Secret Sharing and Secure Computing from Monotone Formulae

Ivan Bjerre Damgård, Jonas Kölker, Peter Bro Miltersen*

Department of Computer Science, Aarhus University

Abstract

We present a construction of log-depth formulae for various threshold functions based on atomic threshold gates of constant size. From this, we build a new family of linear secret sharing schemes that are multiplicative, scale well as the number of players increases and allows to raise a shared value to the characteristic of the underlying field without interaction. Some of these schemes are in addition strongly multiplicative. Our formulas can also be used to construct multiparty protocols from protocols for a constant number of parties. In particular we implement black-box multiparty computation over non-Abelian groups in a way that is much simpler than previously known and we also show how to get a protocol in this setting that is efficient and actively secure against a constant fraction of corrupted parties, a long standing open problem. Finally, we show a negative result on usage of our scheme for pseudorandom secret sharing as defined by Cramer, Damgård and Ishai.

1 Introduction

1.1 A new Construction of Monotone Boolean Formulae

It is well known that there exist log-depth formulae using **and** and **or** gates that compute the majority function, this was shown by Valiant [7] using a probabilistic method. In this paper we first present a variant of of Valiant's construction that produces log-depth formulae for the majority function using $T(2,3)$ gates, where $T(k,m)$ denotes the Boolean function that outputs 1 if and only if at least k of its m inputs are 1. A slight extension of the argument shows that one can efficiently construct such a formula using a procedure that succeeds with overwhelming probability[1].

The formulae we construct can be used to construct new secret sharing schemes and protocols, as we explain below. None of of these constructions would follow from Valiant's original result. This is because the type of gate(s) used in the formula is important for the properties of the secret-sharing scheme or protocol we can build, and **and**/**or** gates do not gives us any of the properties we are after. We also note that existence of *some* formula for majority using $T(2,3)$ gates follows from the work of Hirt and Maurer [6], however that result leads to exponential size formulae.

*An upcoming version of this paper will include Gil Cohen, Yuval Ishai, Ran Raz and Ron Rothblum as coauthors and will contain several additional results.

[1]The construction appears in a set of lecture notes from 1992 by the third author and later appears to have been discovered independently by several other researchers. We have not been able to find it in any published paper, however.

In the standard model for computing with Boolean formulae, it is trivial that a formula for computing majority can be used to compute other thresholds by setting some of the inputs to constant values. However, to be useful in our context, a formula cannot use constants. To get the more general results we describe below, we therefore need to extend the method for constructing formulae to other basic gates than $T(2,3)$. This extension is new to the best of our knowledge.

1.2 New Secret-Sharing Schemes

A secret sharing scheme can be thought of as a probabilistic algorithm that takes a secret s chosen from a finite set and outputs a set of n shares of s. The scheme typically has a threshold $t < n$ such that any subset of the shares of size at most t contains no information on s, whereas s can be computed efficiently from any $t+1$ or more shares. A subset from which the secret can be computed is called qualified.

Secret sharing schemes are essential tools in multiparty computation (MPC) protocols. Here a set of n players wish to compute an agreed function on some inputs they each hold. We want the computation to be secure which roughly speaking means to ensure that the result is correct and that the result is the only new information released, even if up to t players are corrupted by an adversary. A standard approach to MPC is to secret share the inputs initially, give a share of each input to each player, and compute on the shares rather than the inputs. This leads to protocols secure against t corrupted players. The best known secret sharing schemes such as Shamir's are defined over finite fields, which leads to protocols that securely compute arithmetic circuits over the field. For instance, most of the computation one needs to do AES encryption can be naturally described by arithmetic over the field F_{2^8}.

In this paper, we construct a family of secret sharing schemes that have new useful properties. We first present the construction mentioned above of log-depth monotone formulae for the majority function, based on $T(2,3)$-gates. The construction is generalized to other basic gates, such as $T(2,4)$-gates, where the function computed by the large formula is the $T(0.232n, n)$ function. From these formulae, families of linear secret sharing schemes over any finite ring follow immediately from a variant of the construction of Benaloh and Leichter[2], and we can also get integer secret sharing schemes by a simple variant of the construction. If the formula we use computes the $T(k,n)$ threshold function, then the qualified sets will be those that have at least k members. We also get some additional useful properties:

Efficiency: The schemes are efficient, i.e., the number of field elements each player receives when a secret is shared increases polynomially with n, the number of players.

Multiplicativity: The scheme based on $T(2,3)$-gates is *multiplicative*, i.e., if two secrets a, b have been shared, then players can locally compute $c_1, \ldots, c_n$ such that $ab = \sum_i r_i c_i$, where the r_i are fixed and public. When using $T(2,4)$-gates, we get a *strongly* multiplicative scheme, i.e., if t is the size of a maximal unqualified set, then the reconstruction of ab by linear combination can be done by any set of $n-t$ players, i.e., the corrupted players cannot stop the computation.

Additive Reconstruction: The reconstruction of a secret takes place by simply adding the shares, and moreover, the r_i's from the definition of the (strongly) multiplicative property are all 1.

The multiplication property is necessary for use of a secret-sharing scheme in MPC. Note that although we could also get secret sharing schemes from Valiant's original formula construction, these schemes would not be multiplicative.

The additive reconstruction property comes in handy in MPC, as follows: Since the secret sharing schemes used are typically linear, secure addition and multiplication by constants can be done with only local operations, whereas multiplications require interaction. However, the mapping $x \mapsto x^p$ where p is the characteristic of the field is special in that it is an additive homomorphism, we have $(x+y)^p = x^p + y^p$. This now immediately implies that if x has been secret shared using one of our schemes, one can obtain shares in x^p by only local operations, every player raises each field element he holds to power p. Note that this does not work for schemes where reconstruction requires a linear combination using arbitrary coefficients (except if we work over a prime field, but here raising to power p is the identity).

In particular, the above means that using our schemes in characteristic 2, squaring is essentially for free.

Some known linear secret sharing schemes also have additive reconstruction, for instance so called replicated secret sharing (for details, see [3]). These are not efficient, however.

1.3 Black-Box Computation over Non-Abelian Groups

A related usage of monotone formulae in cryptography is to use them to build, not just secret sharing schemes, but multiparty computation protocols. This was suggested by Hirt and Maurer [6]. The idea is very similar to the way secret-sharing schemes are built: if the formula uses $T(2,3)$-gates, then one starts from a 3 party protocol, secure against one corrupted player. If the protocol has certain nice properties (that we consider later in detail), then based on the formula, one can construct an n-party protocol where n is the number of input bits to the formula. If we let subsets of players correspond to n-bit strings where a bit is set if the corresponding player is in the subset, then the protocol will be secure against corruption of those subsets that the formula rejects. If the formula has logarithmic depth, the resulting protocol will typically be efficient, i.e., run in time polynomial in n.

In [6], the goal was to compute Boolean or arithmetic circuits securely, and to have security, not just for so-called threshold adversaries, but for general adversary structures, where the sets the adversary may corrupt are not necessarily characterized by their size.

