum_{k=1}^{t} b_{i,J,k,\ell}$, $z_{i,J,\ell} \coloneqq \sum_{k=1}^{t} z_{i,J,k,\ell}$.
9. Each P_{j_ℓ} sends $b_{i,J,\ell}$ and $z_{i,J,\ell}$ to P_i.
10. P_i checks the values received:
$$\mathrm{HCom}(b_{i,J,\ell}, z_{i,J,\ell}) \overset{?}{=} \prod_{k=1}^{t} C^{(h+1)}_{i,J,k,\ell} \qquad \forall \ell \in \{1, \dots, t\}, \tag{12}$$
and checks 8, 9, 10.
If everything checks out, P_i sets their new share and checking value: $\beta_i^{(h+1)} \coloneqq \sum_{\ell=1}^{t} b_{i,J,\ell}$ and $\gamma_i^{(h+1)} \coloneqq \sum_{\ell=1}^{t} z_{i,J,\ell}$.
11. At the end of the protocol $G^{(h)}$ is deleted or overwritten.
Therefore, there is no longer any memory of $G^{(h)}$ and the old S_{tot}.

Remark 12. Correctness of the new share:
$$\begin{aligned}\beta_i^{(h+1)} &= \sum_{\ell=1}^{t} b_{i,J,\ell} = \sum_{\ell=1}^{t}\sum_{k=1}^{t} b_{i,J,k,\ell} = \sum_{k=1}^{t}\sum_{\ell=1}^{t} b_{i,J,k,\ell} = \sum_{k=1}^{t} \overline{f}(\beta_{j_h}^{(h)}, i, J, k) \\ &= \sum_{k=1}^{t} \beta_{j_k}^{(h)} \cdot e_k \cdot {G_J^{(h)}}^{-1} \cdot G_i^{(h+1)} = (p_0, \dots, p_{t-1}) \cdot G_i^{(h+1)}.\end{aligned}$$

Remark 13. In the protocol described in [1], it is always easy to build G_J, G_i for general J, i, following the construction explained in Definition 1. This is possible because the matrix G remain always fixed during time with the same Vandermonde structure. Now, we change the matrix G at each epoch, hence we are not able anymore to write G_J, G_i for general J, i. For this reason, each time we need $G_J^{(h)}, G_i^{(h)}$ for some J, i, we need to do: $G_J^{(h)} = S_{tot} \cdot G_J$ or $G_i^{(h)} = S_{tot} \cdot G_i$. Therefore, we need to store both G and S_{tot}.

Security analysis First of all, it is fundamental that old versions of $G^{(h)}$ and S_{tot} are deleted, otherwise old shares can still be used to recover the secret. As mentioned before, not only the shares of the current epoch can be used, but also the starting ones, since G is known. For this reason, the advice is to execute a refresh as soon as possible, in order to reduce the time window during which initial shares can be stolen. Now let us compare methods ZeroSharesRefresh and RegenRefresh with the new method MatrixRefresh, introducing a more clear notation in which s denotes the fixed secret p_0.

– In methods ZeroSharesRefresh (4.1) and RegenRefresh (4.2), each share $\beta_i^{(h)}$ of some epoch h gives a relation: $\beta_i^{(h)} = (s, r_1^{(h)}, \dots, r_{t-1}^{(h)}) \cdot G_i$, where:

- $(r_1^{(h)}, \dots, r_{t-1}^{(h)}) = (p_1, \dots, p_{t-1})$ are $t-1$ random elements that are different for each epoch.
- G_i is a column of G, public and fixed over time.

Therefore, with t distinct (related to distinct parties) shares $(\beta_i^{(h)})_{i \in I}$ of the same epoch h (otherwise random elements are different), it is possible to solve the system that consists of t equations $\beta_i^{(h)} = (s, r_1^{(h)}, \dots, r_{t-1}^{(h)}) \cdot G_i,\ \forall i \in I$, and t unknowns $(s, r_1^{(h)}, \dots, r_{t-1}^{(h)})$, and recover the secret.
Therefore, *the attacker can exploit old shares in successive epochs.*

– In method 4.3, each share $\beta_i^{(h)}$ of some epoch h gives a relation: $\beta_i^{(h)} = (s, r_1, \dots, r_{t-1}) \cdot G_i^{(h)}$, where:
  - $(r_1, \dots, r_{t-1}) = (p_1, \dots, p_{t-1})$ are $t-1$ random elements that are fixed over time.
  - $G_i^{(h)}$ is a column of $G^{(h)}$, public and known only in epoch h, it changes over time.

  Therefore, old shares are useless and the only way to solve the system and recover the secret is to know t shares among the initial ones and the current ones.

