Secret Sharing Schemes : A Fine-Grained analysis

Shion Samadder Chaudhury[1], Sabyasachi Dutta[2], and Kouichi Sakurai[3]

[1] Applied Statistics Unit,
Indian Statistical Institute, Kolkata, India
chaudhury.shion@gmail.com
[2] Department of Computer Science,
University of Calgary, Calgary, Canada
saby.math@gmail.com
[3] Faculty of Information Science and Electrical Engineering,
Kyushu University, Japan
sakurai@inf.kyushu-u.ac.jp

Abstract. In this paper we prove that embedding *parity* bits and other function outputs in share string enables us to construct a secret sharing scheme (over binary alphabet) robust against a resource bounded adversary. Constructing schemes robust against adversaries in higher complexity classes requires an increase in the share size and increased storage. By connecting secret sharing with the *randomized decision tree* of a Boolean function we construct a scheme which is robust against an infinitely powerful adversary while keeping the constructions in a very low complexity class, *viz.* AC^0. As an application, we construct a robust secret sharing scheme in AC^0 that can accommodate new participants (dynamically) over time. Our construction requires a new redistribution of secret shares and can accommodate a bounded number of new participants.

Keywords: Robust Secret Sharing, Circuit Complexity, Randomized Decision Tree Complexity, Share Redistribution.

1 Introduction

Classical Secret Sharing

Secret sharing schemes were proposed independently by Shamir [32] and Blakley [6] in 1979. They proposed schemes where any k (or more) out of n participants are qualified to recover the secret with $1 < k \leq n$. The resulting access structure is called a (k, n)-threshold access structure where k acts as a threshold value for being qualified. Both schemes were fairly efficient in terms of the size of the shares and computational complexity. Ito et al. [21] showed that it is possible to construct secret sharing schemes for any monotone (general) access structures but with exponential share sizes. Later, Karchmer et al. [22] provided scheme with share size is polynomial in the monotone span program complexity.

Secret Sharing in AC^0

The motivation to study secret sharing schemes that can be implemented by constant-depth circuits comes from two different sources. First, most well-known secret sharing schemes require computations that can not be implemented by constant-depth circuits (i.e. AC^0 circuits). For example, Shamir's scheme in [32] requires linear algebraic computations over finite field and hence cannot be computed in AC^0. Secondly, the visual secret sharing schemes introduced by Naor and Shamir [27] require only computation of OR during the secret reconstruction phase. This OR function can be implemented by AC^0 circuit. Recent work by Bogdanov et al. [7] considers the question of whether there exists secret sharing scheme such that both share generation algorithm and secret reconstruction algorithm are computable in AC^0. They considered a variant of threshold secret sharing scheme, known as *ramp* schemes where any k participants learn nothing about the secret but when all n participants collaborate together, they are able to reconstruct the secret. The scheme is called ramp because unlike classical secret sharing scheme there is a gap between the privacy threshold viz. k and reconstructability threshold viz. n. Their construction connects the idea of *approximate degree* of a function with the privacy threshold of a secret sharing scheme. Existing literature on the approximate degree lower bounds gives several secret sharing schemes in AC^0. Their schemes however achieve large privacy threshold $k = \Omega(n)$ when the alphabet size is $2^{poly(n)}$ and achieve $k = \Omega(\sqrt{n})$ for binary alphabets. The work of Bogdanov et al. [7] was followed up by the work of Cheng et al. [11] who achieved privacy threshold $k = \Omega(n)$ with binary alphabets by allowing negligible privacy error. They have also considered robustness of the schemes in presence of honest majority with privacy threshold $\Omega(n)$, privacy error $2^{-n^{\Omega(1)}}$ and reconstruction error $\frac{1}{poly(n)}$. Our constructions in this paper are based on the the techniques in the paper by Cheng et al. [11]. For robustness we use the schemes of Cheragchi [12].

Redistribution of secret shares

Many secret sharing schemes have been proposed where the access structure changes over time and the main aim is, without reconstructing the shared secret, to add or delete shareholders, to renew the shares, and to modify the conditions for accessing the secret. This important primitive of redistributing the secret was initially considered by Chen et al. [10], Frankel et al. [18] and Desmedt-Jajodia [16].

More formally, let us consider two sets of participants $\mathcal{P}$ and $\mathcal{P}'$ containing n and n' many participants respectively. Let us suppose that each participant P_j in $\mathcal{P}$ has received a share s_j of the secret value s. $\Gamma_{\mathcal{P}}$ denote the access structure that specifies which subsets of $\mathcal{P}$ are *authorized* to recover the secret s from their shares. The *goal* of redistribution is that *without* the help of the original dealer, the participants in $\mathcal{P}'$ will receive the shares of s in accordance with a possibly different access structure $\Gamma_{\mathcal{P}'}$. In the protocol, the participants in $\mathcal{P}$ act like virtual dealers, while participants in $\mathcal{P}'$ are the ones who receive shares.

Nojoumian-Stinson [28] proposed unconditionally secure share re-distribution schemes, in absence of a dealer, based on a previously existing VSS protocol of Stinson-Wei [35]. In their construction, they have assumed less than one-fourth of participants behave dishonestly and also that the number of participants is fixed throughout. Their work was improved upon by the work of Desmedt-Morozov [17] who relaxed the proportion of dishonest participants to one-third of the total population and also allowed the number of participants to change.

Secure Computation against moderately complex adversaries

An important requirement of cryptography is to protect not only information but also the computations that are performed on data. The traditional cryptographic approach has been based on computational tasks which are easy for the honest parties to perform and hard for the adversary. We have also seen a notion of moderately hard problems being used to attain certain security properties. Degwekar et al. [15] show how to construct certain cryptographic primitives in NC^1 [resp. AC^0] which are secure against all adversaries in NC^1 [resp. AC^0]. Ball et al. [5] present computational problems which are "moderately hard" on average. Continuing in this line Campanelli and Gennaro [9] prove that it is possible to construct secure computation primitives that are secure against *moderately complex* adversaries. They present definitions and constructions for the task of fully homomorphic encryption and Verifiable Computation in the fine-grained model. For possible applications of AC^0 secret sharing to secure broadcasting in presence of external adversaries we refer to Section 7.3 of [11].

.

1.1 Our Contribution

In this paper we consider the following:

A fine-grained analysis of robust secret sharing schemes: We prove that by embedding outputs of certain functions in the share string we can achieve robustness against computationally bounded adversaries. This step removes the requirement to compute the random permutation in the scheme of Cheng et al. [11] which in turn helps us to reduce the share size. We consider adversarial algorithms belonging to AC^0, $AC^0[p]$, ACC^0, TC^0 and PP classes and analyse the share sizes.