In this paper, we observe than even for the threshold case, the idea of using monotone formulae can be very useful, namely for the case of black-box secure computing over non-abelian finite groups. Here, the goal is to compute securely the product of some elements taken from a group G, and the protocol must work by using only the group operation, taking inverses and random sampling from G. This problem is complete for multiparty computation in general. This follows from a result by Barrington [1].

The black-box computing problem was considered in [5], where a passively secure protocol was obtained, secure against corruption of $t < n/2$ parties. The construction of the general protocol is highly non-trivial and uses some rather deep results on coloring of planar graphs.

Here, we observe that from a protocol for three players and our monotone formulas computing the majority function, we can obtain the same result using completely elementary methods. The three-player solution from [5] can be used here as a building block. However, some additional work is required because Hirt and Maurer's approach is based on having certain sets of players emulate the actions of "virtual" players in other instances of the protocol. Therefore, anything a player

does locally must be implemented by a secure protocol. The protocol from [5] only provides secure multiplication of secret-shared group elements, so we must construct new protocols for sampling random group elements and inverting a shared group element.

Our observation also opens the door to obtaining a protocol with active security, which was a main open problem in [5]. Active security was also considered very recently in [4], where an actively secure protocol for any so called Q3 adversary structure was presented. This is optimal as far as resilience is concerned, but in general the protocol is very inefficient: the complexity is polynomial in the size of the adversary structure which is usually exponential in the number of players. Building a protocol with complexity polynomial in the number of players with active security against a constant fraction of corrupted players was left as an open problem in [4].

In this paper we solve the problem by first building protocols for 12 players that is secure against 1 actively corrupted player. This can then be combined with a formula built from $T(2, 12)$ gates to get a polynomial time n-player protocol secure against active corruption of a constant fraction ≈ 0.017 of players. The threshold could be improved significantly if we could build a (set of) protocol(s) with the right properties for 4 players that is secure against 1 actively corrupted player, but so far we were not able to do this. Note that while the result from [4] implies a 4-player protocol for secure multiplication this is not enough by itself. In addition to the extra protocols for random sampling and inversion that we needed for the passive case, we now also need comparison and this must be all compatible with the secret sharing of group elements used by the multiplication. This can be done for the 12-player protocol we mentioned, but at the time of writing we do not yet know if can be done for the 4-player case.

1.4 A Negative Result for Share Conversion.

In the final part of the paper we consider the notion of pseudorandom secret sharing (PRSS) introduced in [3], which is a tool that allows players to generate consistent shares from some secret sharing scheme using only local computation, and where the secret is pseudorandom in the view of the adversary. In [3] a PRSS construction was proposed and a lower bound was shown demonstrating that PRSS cannot scale well with the number of players, if the target secret-sharing scheme is in a certain class. Our scheme from this paper is not in this class, and furthermore, it is based on so called replicated secret sharing which was the basis of the positive result on PRSS from [3]. It is therefore very natural to speculate that our scheme could be used to circumvent the lower bound. However, this turns out not to be the case, we show how to strengthen the bound from [3] to cover any secret-sharing scheme.

2 Formulae for the Majority Function

A 2-out-of-3 threshold gate is a function on 3 bits that outputs 1 if at least 2 of input bits are 1, and 0 otherwise. Now, the main result of this section is that first of all, formulas of 2-of-3 gates for the majority function of low height exist, and secondly that for sufficiently large height they are abundant. In both cases, the formulas have height $O(\log n)$, where $n = 2m + 1$ is the number of input bits.

Consider full monotone formulas of height d, i.e. with $\sum_{i=0}^{d-1} 3^i$ (inner) threshold gates and 3^d (leaf) input gates. We want to argue that if we pick a random such formula, i.e. given the height we uniformly choose an input bit position 3^d times, this randomly picked formula will compute the

majority function with good probability; from this, the two above results will follow.

Lemma 1. *For some constants a and b and every $d \geq a + b \log_2 k$, consider full formulas of 2-of-3 gates of height d. For any bit string v, a randomly chosen such formula computes the majority function on v with probability exceeding $1 - (\frac{1}{2})^{2^k}$.*

Proof. Let F be such a randomly picked formula of height d and let $v \in \{0,1\}^n$ be a fixed bit vector. Let p_d be the probability over the choice of F that F doesn't compute the majority function on input v; that is, $p_d := Pr(F(v) \neq MAJ(v))$.

We can express p_d recursively:

$$p_0 = Pr(v_i \neq MAJ(v)) \leq \frac{m}{2m+1} = \frac{1}{2} - \frac{1}{2n}$$

and

$$p_{d+1} = Pr(Th_3^2(F_1(v), F_2(v), F_3(v)) \neq MAJ(v)) = p_d^3 - 3p_d^2(1-p_d) = 3p_d^2 - 2p_d^3,$$

for some formulas F_1, F_2 and F_3, where Th_b^a is the a-of-b threshold function.

Consider then the function $f(x) = 3x^2 - 2x^3$ and the sequence $p_0, f(p_0), f(f(p_0)), \ldots$; this is the probability that F computes something wrong (i.e. not the majority function) and this probability quickly becomes small. More precisely, we want to show that $\lim_{i\to\infty} f^{(i)}(p_0) = 0$, with a precise analysis of the depth required to go below 2^{-k}.

Observe first that $f(x) - x$ has roots in $\{0, \frac{1}{2}, 1\}$ and so f has exactly that set of fixed points. Note also that the sequence starts in $p_0 \in [0, \frac{1}{2}[$, and that $f(\frac{1}{2} - \varepsilon) = \frac{1}{2} - \frac{3}{2}\varepsilon + 2\varepsilon^3$. Let $0 < \gamma < \frac{3}{2}$ and $\varepsilon_0 = (\frac{1}{2}(\frac{3}{2} - \gamma))^{\frac{1}{2}}$. Then for all $\varepsilon \in]0, \varepsilon_0[$ we have $p_d \leq \frac{1}{2} - \varepsilon \Rightarrow p_{d+1} = f(p_d) < \frac{1}{2} - \gamma\varepsilon$.

By induction, this means that after d iterations of f, $f^{(d)}(p_0) < \frac{1}{2} - \gamma^d \frac{1}{2n}$ whenever $\gamma^d \frac{1}{2n} < \varepsilon_0$ or equivalently $d < \log_\gamma 2\varepsilon_0 + \log_\gamma n$, so let $d_0 := \log_\gamma 2\varepsilon_0 + \log_\gamma n$. Then $f^{(\lceil d_0 \rceil)}(p_0) \leq \frac{1}{2} - \varepsilon_0$.

This is when we bound f by the chord through $(\frac{1}{2}, f(\frac{1}{2}) = \frac{1}{2})$ with slope γ. Next, we want to bound f by the chord through $(\frac{1}{6}, f(\frac{1}{6}) = \frac{2}{27})$ and $(\frac{1}{2}, \frac{1}{2})$, which has slope $\frac{23}{18}$. Note that when $\frac{1}{6} < \frac{1}{2} - \varepsilon < \frac{1}{2}$, $f(\frac{1}{2} - \varepsilon) = \frac{1}{2} - \frac{23}{18}\varepsilon$, and so after $d_1 := \log_{\frac{23}{18}} \frac{1}{6\varepsilon_0}$ iterations, $f^{(\lceil d_0 + d_1 \rceil)}(p_0) \leq \frac{1}{6}$. After k further steps, $f^{(\lceil d_0 + d_1 + k \rceil)} < (3 \cdot \frac{1}{6})^{2^k} = (\frac{1}{2})^{2^k}$ since $p_{d+1} < 3p_d^2$.