Proposition 2. *The attacker can not exploit old shares in successive epochs.*

Proof. Suppose that we are in epoch h and an attacker gets t shares of a old epoch k. We prove that they are not able to recover any info about the secret. The attacker knows: $G^{(h)}$, S_{tot}, G, $\{\beta_i^{(k)}\}_{i \in I, |I|=t}$. In epoch k, the used matrix was $G^{(k)} = S \cdot G$, for some unknown $t \times t$ invertible matrix S. The attacker knows only G, while $G^{(k)}, S$ are not known in epoch h. We denote with $v = (s, r_1, \dots, r_{t-1})$ the vector with secret as first component and other components random. Exploiting the shares obtained, the system that consists of t equations $\beta_i^{(k)} = v \cdot S \cdot G_i,\ \forall i \in I$ holds. In this system, v, S are unknowns. We can reduce the unknowns by denoting $x := (x_1, \dots, x_t) := v \cdot S$. Now the system can be rewritten as $(\dots, \beta_i^{(k)}, \dots)_{i \in I} = (x_1, \dots, x_t) \cdot G_I$. This is a linear system of rank t and it can be solved knowing the t shares, hence the attacker can find the vector x. Knowing x, it is possible to compute every share of epoch k, but they are useless, as we are proving. In order to discover the secret, the attacker should recover at least the first component of v, solving $v \cdot S = (x_1, \dots, x_t)$, that consists of t equations $< v, S_i >= x_i,\ i \in \{1, \dots, t\}$, where S_i denotes the i-th column of S. This is a quadratic system. To summarize: the attacker knows the vector $(x_1, \dots, x_t)$, that is the image of the unknown secret vector v through the linear random invertible unknown map S. Hence, knowing only x, it is impossible for the attacker to discover v and in particular its first component.

Limits and Scenario In this last subsection, we will show a limit of the MatrixRefresh protocol. The newly discovered limitation in the protocol indicates a lack of security under a different adversarial model, which suggests that the protocol may not be generally advantageous except in a specific scenario.

We consider a new model of adversary different from the previous one: Continuous Shot, Non-Mobile, Adaptive adversary. In particular, suppose that the attacker has full control on a party P_i. Therefore, they see everything that P_i sees. We are going to show that, collecting t shares $\{\beta_i^{(h_k)}\}_{1\le k\le t}$ of P_i related to distinct epochs, the attacker *could* be able to recover the secret (it is theoretically possible but difficult). This is caused by the fact that the matrix S, that premultiplies G, could alter the dependency relationships between the columns $G_i^{(h)}$ of different epochs h (related to the same party P_i).

Conversely, in the two methods ZeroSharesRefresh (4.1) and RegenRefresh (4.2), the matrix G is fixed across epochs, and in particular it is the Vandermonde matrix described in 1, with $J = [1, \dots, n]$. Consequently, two shares of different epochs but related to the same P_i, correspond to the same column G_i and therefore they produce two dependent linear equations (one is useless).

Since the attacker controls P_i, we suppose that they know also all the matrices $G^{(h_1)}, G^{(h_2)}, \dots, G^{(h_t)}$ and the respective S_{tot}. We denote with $S^{(h_1)}$, $S^{(h_2)}, \dots, S^{(h_t)}$ the matrices $t \times t$ such that $G^{(h_k)} = S^{(h_k)} \cdot G,\ \forall k \in \{1, \dots, t\}$.

Recalling the notation of the proof of Proposition 2, the system that consists of the following t equations holds:

$$\beta^{(h_k)} = v \cdot S^{(h_k)} \cdot G_i \qquad \forall k \in \{1, \dots, t\}. \tag{13}$$

The attacker knows everything except v. This is a linear system of easy solution only if the t equations are linearly independent. In general, it depends on the matrices $S^{(h_k)}$, but we will see at least an example with independent equations. The t equations are linearly independent whenever the equation $c_1(S^{(h_1)} \cdot G_i) + c_2(S^{(h_2)} \cdot G_i) + \dots + c_t(S^{(h_t)} \cdot G_i) = \mathbf{0}$ holds if and only if $(c_1, \dots, c_t) = (0, \dots, 0)$. Rewriting: $G_i(c_1 S^{(h_1)} + c_2 S^{(h_2)} + \dots + c_t S^{(h_t)}) = \mathbf{0}$, where G_i is the i-th column of the Vandermonde matrix G, hence it is not equal to the zero vector, while $c_1 S^{(h_1)} + c_2 S^{(h_2)} + \dots + c_t S^{(h_t)}$ is a matrix that can be either singular or non-singular.