Connecting secret sharing schemes with the randomized decision tree complexity of a Boolean function: Extending the previous idea we embed a function with known randomized decision tree complexity in the string generated by the method of Cheng et al. [11]. We argue that the scheme is robust against an adversary who can see and modify a constant fraction of the share string. To keep the whole scheme AC^0-computable, it is ensured that the functions we choose can be computed in AC^0. In this case no assumption is made on the

resource bound of the adversary. Hence this idea can be applied for functions with known quantum query complexity and thereby to quantum adversaries too. As in the previous case, embedding function output bits removes the requirement to use the random permutation in the scheme of Cheng et al. [11].

Application As an application, we study redistribution of secret shares if a bounded number of new participant(s) need to be accommodated (dynamically) to the scheme constructed in the previous section. Our idea of redistribution uses *random partitions* and is different from the ones previously used in the literature. All the computations are in AC^0.

2 Preliminaries

We discuss some definitions and results that will be needed throughout the paper. We mainly adopt the notations and definitions of [7], [11].

For a positive integer n the set $\{1, 2, \ldots, n\}$ is denoted by $[n]$. Let $\mathcal{P}_n = [n]$ be a set of n participants. Let $2^{\mathcal{P}_n}$ denote the power set of $\mathcal{P}_n$. A collection $\mathcal{A} \subset 2^{\mathcal{P}_n}$ is said to be *monotone* if $A \in \mathcal{A}$ and $A \subset B$ imply $B \in \mathcal{A}$.

Definition 1. *(Access structure) $\mathcal{A} \subset 2^{\mathcal{P}_n}$ is called a monotone access structure if the collection $\mathcal{A}$ is monotone. Any subset A of $\mathcal{P}_n$ which are in $\mathcal{A}$ are called qualified sets and $F \notin \mathcal{A}$ are called forbidden.*

Definition 2. *(Threshold Access structure) Let $n \in \mathbb{N}$ and $0 < k \leq n$. A (k, n)-threshold access structure $\mathcal{A}$ on a participant set $[n]$ is defined by $\mathcal{A} = \{X \subset [n] : |X| \geq k\}$.*

2.1 Secret Sharing Scheme

In a secret sharing scheme there is a dealer who has a secret s, a set of participants $[n]$ and an access structure $\mathcal{A}$. The dealer shares the secret among the participants in such a way that any qualified set of participants can recover the secret but any forbidden set of participants has no information about the secret.

Definition 3. *(Secret Sharing Scheme) A secret sharing scheme $\mathcal{S}$ for an access structure $\mathcal{A}$ consists of a pair of algorithms $(Share, Rec)$. Share is a probabilistic algorithm that gets as input a secret s (from a domain of secrets S) and a number n, and generates n shares $\Pi_1^{(s)}, \Pi_2^{(s)}, \ldots, \Pi_n^{(s)}$. Rec is a deterministic algorithm that gets as input the shares of a subset B of participants and outputs a string. The requirements for defining a secret sharing scheme are as follow:*

1. *(Correctness) For every secret $s \in S$ and every qualified set $B \in \mathcal{A}$, it must hold that $Pr[Rec(\{\Pi_i^{(s)}\}_{i \in B}, B) = s] = 1$.*
2. *(Security) For every forbidden set $B \notin \mathcal{A}$ and for any two distinct secrets $s_1 \neq s_2$ in S, it must hold that the two distributions $\{\Pi_i^{(s_1)}\}_{i \in B}$ and $\{\Pi_i^{(s_2)}\}_{i \in B}$ are identical.*

The *share size* of a secret sharing scheme $\mathcal{S}$ is the maximum number of bits each participant has to hold in the worst case over all participants and all secrets.

Definition 4. *(Ramp Secret Sharing Scheme) A (k, l, n) ramp secret sharing scheme with $k < l \leq n$, on a set of n participants is such that any subset of participants of size greater than equal to l can recover the secret whereas, any subset of size less than k has no information about the secret.*

2.2 k-wise indistinguishability

A construction of K-wise independent generators based on unique neighbour expander graphs were proposed by Guruswami-Smith [20] . A set of n random variables, $X_1, ..., X_n$, is said to be k-wise independent(and uniform) if any k of them are independent(and uniformly distributed). For any $r, n, k \in \mathbb{N}$, a function $g : \{0,1\}^r \to \Sigma^n$ is a k-wise (uniform) independent generator, if for the uniform distribution U on $\{0,1\}^r$, the random variables $g(U) = \{Y_1, ..., Y_n\}$ are k-wise independent (and uniform).

Let Σ denote the set of alphabets. Two distributions μ and ν over Σ^n are called k-wise indistinguishable if for all subsets $S \subset [n]$ of size k, the projections $\mu|_S$ and $\nu|_S$ of μ and ν to the coordinates in S are identical. Thus, while sharing the secret bit 0 (resp. 1) if sampling is done using μ (resp. ν) then we wee a direct connection to the fact that any k participants gain no information about the secret bit. However, if there is a function $f : \Sigma^n \to \{0,1\}$ which can tell apart the distributions then f can be thought of as a reconstruction function. Of course, the gap between the privacy threshold k and the reconstructability threshold n makes the scheme a ramp scheme.

The definition is as follows.

Definition 5. *(Secret Sharing for 1-bit secret) An (n, k, r) bit secret sharing scheme with alphabet Σ, reconstruction function $f : \Sigma^r \to \{0,1\}$ and reconstruction advantage α is a pair of k-wise indistinguishable distributions μ and ν over Σ^n such that for every subset S of size r we have $Pr[f(\mu|_S) = 1] - Pr[f(\nu|_S) = 1] \geq \alpha$.*

2.3 Circuit Complexity

AC^0 is the complexity class which consists of all families of circuits having constant depth and polynomial size. The gates in those circuits are NOT, AND and OR, where AND gates and OR gates have unbounded fan-in. Integer addition and subtraction are computable in AC^0. It is also well known that calculating the *parity* of an input cannot be decided by any AC^0 circuit [19]. For any circuit C, the size of C is denoted by $size(C)$ and the depth of C is denoted by $depth(C)$. Recently, a lot of research [1], [2], [3], [5], [24] have been done focusing on possibilities of obtaining cryptographic primitives in low complexity classes e.g. AC^0 or NC^1. NC^1 is the class of decision problems decidable by uniform Boolean circuits with a polynomial number of gates of at most two inputs and

depth $O(logn)$. Some other important complexity classes we consider in this paper are :

- $AC^0[p]$: Let

$$MOD_p(x_1, ..., x_n) = \begin{cases} 0, & \text{if } \Sigma_i^n x_i = 0(modp) \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

  $AC^0[p]$ denote the set of functions $f : \{0,1\}^* \to \{0,1\}^*$ computable by constant-depth unbounded fan-in circuits of NOT, OR and MOD_p (p prime) gates. The following is well known.