So, choose a γ such that $1 < \gamma < \frac{3}{2}$ and compute ε_0. Let $a \geq \log_\gamma 2\varepsilon_0 + \log_{\frac{23}{18}} \frac{1}{6\varepsilon_0}$ and $b = 1 + \log_\gamma 2$. Then for every $d \geq a + b \log_2 k$ the error probability is small, $p_d < (\frac{1}{2})^{2^k}$, and so the probability that a formula of height d computes the majority function is $1 - p_d > 1 - (\frac{1}{2})^{2^k}$ as claimed.

Corollary 1. *For every n, there is a monotone formula of height $d = \lceil a + b \log_2 n \rceil$ which computes the majority function, where a and b are as above.*

Proof. A randomly chosen formula F of height d will compute the majority function on a fixed input v except with probability strictly less than 2^{-n}. But there are only 2^n possible values of v, so the wrong formulas are sparse, relative to the inputs. Formally speaking,

$$Pr(F = MAJ_n) \geq 1 - \sum_v Pr(F(v) \neq MAJ_n(v)) = 1 - 2^n p_d > 1 - 2^n 2^{-n} = 0.$$

Since the probability of a random F computing the majority function is positive, there exists such an F.

Corollary 2. *There is a probabilistic polynomial-time algorithm for emitting a formula of depth* $d = \lceil a + b \log_2 2n \rceil$ *which computes the majority function on* n *bits except with negligible probability.*

Proof. The algorithm simply computes d and outputs a random formula F of height d. This computes the majority function except with probability 2^{-2n}. Then

$$Pr(F = MAJ_n) \geq 1 - \sum_v Pr(F(v) \neq MAJ_n(v)) = 1 - 2^n p_d > 1 - 2^n 2^{-2n} = 1 - 2^{-n},$$

and so the algorithm behaves as claimed.

For example, taking $Th_2^3(b_1,\ b_2,\ b_3)$ to be the majority of three bits, one can straightforwardly (somewhat laboriously) verify that the following formula computes the majority function on five bits. We claim without proof that no formula of smaller height computes the same function.

$$Th_2^3(x_1,\ Th_2^3(x_2,\ x_3,\ x_4),\ Th_2^3(x_1,\ Th_2^3(x_2,\ x_3,\ x_5),\ Th_2^3(x_2,\ x_4,\ x_5)))$$

3 A Linear Secret-Sharing Scheme based on $T(k, m)$ Formulae

A secret sharing scheme $\mathcal{S}$ is a probabilistic algorithm that takes as input a secret s (a bit string) and a security parameter k, and outputs a set of *shares* $S_1, \ldots, S_n$. We require *correctness*, i.e., that from certain subsets of the shares s is uniquely determined. Such sets are called *qualified*, and the family of qualified sets is called the *access structure*. We will consider threshold-t access structures that contain all sets of size larger than t. Finally, we require *privacy*: for any non-qualified set A of players, let $D_A(s)$ be the joint distribution of shares given to A when s is the secret. Then privacy simply means that $D_A(s)$ is that same for any s.

A linear secret sharing scheme (LSS) is a scheme where the secret s is chosen from a finite ring R, and where each share S_i is a tuple of j_i elements in R, $S_i = (s_{i,1}, \ldots, s_{i,t_i})$, the $s_{i,j}$'s are called *share components*. The shares are computed from the secret s and some some random elements chosen from R. Furthermore, it must be the case that reconstruction of s can be done by taking an integer linear combination of the share components. We say the scheme is *multiplicative* if the following properties holds.

Definition 1. *A LSS* $\mathcal{S}$ *is* multiplicative *if the following holds: suppose secrets* a *and* b *have been shared using* $\mathcal{S}$*. Consider the shares of* a *and* b*,* $A_i = (a_{i,1}, ..., a_{i,t_i}), B_i = (b_{i,1}, ..., b_{i,t_i})$ *held by the* i*'th player and let* $c_{i,j} = a_{i,j} b_{i,j}$*. The for a fixed set of integer coefficients* $\alpha_{i,j}$ *we have:*

$$ab = \sum_{i=1}^{n} \sum_{j=1}^{t_i} \alpha_{i,j} c_{i,j} \tag{1}$$

Let C *be the set of indices of players in the complement of any unqualified set. The scheme is said to be* strongly multiplicative *if for any such* C*, there exists coefficients* $\beta_{i,j}$ *such that*

$$ab = \sum_{i \in C} \sum_{j=1}^{t_i} \beta_{i,j} c_{i,j}$$

Put differently, once a and b have been shared, the shares can be locally converted to shares of ab in a new LSS $\mathcal{S}'$, and the set of all players is qualified in $\mathcal{S}'$. If the scheme is strongly multiplicative, the set of honest players is qualified in $\mathcal{S}'$.

A simple example of a multiplicative LSS is the following scheme $\mathcal{S}_{2/3}$, producing 3 shares where any set of two shares is qualified: choose s_1, s_2 uniformly at random from R, and set $s_3 = s - s_1 - s_2$. Then we set

$$S_1 = (s_2, s_3), S_2 = (s_1, s_3), S_3 = (s_1, s_2).$$

This scheme is clearly a multiplicative LSS. One way to argue privacy is by considering the distribution seen by any non-qualified set, when the secret is 0, and show that this is the same as the one you get when the secret is s. This is trivial for S_3, and for the other shares, for instance S_1, it follows from the fact that we can change the secret from 0 to s leaving S_1 the same by replacing s_1 by $s_1 + s$. This new choice has the same probability as the original one.

Given a Boolean formula F consisting of 2-out-of-3 threshold gates, we can construct a LSS $F(\mathcal{S}_{2/3})$, as follows: we identify the input bits $(b_1, \ldots, b_n)$ of F with the shares that are to be produced, and the share corresponding to an input bit b_i contains one share component for each position where F refers to b. The construction then works as follows:

Secret Sharing Scheme $F(\mathcal{S}_{2/3})$.

- If F consists of a single gate we execute $\mathcal{S}_{2/3}$ on input secret s.
- Otherwise, we have that

  $$F(b_1, \ldots, b_n) = T_{2/3}(F_1(b_1, \ldots, b_n), F_2(b_1, \ldots, b_n), F_3(b_1, \ldots, b_n)),$$

  for formulas F_1, F_2, F_3 and where $T_{2/3}$ is the 2-out-of-3 Boolean threshold gate. Now first execute $\mathcal{S}_{2/3}$ on input secret s, to get share components s_1, s_2, s_3. Then execute $F_1(\mathcal{S}_{2/3})$ twice, on input secret s_2 and s_3. Similarly, execute $F_2(\mathcal{S}_{2/3})$ on s_1, s_3 and $F_3(\mathcal{S}_{2/3})$ on s_1, s_2.