In conclusion, whenever the matrices $\{S^{(h_k)}\}_{1\le k\le t}$ are such that every non-trivial (c_i not all zeros) linear combination of them gives a non-singular matrix, then the equations in 13 are linearly independent and so the attacker can recover the secret. Therefore, with MatrixRefresh method, we lose a security goal guaranteed by the structure of G: the uselessness of shares owned by the same party P_i.

Example 1. Suppose $t = 2$. We take $S^{(h_1)} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ and $S^{(h_2)} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Then we have $c_1 \cdot S^{(h_1)} + c_2 \cdot S^{(h_2)} = c_1 \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} + c_2 \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} c_1 + c_2 & c_1 \\ c_2 & c_1 + c_2 \end{pmatrix}$. We compute $\det \begin{pmatrix} c_1 + c_2 & c_1 \\ c_2 & c_1 + c_2 \end{pmatrix} = c_1^2 + c_2^2 + c_1 c_2 \neq 0, \forall (c_1, c_2) \neq (0, 0)$. Since the determinant is always different from zero, $S^{(h_1)}, S^{(h_2)}$ are such that every non-trivial linear combination of them gives a non-singular matrix, hence the columns $S^{(h_1)} \cdot G_i, S^{(h_2)} \cdot G_i$ are linearly independent for every $i \in \{1, \ldots, n\}$. Consequently, the related system in 13 can be resolved, recovering the secret.

Remark 14. $S^{(h_1)}, S^{(h_2)}$ are also such that there exists a matrix S, invertible $t \times t$, s.t. $S^{(h_2)} = S \cdot S^{(h_1)}$, indeed $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

In conclusion, in methods 4.1 and 4.2 we have *vertical security* (t shares of the same P_i across different epochs are useless) but not *horizontal security* (with t distinct shares of a past epoch the secret is discovered), whereas in method 4.3, we have *horizontal security* (t distinct shares of a past epoch are useless) but not always *vertical security* (with t shares of the same P_i across different epochs could be possible to recover the secret), as explained in figure 10.

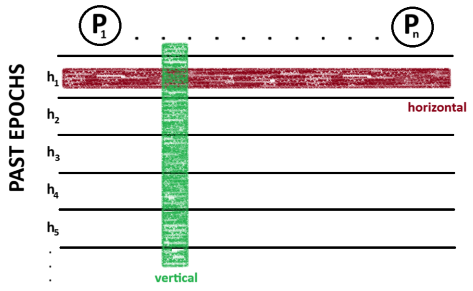

Fig. 10. Horizontal and vertical security for Refresh methods.

To sum up, MatrixRefresh method 4.3 has a vertical lack of security that makes it unsuitable for many decentralized contexts. A potential scenario where it could be applied is an *embedded system* in which the shares from various epochs are the only values stored locally and privately by each participant, while the $G^{(h)}$ and S_{tot} matrices are not saved. These matrices are managed by a resource-constrained processor dedicated solely to computations that can also manage the generation of the matrix S (step 1 of protocol 4.3). In this way, even if a participant is corrupted and controlled over time, $G^{(h)}$ and S_{tot} matrices remain unknown, thus ensuring vertical security.

Complexity Analysis Table 3 summarizes the complexity analysis for ZeroSharesRefresh, RegenRefresh, MatrixRefresh following the notation introduced in 2.6.

		ZeroSharesRefresh	RegenRefresh	MatrixRefresh
Communic.	**Priv**	τn	τn	$nt^2 + nt$
	Pub	$\tau(t-1)$	$\tau(t-1)$	nt^2
Initial DS	**Priv**	2 in $\mathbb{F}_q$	1 in $\mathbb{F}_q$	2 in $\mathbb{F}_q$
	Pub	2 in $\mathbb{G}$ 1 in $\mathbb{F}_q$	1 in $\mathbb{F}_q$ $(\tau + 2)$ in $\mathbb{G}$	$(2tn + t^2 + 1)$ in $\mathbb{F}_q$ $(t^2\tau + t\tau + 2)$ in $\mathbb{G}$
Runtime DS	**Priv**	$2(t-1+\tau)$ in $\mathbb{F}_q$	$2(t-1+\tau)$ in $\mathbb{F}_q$	$(2tn + 2n)$ in $\mathbb{F}_q$
	Pub	$\tau(t-1)$ in $\mathbb{G}$	$\tau(t-1)$ in $\mathbb{G}$	-
Final DS	**Priv**	2 in $\mathbb{F}_q$	2 in $\mathbb{F}_q$	2 in $\mathbb{F}_q$
	Pub	-	-	$(tn + t^2)$ in $\mathbb{F}_q$ $(t^2\tau)$ in $\mathbb{G}$
Rand. Gen.		$2\tau(t-1)$	$2\tau(t-1)$	$t^2 + 2tn(t-1)$
Comp. Cost		$2\tau(t-1) + n\tau(t+1)$ (exp in $\mathbb{G}$)	$2\tau(t-1) + n\tau(t+1)$ (exp in $\mathbb{G}$)	$t^3n + 8t^2n + 5tn$ (exp in $\mathbb{G}$)

Table 3. Complexity Analysis of ZeroSharesRefresh, RegenRefresh,MatrixRefresh.