  Theorem 1. *[31], [33] $Majority \notin AC^0[p]$ for all primes p.*

- ACC^0 : A language belongs to ACC^0 if it belongs to $AC^0[m]$ for some m. It is known that $AC^0 \subsetneq ACC^0$. It is also known that $AC^0 \subset TC^0$ and it is conjectured that ACC^0 cannot compute *majority* implying that the inclusion in TC^0 is strict. From the work of Smolensky [33] it is known that sub-exponential size AC^0 circuits augmented with MOD_m gates (such circuits when restricted to polynomial size define the class $ACC^0[m]$) cannot compute MOD_q if $(m, q) = 1$ and m is a prime power.
- TC^0 : TC^0 is the set of a functions computed by constant depth polynomial-size threshold circuits. Threshold gates are defined as :

$$g(x_1, ..., x_l) = \begin{cases} 1, & \text{if } \Sigma_i w_i x_i \geq \theta \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

  The relation between the above mentioned complexity classes is :

$$AC^0 \subsetneq AC^0[p] \subsetneq TC^0 \subset NC^1.$$

  We also have $TC^0 \subsetneq PP$.

- PP : PP is the class of decision problems solvable by a probabilistic Turing machine in polynomial time, with an error probability of less than $1/2$ for all instances. PP contains the complexity classes BPP and NP. Also, PP strictly contains the class TC^0. For our purposes it is important to note that for any integer k, $PP \not\subseteq SIZE(n^k)$ where $SIZE(n^k)$ denote the class of languages accepted by Boolean circuit families of size bounded by n^k.

For more on these complexity classes we refer the reader to [4], [36].

2.4 Statistical Distance

The statistical distance between two random variables X and Y over Σ^n for some alphabet Σ, is $SD(X;Y)$ which is defined as follows,

$$SD(X;Y) = \frac{1}{2} \sum_{a \in \Sigma^n} |Pr[X = a] = Pr[Y = a]|.$$

We say that X is $SD(X;Y)$-close to Y.

2.5 Minsky-Papert CNF function

The sharing function, Share, used in our construction is based on the CNF function given by Minsky-Papert [26] . This scheme can share one bit among n participants, with binary alphabet, privacy threshold $\Omega(n^{1/3})$ and perfect reconstruction.

2.6 Robust Secret Sharing Scheme

The $Share_C$ function in the construction is the same as the sharing function for secret sharing schemes based on error correcting codes. The construction first amends the secret with a tag using an AMD code (such as the one in [12], [14]). Then, it uses Shamir's scheme to encode the result into mn shares, for a carefully chosen integer parameter $m > 1$. Finally, the resulting shares are bundled into n groups of size m each which are distributed among the n participants. In other words, we use a variant of Shamir's scheme based on folded Reed-Solomon codes (instead of plain Reed-Solomon codes) combined with an AMD pre-code. This is used to provide robustness in the sense of error-detection.

2.7 Randomized decision tree complexity

Decision trees form a model for computing Boolean functions by successively reading the input bits until the value of the function can be determined. In this model, the only cost we consider the number of input bits queried. This allows us to study the complexity of computing a function in terms of its structural properties. Formally, a deterministic decision tree algorithm A on n variables is a binary tree in which each internal node is labeled with an input variable x_i, and the leaves of the tree are labeled by either 0 or 1. Each internal node has two outgoing edges, one labeled with 0, the other with 1. Every input $x = (x_1, ..., x_n)$ determines a unique path in the tree leading from the root to a leaf: if an internal node is labeled by x_i, we follow either the 0 or the 1 outgoing edge according to the value of x_i. The value of the algorithm A on input x, denoted by $A(x)$, is the label of the leaf on this unique path. Thus, the algorithm A computes a Boolean function $A : \{0,1\}^n \rightarrow \{0,1\}$. We define the cost $C(A, x)$ of a deterministic decision tree algorithm A on input x as the number of input bits queried by A on x. Let P_f be the set of all deterministic decision tree algorithms which compute f. The deterministic complexity of f is

$$D(f) = min_{A \in P_f} max_{x \in \{0,1\}^n} C(A, x).$$

In an extension of the deterministic model, we can also permit randomization in the computation. A randomized decision tree algorithm A on n variables is a distribution over all deterministic decision tree algorithms on n variables. Given an input x, the algorithm first samples a deterministic tree $B \in_R A$, then evaluates $B(x)$. The error probability of A in computing f is given by

$$max_{x \in \{0,1\}^n} Pr_{B \in_R A}[B(x) \neq f(x)].$$

The cost of a randomized algorithm A on input x, denoted also by $C(A,x)$, is the expected number of input bits queried by A on x. Let P_f^δ be the set of randomized decision tree algorithms computing f with error at most δ. The two-sided bounded error randomized complexity of f with error $\delta \in [0,1/2)$ is

$$R_\delta(f) = min_{A \in P_f^\delta} max_{x \in \{0,1\}^n} C(A,x).$$

We write $R(f)$ for $R_0(f)$. We know the exact randomized complexity of very few Boolean functions. For more on randomized decision tree complexity we refer the reader to [8], [13], [29], [25]. The functions for which exact values or bounds on the randomized complexity are known will be used in the paper. They are the $NAND_h$ and the *recursive majority* function denoted by $3MAJ$. $NAND_h$ denotes the complete binary tree of height h with $NAND$ gates, where the inputs are at the $n = 2^h$ leaves. The following was proved by Snir [34].

Theorem 2. *[34]* $R(NAND_h) \in O(n^c)$ *where* $c = log_2(\frac{1+\sqrt{33}}{4}) \approx 0.753$.

Let $MAJ(x)$ denote the Boolean majority function of its input bits. The ternary majority function $3MAJ_h$ is defined recursively on $n = 3^h$ variables, for every $h \geq 0$. We omit the height h when it is obvious from context. For $h = 0$ it is the identity function. For $h \geq 0$, let x be an input of length n and let $x^{(1)}, x^{(2)}, x^{(3)}$ be the

first, second, and third $n/3$ variables of x. Then

$$3MAJ_h(x) = MAJ(3MAJ_{h-1}(x^{(1)}), 3MAJ_{h-1}(x^{(2)}), 3MAJ_{h-1}(x^{(3)}))$$

In other words, $3MAJ_h$ is defined by the read-once formula on the complete ternary tree T_h of height h in which every internal node is a majority gate. We identify the leaves of T_h from left to right with the integers $1, ..., 3^h$. The known bound for the randomized complexity of $3MAJ$ was given by Magniez et al.[25].

Theorem 3. *[25] For all* $\delta \in [0,1/2]$, *we have*

$$(1/2 - \delta) \cdot 2.57143^h \leq R_\delta(3MAJ_h) \leq (1.007) \cdot 2.64944^h.$$

3 Main results and technical details

Here we mention that unless specified, all computations are done by taking the alphabets to be binary alphabets.