Note also that we only define this secret sharing scheme for formulas that are *full*; that is, we require that the number of gates in a formula of height h is $\sum_{i=0}^{h-1} 3^h$. This can of course be generalized to non-full trees (formulas), but this more restricted definition is sufficient for the results in this paper, and it makes the proofs simpler.

Theorem 1. *$F(\mathcal{S}_{2/3})$ is multiplicative, with all $\alpha_{i,j} = 1$.*

Proof. Let a and b be given. We want to argue by induction on the height of the formula tree that the secret sharing scheme is multiplicative, with all $\alpha_{i,j} = 1$.

So let's first consider what happens if F is a single gate. Then $a = a_1 + a_2 + a_3$ and $b = b_1 + b_2 + b_3$; the first player receives share components (a_2, a_3, b_2, b_3), player 2 receives (a_1, a_3, b_1, b_3) and player 3 receives (a_1, a_2, b_1, b_2). Note that $ab = \sum_{i=1}^{3} \sum_{j=1}^{3} a_i b_j$; we let $(c_{1,1}, c_{1,2}, c_{1,3}) = (a_2 b_2, a_2 b_3, a_3 b_2)$. Similarly, $(c_{2,1}, c_{2,2}, c_{2,3}) = (a_3 b_3, a_1 b_3, a_3 b_1)$ and $(c_{3,1}, c_{3,2}, c_{3,3}) = (a_1 b_1, a_1 b_2, a_2 b_1)$. Then clearly each player i can compute $c_{i,1\ldots3}$, and they sum up as desired: $ab = \sum_{i,j}(1 \cdot c_{i,j})$.

Next, consider what happens if F consists of multiple gates. We still have $a = a_1 + a_2 + a_3$ and $b = b_1 + b_2 + b_3$, with the same share tuples being given to each player, but now the share components are themselves shared recursively.

Consider the $c_{i,j}$ defined above, e.g. $c_{1,1\ldots3} = (a_2 b_2, a_2 b_3, a_3 b_2)$. Since F_1 is used to reshare a_2 and a_3, and $F_1(\mathcal{S}_{2/3})$ is (by induction hypothesis) a multiplicative LISS with all $\alpha_{i,j} = 1$, from the

shares given to player i one can compute $c'_{i,(2,2),1}, \ldots, c'_{i,(2,2),t_{i,(2,2)}}$ such that $a_2 b_2 = \sum_{i,j} c'_{i,(2,2),j}$. Similarly, one can write $a_2 b_3 = \sum_{i,j} c'_{i,(2,3),j}$ and vice versa for $a_3 b_2$, and for the remaining players and the share components of both a and b.

But then we can let $t_i = \sum_{k,m} t_{i,(k,m)}$, summing over all (k,m) where player i reshares $a_{k,m}$ and $b_{k,m}$. We also let $c''_{1,1\ldots t_1}$ be the concatenation of all the $c_{i,(\cdot,\cdot),j}$-values summing up to $a_2 b_2$, $a_2 b_3$ and $a_3 b_2$, respectively. Similarly for $c''_{2,1\ldots t_2}$ and $c_{3,1\ldots t_3}$.

Then for all i we have $\sum_j c_{i,j} = \sum_j c''_{i,j}$, and thus $ab = \sum_{i=1}^{n} \sum_{j=1}^{t_i} (1 \cdot c''_{i,j})$, which was exactly what we wanted to show.

Theorem 2. *$F(\mathcal{S}_{2/3})$ is a correct and private LSS.*

Proof. If F is just a single gate, then $F(\mathcal{S}_{2/3})$ is equivalent to $\mathcal{S}_{2/3}$.

Otherwise, $F(\mathcal{S}_{2/3}) = T_{2/3}(F_1, F_2, F_3)$ for some formulae F_1, F_2 and F_3. We want to argue by induction in the height of the formula tree that the scheme is correct.

Given a secret $s = s_1 + s_2 + s_3$, when we execute $F_1(\mathcal{S}_{2/3})$ on s_2 and s_3 we get tuples $S_{1,2} = (s_{1,2,1}, \ldots, s_{1,2,n_{1,2}})$ and $S_{1,3}$ of share components; similarly, F_2 and F_3 yield $S_{i,j}$ for $i = 2, 3$ and $j = 1, 2, 3$, $j \neq i$. Clearly the concatenation of these tuples is itself a tuple.

We know that s is a linear combination of the share components, $s = \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3$, since $F(\mathcal{S}_{2/3})$ is a LISS—in fact, we know that $\lambda_i = 1$ for all i. Similarly, s_2 is shared by a LISS, $F_1(\mathcal{S}_{2/3})$, so $s_2 = \lambda_{1,2,1} s_{1,2,1} + \ldots + \lambda_{1,2,n_{1,2}} s_{1,2,n_{1,2}}$, where once again all the λ's are $= 1$ (recall that $F_1(\mathcal{S})$ is a LISS). Similarly for s_1 and s_3, so

$$
\begin{aligned}
s &= \lambda_1(\lambda_{2,1,1} s_{2,1,1} + \ldots + \lambda_{2,1,n_{2,1}} s_{2,1,n_{2,1}}) \\
&\quad + \lambda_2(\lambda_{1,2,1} s_{1,2,1} + \ldots + \lambda_{1,2,n_{1,2}} s_{1,2,n_{1,2}}) \\
&\quad + \lambda_3(\lambda_{1,3,1} s_{1,3,1} + \ldots + \lambda_{1,3,n_{1,3}} s_{1,3,n_{1,3}}) \\
&= (\lambda_1 \lambda_{2,1,1}) s_{2,1,1} + \ldots + (\lambda_1 \lambda_{2,1,n_{2,1}}) s_{2,1,n_{2,1}} \\
&\quad + (\lambda_2 \lambda_{1,2,1}) s_{1,2,1} + \ldots + (\lambda_2 \lambda_{1,2,n_{1,2}}) s_{1,2,n_{1,2}} \\
&\quad + (\lambda_3 \lambda_{1,3,1}) s_{1,3,1} + \ldots + (\lambda_3 \lambda_{1,3,n_{1,3}}) s_{1,3,n_{1,3}} \\
&= s_{2,1,1} + \ldots + s_{2,1,n_{2,1}} \\
&\quad + s_{1,2,1} + \ldots + s_{1,2,n_{1,2}} \\
&\quad + s_{1,3,1} + \ldots + s_{1,3,n_{1,3}}
\end{aligned}
$$

and thus s is not just a linear combination of share components, but a sum—i.e. all λ's are $= 1$. (Note that we reconstruct s by taking s_1 from F_2 and s_2 and s_3 from F_1. This choice is arbitrary, and we could reassemble s in eight distinct ways.) Finally, for privacy we use the same as before to rewrite $F(\mathcal{S}_{2/3}) = T_{2/3}(F_1, F_2, F_3)$. Say F has height h, then the F_i have height $h - 1$. From each of the F_i, we can get 3 formulae of height $h - 2$, etc. down to height 1. Let G be any of these formulae, of height h'. We claim it holds that if numbers $s, s + \delta$ has been shared using $G(\mathcal{S}_{2/3})$, where $\delta \in R$, then any unqualified set will see the same distributions in the two cases. applying this result with $G = F$ clearly implies privacy. We show the claim by induction on h', where $h' = 1$ is clear from the argument we gave above for $\mathcal{S}_{2/3}$. For the induction step, recall that we are executing $G(\mathcal{S}_{2/3}) = T_{2/3}(G_1, G_2, G_3)$ for formulae G_i of height $h' - 1$, and this works by choosing s_1, s_2 at random and setting $s_3 = s - s_1 - s_2$. Then we share s_2, s_3 using G_1, s_1, s_3 using