5 Conclusions

In this paper, starting from a basic threshold policy, we extended a secret sharing protocol to a general threshold-gate access tree. In particular, we found a recursive strategy for describing the generator matrix that encodes the secret vector into the final shares of the tree, extending Shamir's strategy layer-by-layer. Furthermore, we explored the introduction of a Refresh phase to the secret sharing protocol, in order to achieve proactive security. We contributed to the current state of the art by describing three distinct methods: ZeroSharesRefresh, RegenRefresh and MatrixRefresh. We proved that all of them are secure against a snapshot, mobile, adaptive, active and computationally bounded adversary. We also demonstrated that the first two methods are perfectly equivalent and secure even against a continuous-shot, non-mobile, and adaptive adversary (vertical security), though they lack forward secrecy against a snapshot, mobile, and adaptive adversary. In contrast, the third method is secure with respect to forward secrecy against a snapshot, mobile, and adaptive adversary (horizontal security), but it is not secure against a continuous-shot, non-mobile, and adaptive adversary. We also performed an in-depth complexity analysis for each phase of the protocol, taking into account computational time, required memory, communication rounds, and the number of random generations. This work represents a significant contribution to the study of secure secret sharing techniques that remain resilient over time and can be adapted to complex access structures. Future research could explore integrating post-quantum cryptography to ensure security against quantum computing capable adversaries.

References

1. Battagliola, M., Longo, R., Meneghetti, A.: Extensible decentralized secret sharing and application to schnorr signatures. Cryptology ePrint Archive, Paper 2022/1551 (2022), https://eprint.iacr.org/2022/1551
2. Beimel, A.: Secure schemes for secret sharing and key distribution. PhD thesis, Israel Institute of Technology, Technion (1996)
3. Chu, H., Gerhart, P., Ruffing, T., Schröder, D.: Practical schnorr threshold signatures without the algebraic group model. In: Annual International Cryptology Conference. pp. 743–773. Springer (2023)
4. Di Nicola, V., Longo, R., Mazzone, F., Russo, G.: Resilient custody of crypto-assets, and threshold multisignatures. Mathematics **8**(10), 1773 (2020)
5. Feldman, P.: A practical scheme for non-interactive verifiable secret sharing. In: 28th Annual Symposium on Foundations of Computer Science (sfcs 1987). pp. 427–438. IEEE (1987)
6. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. Journal of Cryptology **20**, 51–83 (2007)
7. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 89–98 (2006)
8. Komlo, C., Goldberg, I.: Frost: Flexible round-optimized schnorr threshold signatures. In: Dunkelman, O., Jacobson, Jr., M.J., O'Flynn, C. (eds.) Selected Areas in Cryptography. pp. 34–65. Springer International Publishing, Cham (2021)
9. Liu, Z., Cao, Z., Wong, D.S.: Efficient generation of linear secret sharing scheme matrices from threshold access trees. Cryptology ePrint Archive, Paper 2010/374 (2010), https://eprint.iacr.org/2010/374
10. Nikov, V., Nikova, S.: On proactive secret sharing schemes. In: Handschuh, H., Hasan, M.A. (eds.) Selected Areas in Cryptography. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
11. Pedersen, T.P.: A threshold cryptosystem without a trusted party. In: Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10. pp. 522–526. Springer (1991)
12. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology — CRYPTO '91. pp. 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
13. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
14. Soltani, R., Nguyen, U.T., An, A.: Decentralized and privacy-preserving key management model. In: 2020 International Symposium on Networks, Computers and Communications (ISNCC). pp. 1–7 (2020). https://doi.org/10.1109/ISNCC49221.2020.9297294
15. Stinson, D.R., Wei, R.: Combinatorial repairability for threshold schemes. Designs, Codes and Cryptography **86**, 195–210 (2018)
16. Zhang, Y., Wang, M., Guo, Y., Guo, F.: Towards dynamic and reliable private key management for hierarchical access structure in decentralized storage. In: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management. pp. 3371–3380 (2023)