3.1 Fine grained analysis of robust secret sharing schemes

Overview : Scheme Computable in AC^0 We first describe the scheme of Cheng et.al.[11] which has the following steps. First divide the parties obtained into small blocks (see steps 1, 2 and 3 of Algorithm : Construction 1, section 3.1.3 of this paper), and then for each small block we use a *good* secret sharing

scheme based on error correcting codes. Suppose the adversary gets to see a constant fraction of the shares, then on average for each small bock the adversary also gets to see only a constant fraction of the shares. Thus the adversary only gets to learn the information from a constant fraction of the blocks. However, this is still not enough, since the outer protocol only has threshold $n(1)$. To solve this problem use a threshold amplification technique in the second step. Turn the inner protocol itself into another concatenated protocol (i.e., a larger outer protocol combined with a smaller inner protocol), and then apply a random permutation. Specifically, choose the size of the block mentioned above to be $O(log^2 n)$, apply a secret sharing scheme based on asymptotically good error correcting codes and obtain $O(log^2 n)$ shares. Divide these shares further into $O(logn)$ smaller blocks each of size $O(logn)$, and now apply a random permutation of these smaller blocks. If we use a slightly larger alphabet, we can now store each block together with its index before the permutation as one share. Note that we need the index information when we try to reconstruct the secret, and the reconstruction can be done in AC^0.

If the adversary modifies some indices, then we could run into situations where more than one block have the same index. To overcome this difficulty, the index is stored multiple times among the blocks in the second level. Specifically, after we apply the random permutation, for every original index we randomly choose $O(logn)$ blocks in the second level to store it. As the adversary can only corrupt a small constant fraction of the blocks in the second level, for each such block, we can correctly recover its original index with probability $1 - 1/poly(n)$ by taking the majority of the backups of its index. Thus with probability $1 - 1/poly(n)$ all original indices can be correctly recovered. In addition, we use the same randomness for each block to pick the $O(logn)$ blocks, except we add a different shift to the selected blocks. This way, we can ensure that for each block the $O(logn)$ blocks are randomly selected and thus the union bound still holds. Furthermore the randomness used here is also stored in every block in the second level, so that we can take the majority to reconstruct it correctly. We need to take majority for n inputs, which is not computable in AC^0. However, we note that by adjusting parameters we can ensure that at least say 2/3 fraction of the inputs are the same, and in this case it suffices to take *approximate majority*, which can be computed in AC^0.

Overview of our method : Robustness against AC^0 adversary Suppose the adversary is AC^0 powerful. Since we know that *parity* is not in AC^0, we compute the *parity* of the bits in each of the smaller blocks and store them at the end of the respective block. Since the blocks have size $O(logn)$, the *parity* of the blocks can be computed by AC^0 circuits. Now we take $O(logn)$ many of the *parity* bits computed before and compute their *parity*. So we have computed the *parity* of the all the elements of the larger block. Now replace each of the parities stored at the end of each of the smaller blocks with the *parity* of all the elements of the larger block. Repeat the steps for each of the larger block. Lastly,

change the positions of each of the computed parity bits to any *random position* inside the block. Clearly the whole operation can be done by AC^0 circuits.

The adversary sees (queries) a constant fraction of the string. Let us suppose, without loss of generality, that a complete large block is included in the fraction observed. We have stored the *parity* bits in random locations of the blocks, Since the adversary is AC^0-powerful and hence cannot compute *parity*, it cannot find out which bit is the *parity* bit. Since the parity bits go into the computation for the error correcting codes, they form a part of the secret. So we can conclude that the adversary cannot get any information about the secret. To safeguard against the adversary modifying some of the indices, we store the random location of the bits multiple times.

1. The advantage of this operation is that we no longer need the random permutations to fool the adversary. This means that we no longer need to store *all the original indices* multiple times in the string. We need to store the locations of only the *parity* bits multiple times in the block. This step reduces the size of the share string.
2. If the adversary only observes the fraction, then the previous argument goes through. On the other hand if the adversary modifies some of the shares, we can use asymptotically good error correcting codes to ensure robustness.
3. Contrary to [11], where robustness is information theoretic, the robustness in our case comes from the inability of the adversary to compute *parity*. Hence in a sense it is computational.

Proceeding similarly as before our strategy for adversaries with bounded power (in higher complexity classes) is to find a function which is not computable by the adversary. Then we compute the function by breaking it in small blocks and embed them in the secret string. Since the adversary cannot compute the function, it cannot derive any information of the secret by observing and/or modifying a constant fraction of the secrets.

Construction 1: Embedding PARITY in the share string We shall assume previous constructions using k-wise independent generators and the ones using asymptotically good error correcting codes. We construct the secret sharing scheme $(Share_1, Rec_1)$ as follows. For the sake of completeness we recall some notations adopted from Cheng et al. [11] which we use frequently in the paper. .

For any $n, k, m \in \mathbb{N}$ with $k, m \leq n$, alphabets Σ_0, Σ, let $(Share, Rec)$ be an (n, k) secret sharing scheme with share alphabet Σ, message alphabet Σ, message length m.

Let $(Share_C, Rec_C)$ be an (n_C, k_C) secret sharing scheme from Lemma 3.13 of [11] with alphabet Σ, message length m_C, where $m_C = \delta_0 n_C$, $k_C = \delta_1 n_C$, $n_C = O(logn)$ for some constants δ_0 and δ_1.

For any constant $a \geq 1$, $\gamma \in (0,1]$, the paper by Cheng et al. [11] constructs the following $(n_1 = O(n^a), k_1 = \Omega(n_1)$ secret sharing scheme $(Share_1, Rec_1)$ with share alphabet $\Sigma \times [n_1]$, message alphabet Σ, message length $m_1 = \Omega(n_1)$. For clarity we include the algorithm as in [11]. .

Algorithm : Construction 1

– The $Share_1$ function is as follows :- $Share_1 : \Sigma^{m_1} \to (\Sigma \times [n_1])^{n_1}$.

1. *Let* $\bar{n} = \Theta(n^{a-1})$ *with large enough constant factor.*
2. *(Independent generator step) :- Let* $g_\tau : \Sigma_0^{m\bar{n}} \to \Sigma^{m_1}$ *be the l-wise independent generator* where $l = \Omega(\frac{m\bar{n}log|\Sigma_0|}{log|\Sigma|})^{1-\gamma}$.
3. *For a secret* $x \in \Sigma^{m_1}$ *, we draw a string* $r = (r_1, ..., r_{\bar{n}})$ *uniformly from* $\Sigma_0^{m\bar{n}}$.
4. *Let* $y = (y_s, y_g)$*, where* $y_s = (Share(r_1), ..., Share(r_n)) \in (\Sigma^n)^{\bar{n}}$ *and* $y_g = g_\tau(r) \oplus x \in \Sigma^{m_1}$.
5. *Get* $\hat{y}_s \in (\Sigma^{m_C})^{n_s}$ *from* y_s *by parsing* $y_{s,i}$ *to be blocks each having length* m_C *for every* $i \in [\bar{n}]$*, where* $n_s = \lceil \frac{n}{m_C} \rceil \bar{n}$.
6. *Get* $\hat{y}_g \in (\Sigma^{m_C})^{n_g}$ *from* y_g *by parsing* y_g *to be blocks each having length* m_C*, where* $n_g = \lceil \frac{m1}{m_C} \rceil$.
7. *Compute*

$$(Share_C(\hat{y}_{s,1}), ..., Share_C(\hat{y}_{s,n_s}), Share_C(\hat{y}_{g,1}), ..., Share_C(\hat{y}_{g,n_g}))$$

   and parse it to be $y1 = (y1_1, ..., y1_{n_1})$*, where* $n_1 = (n_s + n_g)n_C$.
8. *(Generate a random permutation)* $\pi : [n_1] \to [n_1]$ *apply it on* $y1$ *and this is the output.*

– We modify the steps 7 and 8 of Construction 1. The first six steps stay the same. For brevity we denote $Share$ by Sh.