G_2 and s_1, s_2 using G_3. If set A is unqualified w.r.t. G, it can be qualified w.r.t to only one of G_1, G_2, G_3. If A is qualified w.r.t. G_3, note that sharing $s + \delta$ instead of s using the same choices of s_1, s_2 will lead to the same shares being produced by G_3 while the secrets being shared by G_1 and G_2 change, but A will still see the same distribution by induction hypothesis. If A is qualified w.r.t. G_2, the secret can changed to $s + \delta$, while keeping s_1, s_3 constant by changing s_2 to $s_2 + \delta$. Since A is unqualified w.r.t. G_1 and G_3, this change will again lead to the same distribution of shares by induction hypothesis. The case where A is qualified w.r.t. G_1 is similar.

This construction is a variant of the Benaloh-Leichter construction from [2], the difference is they would have used additive secret sharing to handle each gate in the formula. This would have given us shorter shares, but the scheme would not be multiplicative.

This construction generalizes trivially to any formula containing threshold gates and no constants, and the qualified sets will be those corresponding to inputs that the formula accepts.

Note that reconstruction in the schemes we build indeed takes place by simply adding shares, as promised in the introduction.

4 A strongly multiplicative secret sharing scheme

We have looked at a secret sharing scheme based on 2-out-of-3 threshold gates. We can think of those as the smallest Q_2 gate: where the string of all ones can't be written as the bitwise OR of two inputs which both produce 0 as the output. Equivalently, the access structure of the corresponding secret sharing scheme is Q_2.

We now want to study what happens when the primitive gates are Q_3: when not even the OR of three 0-yielding inputs is sufficient to cover all inputs. In particular, we look at $(t+1)$-out-of-$(3t+1)$ threshold gates, i.e. 2-out-of-4, 3-out-of-7 and so forth.

Our first theorem will show that these compose well:

Theorem 3. *Suppose we have a formula F of gates, each of which are Q_3. Then the formula itself (over, say, k inputs) is also Q_3.*

Proof. We will show by induction on the height of the formula that if a triple of inputs which violate the Q_3 condition exist for any formula of that height, then we arrive at a contradiction.

Suppose the formula has height 0, i.e. it takes a single input and outputs it. Then any three inputs which all yield zero as the output must all be zero. But then their OR is 0, and if their OR was also 1 (i.e. the inputs witness the Q_3 violation) we would clearly have a contradiction.

Suppose next that the formula has height $n + 1$ for some $n \geq 0$, and that (x_1, x_2, x_3) is given with $F(x_1) = F(x_2) = F(x_3) = 0$ and $x_1 \vee x_2 \vee x_3 = 1^k$, i.e. the x_i's violate the Q_3 property of F.

By assumption all gates in F, including the root gate G, are Q_3. Consider its input gates G_j for $j = 1, \ldots, m$. Let y_1 be a string of bits of length m, where $(y_i)_j = G_j(x_i)$. That is, the j'th bit of y_i is 1 if and only if the input bits of x_i cause the j'th gate to output 1.

Clearly $G(y_i)$ must be 0 for all i since $F(x_i) = 0$. Since the top gate is Q_3, the OR of all three y_i's can't be 1^m. Let's say the j'th bit of the OR is zero. That means that $G_j(x_1) = G_j(x_2) = G_j(x_3) = 0$, but since the OR of (x_1, x_2, x_3) is all ones, the OR is one at all the bit positions which (eventually) feed into G_j. That is, they witness that G_j is not Q_3. But by induction that is a contradiction. Thus, all formulae of Q_3 gates are themselves Q_3.

Not only is the Q_3 property preserved across composition, so is strong multiplicativity. Recall from definition 1 that strong multiplicativity means that the complement of an unqualified set of players can compute the product of two secrets $r \cdot s$ as a linear combination of locally computable products of share components.

Theorem 4. *Suppose we have a formula F of gates, each of whose secret sharing schemes are strongly multiplicative. Then the secret sharing scheme induced by F is strongly multiplicative.*

Proof. We will prove this by induction in the height of F. If F has height one, it's a single gate which is strongly multiplicative by assumption. If F has height $k+1$, let R be the complement of a set of corruptible players. Given the set of shares of s and r held by the players in R we can reconstruct s and r.

This means we can reconstruct a set of inputs to the root gate the complement of which is corruptible. Let us say the root gate distributes shares $r_1 \cdots r_d$ and $s_1 \cdots s_d$ to those recipients (players or subformulae). Then, since the root gate is strongly multiplicative, $rs = \sum_{i,j} \lambda_{ij} r_i s_j$ for (only) those pairs of i and j where r_i and r_j are sent to the same recipient (player or other gate), for some constants λ_{ij}.

Since each subformula has height at most k we know by induction that they're strongly multiplicative, so $r_i s_j = \sum_{a,b} \kappa_{ab} (r_i)_a (s_j)_b$, that is, we can write $r_i s_j$ as a linear combination of the shares of r_i and s_j, respectively, that are sent to the same subformula. Since we only look at inputs to the root gate we can reconstruct, the sets of corrupted inputs to the subformulae are each contained in the adversary structures of each of those formulae.[2]

Thus, we can write rs as a linear combination of linear combinations of shares held by the players in R. But that's a linear combination, $\sum_{i,j,a,b} (\lambda_{ij}\kappa_{ab})(r_i)_a (s_j)_b$, of shares from R, which is what we wanted to show the existence of.

Next, we want to show that similar to the 2-out-of-3 case, when we compose enough 2-out-of-4 gates, we approach computing a certain threshold function.

Lemma 2. *Let $t_0 = \frac{1}{6}(5 - \sqrt{13})$ and let T be the threshold function $Th^n_{\lceil nt_0 \rceil}$. For some integer ℓ which is $O(\log n)$ and every $d \geq \ell + k$, consider full formulae of 2-of-4 gates of height d. For any bit string v, a randomly chosen such formula computes the threshold function T on v with probability exceeding $1 - (\frac{1}{2})^{2^k}$.*

Proof. Let p_d be the probability that such a formula F of depth d doesn't compute $T(v)$. We can express p_{d+1} recursively when $d \geq 1$. If $T(v) = 0$ then p_{d+1} is the probability that $F(v) = 1$, which happens when at least two inputs to the root gate are $1 \neq T(v)$, i.e. when at least two inputs are themselves wrong, which happens with probability

$$p(p_d) = \binom{2}{4} p_d^2 (1-p_d)^2 + \binom{3}{4} p_d^3 (1-p_d)^1 + \binom{4}{4} p_d^4 (1-p_d)^0 = p_d^2 \cdot (3p_d^2 - 8p_d + 6).$$

This polynomial p in p_d has four fixed points: $\{0, t_0, 1, \frac{1}{6}(5+\sqrt{13})\}$. Note that $\frac{1}{6}(5+\sqrt{13}) \approx 1.434 > 1$ and $t_0 \approx 0.2324$.

Note that since t_0 is irrational, $\lfloor nt_0 \rfloor < nt_0 < \lceil nt_0 \rceil = \lfloor nt_0 \rfloor + 1$. Let $t = \lfloor nt_0 \rfloor$, and recall that the threshold is $t+1$. In a formula of depth 0, the probability p_0 of falsely outputting 1 is the

[2] That is, the adversary is allowed to corrupt the corrupted sets. In other words the corrupted sets are corruptible.

number of ones divided by the number of inputs, at most $\frac{t}{n}$. We want to bound this away from t_0 by the inverse of a polynomial in n. Then we want to show that by increasing the depth of the formula, thus repeatedly applying p to the error probability, we can reduce the error probability first linearly, then exponentially.

To bound $\frac{t}{n}$ away from t_0, we use continued fractions. Note that we have $t_0 = [0; 4, \overline{3}]$. That is, t_0 has quotients $a_0 = 0$, $a_1 = 4$ and $a_i = 3$ for $i \geq 2$. We define two recursive series:

$$h_{-2} = 0 \qquad h_{-1} = 1 \qquad h_n = a_n h_{n-1} + h_{n-2} \tag{2}$$

$$k_{-2} = 1 \qquad k_{-1} = 0 \qquad k_n = a_n k_{n-1} + k_{n-2} \tag{3}$$

For instance, $h_0 = 0$, $h_1 = 3$ and $h_2 = 10$ while $k_0 = 1$, $k_1 = 4$ and $k_2 = 13$. Some well-known facts about continued fractions are that the series $\frac{h_i}{k_i}$ approaches t_0, that each such *convergent* is a better approximation of t_0 than any fraction with a smaller denominator, that the convergents alternate between being greater and less than t_0, and that

$$\frac{1}{k_i(k_i + k_{i+1})} < \left| t_0 - \frac{h_i}{k_i} \right|.$$

So consider $\frac{t}{n}$. Note that $0 < k_i < k_{i+1} \leq 4k_i$ for $i \geq 0$: we directly observe it to hold for $i = 0$ and $i = 1$, and as $a_i = 3$ for $i \geq 2$ it holds by the recursive definition for those i as well. Let j be minimal such that $k_j \geq n$, and note that $k_j \leq 4n$. Let i be j or $j + 1$, such that $\frac{h_i}{k_i} < t_0$, and note that $k_i \leq 16n$. Also, $k_i(k_i + k_{i+1}) \leq k_i^2 + k_i(4k_i) = 5k_i^2 \leq 5(16n)^2 = 1280n^2$, and so

$$t_0 - \frac{t}{n} \geq t_0 - \frac{h_i}{k_i} = \left| t_0 - \frac{h_i}{k_i} \right| > \frac{1}{k_i(k_i + k_{i+1})} \geq \frac{1}{1280n^2}.$$

Next, we want to find a constant c such that $p(x) < \frac{1}{2}x$ for $x \leq c$. Note that $t_0 < \frac{1}{4}$, so if $0 < x \leq t_0$ then $x < \frac{1}{4}$. In that case, $p(x) = 3x^4 - 8x^3 + 6x^2 < 6x^2 + 3x^2 \cdot (\frac{1}{4})^2 = \frac{99}{16}x^2$. Let $c = \frac{1}{2} \cdot \frac{16}{99} < t_0$. If $0 < x \leq c$ then $p(x) < \frac{99}{16}x^2 \leq c\frac{99}{16}x = \frac{1}{2}\frac{16}{99}\frac{99}{16}x = \frac{1}{2}x$.

Consider the secant line through $(c, p(c))$ and $(t_0, p(t_0)) = (t_0, t_0)$. This has slope $m := \frac{t_0 - p(c)}{t_0 - c}$, so let $f(x) = t_0 + m(x - t_0) = t_0 - m(t_0 - x)$, such that the graph of f is the secant line. Note that $p(c) = (3 \cdot 8^4 - 8 \cdot 8^3 \cdot 99 + 6 \cdot 8^2 \cdot 99^2)/99^4 < c$ so $m > 1$. Also, $f(x) \leq p(x)$ when $c \leq x \leq t_0$; $f(x) < c$ when $x < c$; and $f(t_0 - \varepsilon) = t_0 - m(t_0 - (t_0 - \varepsilon)) = t_0 - m\varepsilon$ when $c \leq t_0 - \varepsilon \leq t_0$.

Look at the series $\frac{t}{n}, p(\frac{t}{n}), p(p(\frac{t}{n})), \ldots$ with $x_i = p^{(i)}(\frac{t}{n})$. Recall that $t_0 - x_0 > \frac{1}{1280n^2}$ and note that $c \leq x_i \leq t_0 \Rightarrow t_0 - x_i = t_0 - p^{(i)}(\frac{t}{n}) \geq t_0 - f^{(i)}(\frac{t}{n}) = m^i(t_0 - \frac{t}{n}) > \frac{m^i}{1280n^2}$. This means that if $m^i \geq 1280n^2 \cdot (t_0 - c)$ then $t_0 - x_i > t_0 - c$, that is, $x_i < c$.[3] This happens when $i \geq \log_m(1280n^2 \cdot (t_0 - c)) = 2\log_m n + \log_m 1280(t_0 - c)$. Let j be minimal subject to $x_j < c$, that is, $j = \lceil 2\log_m n + \log_m 1280(t_0 - c) \rceil$. Then $p(x_{j+k+1}) < \frac{1}{2}x_{j+k}$ for $k \geq 0$ so $p_{j+k} = p^{(j+k)}(p_0) < 2^{-k}c$.

In the case of the majority function, negating the inputs negates the outputs (on an odd number of bits), which means that the bound on the probability of false positives also bounds the probability of false negatives. This is not the case for non-majority threshold functions, in particular not the 2-out-of-4 function, but we can reuse the overall ideas in the proof.

[3] At the smallest such i we go from $c \leq t_0 - \varepsilon \leq t_0$ via $f(t_0 - \varepsilon) = t_0 - m\varepsilon$ to $\neg(c \leq t_0 - \varepsilon \leq t_0)$, i.e. $t_0 - \varepsilon < c$, and at any greater i we exploit the fact that $p(x) \leq x$ when $0 \leq x \leq t_0$.