Algorithm 1a : Embedding parity

1. *Do steps 1 to 6 of Construction 1.*
2. *Compute* $parity(\hat{y}_{s,i})$ *and* $parity(\hat{y}_{g,j})$ *for all* i *and* j .
3. *We get the string*

$$(\hat{y}_{s,1}, parity(\hat{y}_{s,1})..., \hat{y}_{g,1}, parity((\hat{y}_{g,1}), ...).$$

4. *Change the location of the computed parities to any random position inside each block of the string in* 3.
5. *On the string obtained in 4 compute*

$$(Sh_C(\hat{y}_{s,1}), \ldots, Sh_C(\hat{y}_{s,n_s}), Sh_C(\hat{y}_{g,1}), ..., Sh_C(\hat{y}_{g,n_g}))$$

   and parse it to be $y1 = (y1_1, ..., y1_{n_1})$*, where* $n_1 = (n_s + n_g)n_C$.
6. $y1$ *is the output.*

– The reconstruction function Rec_1 is as follows :

Algorithm 1a : Reconstruction

1. *Compute* Rec_C *on all the elements of* $y1$ *to get* y_1.
2. *Retrieve all the parities from* y_1.
3. *Delete all the parities from* y_1. *to get* y_s *and* y_g.
4. *Apply Rec on every entry of* y_s *to get* r.
5. *Output* $g_\tau(r) \oplus y_g$.

– *Parity* can also be computed at the end. But it results in increased computation. We include the algorithm for completeness.

Algorithm 1b : Embedding parity at the end

1. *Do steps 1 to 7 Construction 1.*
2. *Compute* $parity((Sh_C(\hat{y}_{s,i}))$ *and* $parity((Sh_C(\hat{y}_{g,j}))$ *for all* i *and* j .
3. *We get the string*

$$(Sh_C(\hat{y}_{s,1}), parity((Sh_C(\hat{y}_{s,1}))..,Sh_C(\hat{y}_{g,1}), parity((Sh_C(\hat{y}_{g,1})),..)$$

   Parse it to be $y1 = (y1_1, ..., y1_{n_1})$.
4. *Compute* $parity(parity((Sh_C(\hat{y}_{s,1})), ..., parity((Sh_C(\hat{y}_{g,1})), ...) = p$.
5. *Replace each of* $parity((Sh_C(\hat{y}_{s,i}))$ *and* $parity((Sh_C(\hat{y}_{g,j}))$ *for all* i *and* j *in* $y1$ *with* p.
6. *Change the location of* p *to any random position inside each of* $y1_i$ *for each* i.
7. $y1$ *is the output.*

– The reconstruction function Rec_1 in this case is :

Algorithm 1a : Reconstruction :

1. *Retrieve all the* p*'s from* y_1.
2. *Delete all the* p*'s from* y_1. *Call the resulting string* $y1$.
3. *Compute* Rec_C *on all the elements of* $y1$ *to get* y_s *and* y_g.
4. *Apply Rec on every entry of* y_s *to get* r.
5. *Output* $g_\tau(r) \oplus y_g$.

Theorem 4. *$Share_1$ and Rec_1 can be computed by AC^0 circuits.*

Proof. As Share can be computed by an AC^0 circuit, y can be computed in AC^0. Since we are breaking the string in small blocks, $Share_C$ can also be computed in AC^0. The extra function which we are computing is the *parity* function. To keep the whole computation in AC^0, again we compute *parity* repeatedly in small blocks of size $O(logn)$ each and store the results. This extra storage assumption is necessary for our purpose. The reconstruction function can also be computed in AC^0. In our case the additional computations are retrieving the p's and deleting them from the strings. Both these can be computed in AC^0. Hence, both $Share_1$ and Rec_1 can be computed by AC^0 circuits.

Remark Irrespective of where *parity* is computed before computing $Share_C$ or after computing $Share_C$, the *parity* bits are embedded in random positions in each of the blocks. Since the adversary sees only a constant fraction of the string and it is AC^0 powerful, it cannot compute and find the parity bit. We can thereby conclude that the adversary cannot learn any information about the secret. Computing *parity* before $Share_C$ results in computing lesser number of parity bits but computing *parity* after $Share_C$ increases robustness while requiring to compute more number of *parity* bits.

Theorem 5. *If the reconstruction error of (Share; Rec) is η , then the reconstruction error of $(Share_1, Rec_1)$ is $n' = \bar{n}\eta$.*

Proof. The reconstruction is done by first retrieving the *parity* bits. These parity are then deleted. Reconstruction then proceeds by computing the Rec_C, Rec and the XOR functions as in [11]. We have, as $\forall i \in [\bar{n}]$; $Pr[Rec(Share(x_i)) = x_i] \geq 1 - \eta$, by the union bound, $Pr[Rec_1(Share_1(x)) = x] \geq 1 - \bar{n}\eta$.

Adversaries in higher complexity classes We now look at the cases where the adversary is more powerful. In the previous case we were able to use *parity* as XOR was associative. Hence we could compute *parity* in small blocks.

1. $AC^0[p]$-adversary :- Since it is known that $AC^0[p] \subset TC^0$ and the inclusion is proper we choose a problem in TC^0 that is not in $AC^0[p]$, namely the *majority* function and proceed as before. But *majority* not associative and we cannot compute it recursively as before. In this case we use a theorem by Cohen et.al. in [13].

   Theorem 6. *[13] There exists an algorithm A that given an integer n as input, runs in poly(n)-time and computes a circuit C on $m = 2^{\sqrt{logn}}$ inputs, with the following properties:*
   - *C consists only of $Maj3$ gates and no constants.*
   - *$size(C) = poly(n)$.*
   - *$depth(C) = O(logn)$.*
   - *$\forall x \in \{0,1\}^m$ it holds that $C(x) = Maj(x)$.*

   Here $Maj3$ denotes the *majority* of 3 inputs.