So, if $T(v) = 1$ then p_{d+1} is the probability that $F(v) = 0$, which happens when at most one root gate input is 1, or equivalently at least three inputs are $0 \neq T(v)$, i.e. wrong, which happens with probability

$$q(p_d) = \binom{3}{4} p_d^3 (1-p_d)^1 + \binom{4}{4} p_d^4 (1-p_d)^0 = p_d^3(-3p_d + 4)$$

This polynomial in p_d has four fixed points, $\{0, 1, t_1, \frac{1}{6}(1-\sqrt{13})\}$, where $t_1 = \frac{1}{6}(1+\sqrt{13})$. Note that $t_0 + t_1 = 1$, that $t_1 \approx 0.7676$ and that $\frac{1}{6}(1-\sqrt{13}) < 0$. Note that $p(x) = 1 - q(1-x)$.

Once again, we first want to bound the error probability of a depth 0 formula, then amplify this bound by increasing the depth, which corresponds to repeatedly applying q to the error probability.

If F has depth 0, the probability of falsely outputting zero on v is the number of zeroes divided by the number of bits. Recall that the threshold is $\lceil nt_0 \rceil = t + 1$, thus v has at least this many ones and at most $n - \lceil nt_0 \rceil$ zeroes. Note that since $t_0 + t_1 = 1$, this equals $n - \lceil n(1-t_1) \rceil = n - (n - \lfloor nt_1 \rfloor) = \lfloor nt_1 \rfloor$.

Similar to earlier, we want to bound $\frac{\lfloor nt_1 \rfloor}{n}$ away from t_1. As a continued fraction, $t_1 = [0; 1, \overline{3}]$, having quotients $a_0 = 0$, $a_1 = 1$ and $a_i = 3$ for $i \geq 2$. We define h_i and k_i recursively like before, but of course with respect to this new series of quotients. Once again we see that $0 \leq k_i \leq k_{i+1} \leq 4k_i$ and in fact $k_i < k_{i+1}$ for $i \geq 1$. By the same argument as above, we get that $nt_1 - \lfloor nt_1 \rfloor > \frac{1}{1280n^2}$.

Note that $q(x) = x^3(-3x+4) < 4x^3$ whenever $x > 0$. Let $b = \sqrt{\frac{1}{8}}$. If $0 < x \leq b$ then $q(x) < 4 \cdot x^2 \cdot x = \frac{4}{8}x = \frac{1}{2}x$.

Consider the secant line through $(b, q(b))$ and $(t_1, q(t_1))$. This has slope $s = \frac{q(t_1)-q(B)}{t_1 - B}$ and goes through the point (t_1, t_1); recall that t_1 is a fixed point of q. Let $g(x) = t_1 + s(x - t_1) = t_1 - s(t_1 - x)$ and note that the graph of g is this secant line, and that $g(t_1 - \varepsilon) = t_1 - s(t_1 - (t_1 - \varepsilon)) = t_1 - s\varepsilon$.

Note that $q(\frac{1}{2}) = \frac{5}{16} < \frac{1}{2}$, and that $q(x) - x$ doesn't change sign between two roots (two fixed points of q), thus $q(x) < x$ when $0 < x < t_1$. In particular, $q(b) < b$ and thus $s > 1$. Also, $q(x) \leq g(x)$ whenever $b \leq x \leq t_1$ and $q(x) < b$ whenever $0 \leq x \leq b$.

Let $r = \lfloor nt_1 \rfloor$ and consider, similar to before, the series $r, g(r), g(g(r)), \ldots, g^{(i)}(r)$. Note that $r < nt_1 - 1/1280n^2$ and thus $g^{(i)}(r) < s^i/1280n^2$. Let j' be the smallest value such that $g^{(j)}(r) \leq b$, that is, $j' = \lceil 2\log_s n + \log_s 1280(t_1 - b) \rceil$. For any $k \geq 0$ we have $p_{j'+k} \leq 2^{-k}b$.

It's obvious that $F(v)$ can be unequal to $T(v)$ iff F yields either a false negative or a false positive. The error probability is thus the sum of the probability of those two events. When the depth d of F exceeds both j and j', both error probabilities drop off by a factor two for each increase of d by 1; thus the sum error probability does the same. Note that j and j' are both $O(\log n)$, and that in a constant number of depth increments we can diminish the error probability sufficiently to drown out any constants. Thus, there exists a number ℓ which is $O(\log n)$ such that the error probability of a formula of depth $d \geq \ell$ is at most $(\frac{1}{2})^{d-\ell}$, which is what we wanted to show.

5 Black-Box Computing over Finite Groups

5.1 The problem and a Solution for Three Players

The problem we want to solve is as follows: n players $P_1, ..., P_n$ hold as private input elements $h_1, ..., h_m$ in a finite group G, more precisely, each h_j is held by exactly one player P_i. The goal

is to compute the product $h = h_1 h_2 \cdots h_m$ securely, even if $t < n/2$ players are corrupted by an adversary.

We will first consider semi-honest security where even corrupted players are assumed to follow the protocol. Therefore, a very simple definition of security suffices: we require *correctness*: all honest players output the correct value of h, and *privacy*: there exists an efficient simulator which, when given the input elements of players in corrupted set C as well as the output h, outputs a view that has the same distribution as the joint view of players in C when the actual protocol is executed.

The protocol and simulator must work by only black-box access to G, i.e., the only available operations are the group operation, computing the inverse and random sampling from G.

A main ingredient in the solution is a protocol from [5] for three players which takes two group elements, each secret-shared in a suitable form, and outputs a secret-sharing of their product, which we describe shortly. Our idea is to use this protocol in the player emulation approach of Hirt and Maurer [6]. To do this, we will need some additional 3-player protocols which we return to below. From this, and our formulas for computing the majority function, we can build an n-player protocol that securely computes the product of two secret-shared group elements and returns the product in shared form.

Before describing our n-player protocol, we need to take a closer look at the 3-player protocol from [5] for multiplying two shared group elements: This protocol makes a distinction between left and right hand inputs and output, springing from the fact that the operation of the group might do so. To accommodate this, we need to share secrets in either left or right sharings.

To create a left sharing of x, choose x_1 uniformly in G and let $x_2 := x \cdot x_1^{-1}$, such that $x_1 \cdot x_2 = x$; then send x_1 to player 2 and x_2 to player 1. To make a right sharing of x, generate x_1 and x_2 the same way, but send x_i to player i. Note that in neither case does player 3 have any share.

Also, "protocol", in the singular, is a simplification: we actually need two protocols. They will take two sharings as their input—a left and a right—and output one sharing, either a left or a right.

We now look at the structure of these protocols. See figure 1 for the protocol which yields a left output. We'll explain in detail how it works and leave it to the reader to apply the same principles to the protocol yielding a right output. This is essentially an informal restatement of algorithm 2 from [5].

Each node is labeled with a player ID, the node's *owner*. At each node, that node's owner receives a sequence of inputs, $g_1, \ldots, g_m$. That player computes $g = g_1 \cdot \cdots \cdot g_m$ and sets $g'_1 \cdot \cdots \cdot g'_k = g$ where k is the node's outdegree and $g'_1, \cdots, g'_{k-1}$ are sampled uniformly. Then, the owner sends g'_i to the owner of the i'th neighbour node, going from left ($i = 1$) to right ($i = k$). The inputs to the topmost nodes are the inputs to the protocol. All other inputs to nodes are group elements sent in this way. The outputs at the bottom row are the protocol outputs of the indicated players.