   Unlike the case of *parity*, we only compute *majority* beforehand as in Algorithm 1a.

   Algorithm 1c : Embedding majority

   (a) *Do steps 1 to 6 of Construction 1.*
   (b) *Compute majority$(\hat{y}_{s,i})$ and majority$(\hat{y}_{g,j})$ for all i and j .*

(c) *Change the location of the computed majorities to any random position inside each block of the string in* (b).

(d) *On the string obtained in* (c) *compute*

$$(Sh_C(\hat{y}_{s,1}), \ldots, Sh_C(\hat{y}_{s,n_s}), Sh_C(\hat{y}_{g,1}), ..., Sh_C(\hat{y}_{g,n_g}))$$

and parse it to be $y1 = (y1_1, ..., y1_{n_1})$, *where* $n_1 = (n_s + n_g)n_C$.

(e) $y1$ *is the output.*

Theorem 7. *$Share_1$ in Algorithm 1c can be computed by AC^0 circuits.*

Proof. In this case the number of inputs is $O(logn)$. So, denoting the variable in the size and depth in Theorem 6 as z, we have

$$z = 2^{(log(logn))^2}.$$

This function grows very slowly with n for sufficiently large n and $z \ll n$. By replacing each of the $Maj3$ gate in C with the equivalent circuit comprising of only AND, OR gates, we get a circuit C with $size(C) \ll poly(n)$ and $depth(C) \ll O(logn)$ which computes the *majority* of the block. Hence using sufficiently extra storage and a large constant depth, we can implement the whole computation in the required complexity class AC^0.

- This construction helps us in getting a secret sharing scheme which is robust against an $AC^0[p]$ adversary.
- Comparing against an AC^0 adversary, we see that for robustness against an $AC^0[p]$ adversary we need sufficiently more storage and hence an increase in the length of the share string.
- The reconstruction algorithm is same as the reconstruction of Algorithm 1a.
- Continuing this trend, one would require increased share sizes and storage as we go higher in the complexity ladder.

2. ACC^0- adversary : If we assume the *conjecture* that ACC^0 circuits cannot compute *majority*, we can proceed as in the previous case. Otherwise we choose the problem for the next step.
3. TC^0-adversary :- Since it is known that $TC^0 \subset PP$ and the inclusion is proper we choose a problem in PP that is not in TC^0. For this we refer the reader to section 6.2 of [36].

.

3.2 Secret sharing & Randomized decision tree complexity

We want to embed an *evasive* Boolean function in the secret string in the scheme of Cheng et al. [11]. By the property of *evasive* Boolean functions, every *deterministic decision tree algorithm* has to query all the n variables of the function. Any adversary that observes/queries less than n variables of the function inputs cannot infer any information of the output. The difficulty of this method is that *deterministic decision tree complexity* is a worst case measure. To avoid this situation we use the *randomized decision tree complexity* of a Boolean function. This measure considers the expected number of queries a *randomized decision tree* makes on an input to decide the output of the function. The randomization is a distribution on all the decision trees that compute the function under consideration. The difficulty of using *randomized decision tree complexity* is that while the *randomized complexity* of a large class of Boolean functions with $NAND$ gates is known , we know the exact randomized decision tree complexity of very few Boolean functions. In this context we use two functions for which bounds on the randomized complexity is known, namely the $NAND_h$-function and the $recursive - majority$ function. As it is noted in [11], if the adversary observes a constant fraction of the string, then on an average it observes a constant fraction of the blocks. By ensuring the constant fraction to be not too large, we can ensure that in some blocks the adversary sees less the number of expected bits required to compute/decide the output of the embedded function. Hence it cannot decide which bit is the output bit. Since the bits go into the $Share_C$ computation, it forms a part of the *share*. We can thereby conclude that the adversary cannot learn any information about the secret. While the $NAND_h$ function can be computed by AC^0 circuits, with an extra storage assumption and by computing in small blocks, we can compute the $recursive - majority$ function and the whole computation by AC^0 circuits. Using these functions removes the necessity to compute the random permutations which in turn reduces the size of the share string as we no longer have to store all the indices multiple times. Just like the case for *parity*, only storing the random locations of the function output is enough for our purpose. To maintain robustness against the adversary modifying the bits, we continue to use asymptotically good error correcting codes. We recall the definitions and the properties of the $NAND_h$ and the recursive majority $3MAJ$ function.

- Let $NAND_h$ denote the complete binary tree of height h with NAND gates, where the inputs are at the $n = 2^h$ leaves.
- Let $MAJ(x)$ denote the Boolean majority function of its input bits. The ternary majority function denoted by $3MAJ_h$ is defined recursively on $n = 3^h$ variables, for every $h \geq 0$. We omit the height h when it is obvious from context. For $h = 0$ it is the identity function. For $h \geq 0$, let x be an input of length n and let $x^{(1)}, x^{(2)}, x^{(3)}$ be the first, second, and third $n/3$ variables of x. Then

$$3MAJ_h(x) = MAJ(3MAJ_{h-1}(x^{(1)}), 3MAJ_{h-1}(x^{(2)}), 3MAJ_{h-1}(x^{(3)})).$$

The randomized complexity of these functions are:

1. [34] $R(NAND_h) \in O(n^c)$ where $c = log_2(\frac{1+\sqrt{33}}{4}) \approx 0.753$.
2. [25] For all $\delta \in [0, 1/2]$, we have

$$(1/2 - \delta) \cdot 2.57143^h \leq R_\delta(3MAJ_h) \leq (1.007) \cdot 2.64944^h.$$

Algorithm 2a: Embedding $NAND_h$

1. *Do steps 1 to 6 of Construction 1.*
2. *Compute the greatest integers h_i and $h_j$such that $2^{h_i} \leq |\hat{y}_{s,i}|$ and $2^{h_j} \leq |\hat{y}_{s,j}|$ for all i, j.*
3. *Truncate the strings $\hat{y}_{s,i}$ and $\hat{y}_{s,j}$ to lengths 2^{h_i} and 2^{h_j} respectively.*
4. *On the truncated strings compute $NAND_{h_i}(\hat{y}_{s,i})$ and $NAND_{h_j}(\hat{y}_{g,j})$ for all i and j .*
5. *Embed the computed $NAND_h$'s to any random position inside each block of the string obtained after step 1.*
6. *On the string obtained in 5 compute*

$$(Sh_C(\hat{y}_{s,1}), \ldots, Sh_C(\hat{y}_{s,n_s}), Sh_C(\hat{y}_{g,1}), ..., Sh_C(\hat{y}_{g,n_g}))$$

and parse it to be $y1 = (y1_1, ..., y1_{n_1})$, where $n_1 = (n_s + n_g)n_C$.
7. *$y1$ is the output.*

Algorithm 2b : Embedding $3MAJ_h$

1. *Do steps 1 to 6 of Construction 1.*
2. *Compute the greatest integers h_i and $h_j$such that $3^{h_i} \leq |\hat{y}_{s,i}|$ and $3^{h_j} \leq |\hat{y}_{s,j}|$ for all i, j.*
3. *Truncate the strings $\hat{y}_{s,i}$ and $\hat{y}_{s,j}$ to lengths 3^{h_i} and 3^{h_j} respectively.*
4. *On the truncated strings compute $3MAJ_{h_i}(\hat{y}_{s,i})$ and $3MAJ_{h_j}(\hat{y}_{g,j})$ for all i and j by recursively computing*

$$3MAJ_h(x) = MAJ(3MAJ_{h-1}(x^{(1)}), 3MAJ_{h-1}(x^{(2)}), 3MAJ_{h-1}(x^{(3)}))$$

and storing the intermediate results.
5. *Embed the computed $3MAJ_h$'s to any random position inside each block of the string obtained after step 1.*
6. *On the string obtained in 5 compute*

$$(Sh_C(\hat{y}_{s,1}), \ldots, Sh_C(\hat{y}_{s,n_s}), Sh_C(\hat{y}_{g,1}), ..., Sh_C(\hat{y}_{g,n_g}))$$

and parse it to be $y1 = (y1_1, ..., y1_{n_1})$, where $n_1 = (n_s + n_g)n_C$.
7. *$y1$ is the output.*

Reconstruction : The reconstruction function Rec_1 is as follows (it is same for both $3MAJ$ and $NAND_h$ functions) :

Algorithm

1. *Compute* Rec_C.
2. *Retrieve all the functional values from* y_1.
3. *Delete all the functional values from* y_1. *Call the resulting string* $y1$.
4. *to get* y_s *and* y_g.
5. *Apply Rec on every entry of* y_s *to get* r.
6. *Output* $g_\tau(r) \oplus y_g$.

Theorem 8. *$Share_1$ and Rec_1 can also be computed by AC^0 circuits.*

Proof. Since both the $NAND_h$ and the $3MAJ_h$ can be computed by AC^0 circuits, both the share and the reconstruction function can be computed in AC^0. Also embedding a bit at a random position in a string is equivalent to a random permutation and hence can also be computed by AC^0 circuits.

Theorem 9. *Robustness:- An adversary which sees half of the string cannot compute the $NAND_h$ and the $3MAJ_h$ bits in the string.*

Proof. From the above discussion we know that :- $R(NAND_h) \in O(n^{0.75})$ and $R(3MAJ_h) \geq (1/2)2.57^h$. We prove the result for the $NAND_h$ function. The proof for the $3MAJ$ function is similar. Let us suppose that the adversary sees half of the string. This means that on an average the number queries that the adversary makes in each block is less than the expected number of bits needed to be queried to compute the $NAND_h$ function. Since the function output bits are embedded in random positions inside the block, the adversary cannot compute the function bits and hence cannot get any information about the secret.

4 Application: Redistributing secret shares

In this section we construct a scheme for redistribution of secret shares. Our construction is computationally robust while it modifies the shares of some of the old participants. We use the scheme of Cheng et.al. [11] as a building block and thus to keep the computations in AC^0 we can accommodate only a bounded number of new participants.

Let us suppose that at a certain point the share string has been generated by embedding the function output of a function with known (or bounded) randomized decision tree complexity as in the previous section. At this stage a new participant arrives. We add this participant to the larger block. This keeps the size of the larger block $O(log^2 n)$. Store the additional information like the generation of the new participant and to which block it is added multiples times. Then the share of this participant is computed as follows :

Algorithm 3 : Share of a new participant

1. *Select a random participant from the old set of participants, say A*
2. *Compute a random partition of the share of A into two equal halves. Since a random parition can be obtained by taking a random permutation and dividing the string into two halves, we can conclude that a random partition can be computed in* AC^0.
3. *Divide the share of A as* $\{p(A_1), p(A_2)\}$. *Here* $p(A_1)$ *denotes the first half of the partitioned string of A and* $p(A_2)$ *denotes the second half of the partitioned string of A.*
4. *Take* $p(A_1)$, *and copy it as the first half of the share of t.*
5. *Choose another old participant from the remaining participants, say B.*
6. *Compute* $p(B_1)$ *and* $p(B_2)$ *as before.*
7. *The new share of A* is $\{p(B_1), p(A_2)\}$.
8. *The share of t* is $\{p(A_1), p(B_1)\}$.

As noted in [11], increasing the number of repeated characters does not affect security. Also in this case the adversary (seeing a constant fraction of the string) sees less than the expected number of bits to compute the output of the embedded function. Hence it cannot learn any information about the secret. In each of the random partitions the function output embedded before is also present. We repeat the above process when a group of participants arrive. In this case we might choose different pairs of participants for different newcomers. To reuse shares of older participants we can also divide the shares of old participants into smaller (not arbitrarily) groups.

4.1 Remark

We could have just divided the the share of A into two equal halves $\{A_1, A_2\}$. But since we have not used any random permutation, we cannot ensure if there are the function outputs embedded in the share of the new participant which hinders robustness. Also from the share of the new participant and B, the adversary can recover completely the share of A which cannot be allowed.

Algorithm: Reconstruction

1. *Restore the original shares of the corresponding old participants.*
2. *Delete the new participants.*
3. *Retrieve the function output bits and delete them* $y1$.
4. *Compute* Rec_C *on all the elements of* $y1$ *to get* y_s *and* y_g.
5. *Apply Rec on every entry of* y_s *to get* r.
6. *Output* $g_\tau(r) \oplus y_g$.

– Here we note that to keep the computations in AC^0, we cannot accommodate an arbitrary number of new participants. We have to ensure that the block sizes remain $O(log^2 n)$ and $O(logn)$ respectively.

4.2 Discussions

In this section we see that going to the complexity class NC^1 and using the ideas of Krawczyk [23] it is possible to reduce the share size of the robust secret sharing schemes. In [23] an m-threshold scheme is presented , where m shares recover the secret but $m-1$ shares give no (computational) information on the secret, in which shares corresponding to a secret S are of size $\frac{|S|}{m}$ plus a short piece of information whose length does not depend on the secret size but just in the security parameter. This paper uses the idea of $information\ dispersal$ introduced by Rabin [30]. In this scheme information is distributed among n active processors, in such a way that the recovery of the information is possible in the presence of m active processors (i.e. out of m fragments), where m and n are parameters satisfying $1 \leq m \leq n$. The scheme assumes that active processors behave honestly, i.e. returned fragments are unmodified. The basic idea is to add to the information, say a file F, some amount of redundancy and then to partition it into n fragments, each transmitted to one of the parties. Reconstruction of F is possible out of m (legitimate) fragments. Each distributed fragment is of length $\frac{|F|}{m}$ which is clearly space optimal.

The distribution scheme of Krawczyk [23] proceeds by choosing a random encryption key K. The secret S is encrypted using the encryption key K. Using the $Information\ Dispersal\ Algorithm$ the encrypted file E is partitioned into n fragments, $E_1, E_2, ..., E_n$. Using a perfect secret sharing scheme n shares for the key K are generated, denoted by $K_1, K_2, .., K_n$. Each participant P_i, $i = 1, ..., n$ gets the share $S_i = (E_i, K_i)$. The portion K_i is privately transmitted to P_i (e.g. using encryption or any other secure way).

To reconstruct, collect from m participants P_{i_j}, $j = 1, ..., m$ their shares $S_{i_j}, = (E_{i_j}, K_{i_j})$. Using IDA reconstruct E out of the collected values E_{i_j}, $j = 1, ..., m$. Using the perfect secret scheme recover the key K out of $K_{i_j}, j = 1, ..., m$. Finally decrypt E using K to recover the secret S.

The above scheme constitutes a computationally secure (n, m)- secret sharing scheme. Each share S_i is of length $\frac{|S|}{m} + |K|$. We note that this scheme can be constructed in the complexity class NC^1.

Recently Degwekar et.al [15] showed the existence of secure encryption in NC^1. With this assumption we see that in the class NC^1, it is possible to construct a robust secret sharing scheme which secure against any adversary and the share sizes substantially reduced.

5 Conclusion

In this paper we study robustness of a low complexity secret sharing scheme against bounded adversary. By connecting secret sharing with the *randomized decision tree* of a Boolean function we construct a scheme which is computationally robust against an infinitely powerful adversary (not resource bounded) while keeping the constructions in a very low complexity class, AC^0. As an application of the above we construct a robust secret sharing scheme in AC^0 that

can accommodate new participants (dynamically) over time. We do not consider deletion of parties but only the case when new parties join (sequentially) the scheme in absence of dealer. Our construction requires a new redistribution of secret shares and can only accommodate a bounded number of new participants.

References

1. Adi Akavia, Andrej Bogdanov, Siyao Guo, Akshay Kamath, and Alon Rosen. Candidate weak pseudorandom functions in $ac^0 \circ mod_2$. In *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 251–260, 2014.
2. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in nc^0. *SIAM J. Comput.*, 36(4):845–888, 2006.
3. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography with constant input locality. *J. Cryptology*, 22(4):429–469, 2009.
4. Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
5. Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Average-case fine-grained hardness. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:39, 2017.
6. G. R. Blakley. Safeguarding cryptographic keys. In *AFIPS 1979*, pages 313–317, 1997.
7. Andrej Bogdanov, Yuval Ishai, Emanuele Viola, and Christopher Williamson. Bounded indistinguishability and the complexity of recovering secrets. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 593–618, 2016.
8. Harry Buhrman and Ronald De Wolf. Complexity measures and decision tree complexity: A survey. *Theoretical Computer Science*, 288:2002, 2000.
9. Matteo Campanelli and Rosario Gennaro. Fine-grained secure computation. In *Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II*, pages 66–97, 2018.
10. Liqun Chen, Dieter Gollmann, and Chris J. Mitchell. Key escrow in mutually mistrusting domains. In *Security Protocols, International Workshop, Cambridge, United Kingdom, April 10-12, 1996, Proceedings*, pages 139–153, 1996.
11. Kuan Cheng, Yuval Ishai, and Xin Li. Near-optimal secret sharing and error correcting codes in ac^0. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, pages 424–458, 2017.
12. Mahdi Cheraghchi. Nearly optimal robust secret sharing. In *IEEE International Symposium on Information Theory, ISIT 2016, Barcelona, Spain, July 10-15, 2016*, pages 2509–2513, 2016.
13. Gil Cohen, Ivan Bjerre Damsgrd, Yuval Ishai, Jonas Klker, Peter Bro Miltersen, Ran Raz, and Ron D. Rothblum. Efficient multiparty protocols via log-depth threshold formulae, 2013.
14. Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, pages 471–488, 2008.

15. Akshay Degwekar, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Fine-grained cryptography. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 533–562, 2016.
16. Y. Desmedt and S. Jajodia. Redistributing secret shares to new access structures and its applications. In *George Mason University, Tech. Report ISSE-TR-97-01, July 1997*. ftp://isse.gmu.edu/pub/techrep/97 01 jajodia.ps.gz., 1997.
17. Yvo Desmedt and Kirill Morozov. Parity check based redistribution of secret shares. In *IEEE International Symposium on Information Theory, ISIT 2015, Hong Kong, China, June 14-19, 2015*, pages 959–963, 2015.
18. Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Optimal resilience proactive public-key cryptosystems. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 384–393, 1997.
19. Mike Furst, Jon Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Systems Theory*, 17:13–27, 1984.
20. Venkatesan Guruswami and Adam D. Smith. Optimal rate code constructions for computationally simple channels. *J. ACM*, 63(4):35:1–35:37, 2016.
21. M. Ito, A. Saio, and Takao Nishizeki. Multiple assignment scheme for sharing secret. *J. Cryptology*, 6(1):15–20, 1993.
22. Mauricio Karchmer and Avi Wigderson. On span programs. In *Proceedings of the Eigth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, May 18-21, 1993*, pages 102–111, 1993.
23. Hugo Krawczyk. Secret sharing made short. *Advances in Cryptology - CRYPTO 93*, pages 136–146, 1994.
24. Russell W. F. Lai, Giulio Malavolta, and Dominique Schröder. Homomorphic secret sharing for low degree polynomials. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, pages 279–309, 2018.
25. Frdric Magniez, Ashwin Nayak, Miklos Santha, and David Xiao. Improved bounds for the randomized decision tree complexity of recursive majority, 2010.
26. M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.
27. Moni Naor and Adi Shamir. Visual cryptography. In *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, pages 1–12, 1994.
28. Mehrdad Nojoumian and Douglas R. Stinson. On dealer-free dynamic threshold schemes. *Adv. in Math. of Comm.*, 7(1):39–56, 2013.
29. Ryan O'Donnell. *Analysis of Boolean Functions*. 2012.
30. Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36:335–348, 1989.
31. Alexander Razborov. Lower bounds for the size of circuits of bounded depth with basis ,. *Math. notes of the Academy of Science of the USSR*, 41(4):333–338, 1987.
32. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
33. Roman Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *In Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 87*, pages 77–82, 1987.
34. Marc Snir. Lower bounds for probabilistic linear decision trees. *Theoretical Computer Science*, 38:69–82, 1985.

35. Douglas R. Stinson and Ruizhong Wei. Unconditionally secure proactive secret sharing scheme with combinatorial structures. In *Selected Areas in Cryptography, 6th Annual International Workshop, SAC'99, Kingston, Ontario, Canada, August 9-10, 1999, Proceedings*, pages 200–214, 1999.
36. Heribert Vollmer. *Introduction to Circuit Complexity a Uniform Approach*. 1999.