Thus, if a player owns both a node of outdegree one and its neighbour, that player multiplies the received values, then sends the product to himself. Sending to oneself can of course be optimized away, but the seemingly redundant nodes make the graph satisfy the formal structural precondition of the algorithm 2 referred to earlier.

For instance, at the topmost node of the central column of figure 1, player 1 receives x_2 and y_1, computes $g := x_2 y_1$, samples (g_1, g_2), sets $g_3 = g_2^{-1} g_1^{-1} g$, then sends g_1 to player 2, g_2 to himself and g_3 to player 2. Observe that if $x = x_1 x_2$ and $y = y_1 y_2$ then $xy = x_1 \cdot (x_2 y_1) \cdot y_2 = (x_1 g_1) \cdot (g_2) \cdot (g_3 y_2)$, that is, the left-to-right product of all numbers sent from one layer to the next is always $x \cdot y$.

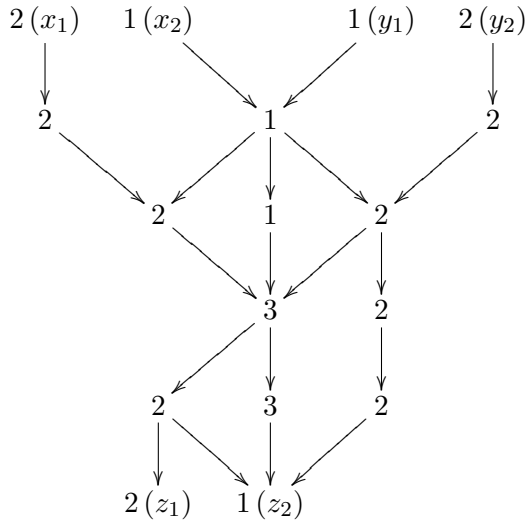

Figure 1: Multiplication protocol (left output)

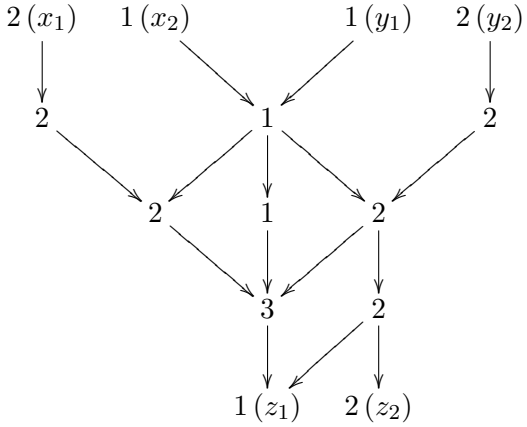

Figure 2: Multiplication protocol (right output)

Correctness, that is, equality at the top and bottom layers, follows as a direct corollary of this invariant. It also relates to security, in the following way: let c be any corrupt player. On every layer of the graph there's a node not owned by c, i.e. there's a uniformly random factor not known by c, so what c knows appears uniformly random. Moreover, one can draw a path in the reflexive closure of the graph that is the tree of two-element multiplication protocols from any top-most input to some bottom-most output, stepping only on nodes hiding these uniformly random factors from c. This means not only are the random factors hidden on their own layers, the adversary can't pick them up indirectly. This is the intuition; a formal proof that this translates into a simulator is to be found in [5], see in particular definition 2 and 3 and lemma 1 and 2. The lemmas assume the two-element multiplication protocols to have certain properties, which we'll establish here.

Let L and R be the protocol graphs, and let $L^{\leftrightarrow}$ and $R^{\leftrightarrow}$ be their undirected counterparts, i.e. their reflexive closures. Observe that that for any one corrupted player c there exists two indices $i_x, i_y \in \{1, 2\}$ such that there is a path from x_{i_x} to z_{i_x} and from y_{i_y} to z_{i_x} in $L^{\leftrightarrow}$, and from x_{i_x} to z_{i_y} and from y_{i_y} to z_{i_y} in $R^{\leftrightarrow}$, such that none of these four paths contain any nodes owned by c.

The existence of a path from x_i to z_i (rather than z_j, $j \neq i$) mean that L is *x-preserving*. Similarly, the existence of a path from y_i to z_i in R (for any c) means that R is *y-preserving*. The fact that for any c, L and R use the same i_x (though in different ways) and the same i_y (ditto) is what [5] refers to as *compatibility*. These three properties are the ones from which the existence of a simulator ultimately follows.

5.2 Construction of a Protocol for n Players

Following [6], we will build an n-party protocol for computing the product $h_1 \cdots h_m$, where each h_i is an input from one of the n players. We build the protocol based on a formula F of $T(2, 3)$-gates, computing the majority function on n input bits. We know from the previous sections that such a formula of logarithmic depth exists.

Before we describe the n player protocol, we specify a few simple protocols for groups of 3

players. The protocols allow 3 players p_1, p_2, p_3 to receive input in secret shared form, do operations on shared values, send shared values securely to another group of 3 players, receive values from another group, and finally to make a shared value public. We maintain as invariant that every shared value held by the three players is stored in the form of a left sharing.

Input: To give an input $x \in G$ to (p_1, p_2, p_3), one creates a random left sharing x_1, x_2 of x and sends x_1 to p_2 and x_2 to p_1.

Multiplication: the multiplication protocol takes two left sharings (x_1, x_2) and (y_1, y_2) as input. It runs the right-output multiplication protocol[4] on (y_1, y_2) and a default right sharing of e (e.g. $e_1 = e_2 = e$), yielding a right sharing (y'_1, y'_2) of $y_1 \cdot y_2$. It then runs the left-output multiplication protocol on (x_1, x_2) and (y'_1, y'_2), and gives the output of this as its own output.

Inversion: the inversion protocol takes a single left sharing (x_1, x_2) of x as its input; then each player inverts their own element, so player 2 holds x_1^{-1} and player 1 holds x_2^{-1}. Since $x^{-1} = (x_1 x_2)^{-1} = x_2^{-1} x_1^{-1}$, what the players hold is a right sharing of the inverse: player 1 holds the left part and player 2 the right part. So, the inversion protocol multiplies this right sharing with a default left sharing of e (e.g. $e_1 = e_2 = e$), and gives the output of the left-output multiplication protocol as its own output.

Sample: to sample a uniformly random group element, players 1 and 2 each sample a group element (x_2 and x_1, respectively) uniformly at random, and output the left sharing (x_1, x_2).

Send/Receive: If virtual player p is to send a value x to virtual Suppose p_1, p_2, p_3 want to send a shared value x to q_1, q_2, q_3. Then p_1 holds x_2 and p_2 holds x_1 such that $x_1 \cdot x_2 = x$. To emulate the send operation, p_1 creates a left-sharing of x_2 and p_2 of x_1; they each send the shares to q_1 and q_2 as appropriate; then q_1, q_2, q_3 execute the Multiplication protocol to get a left-sharing of x.

Output: To make a shared value public, p_1, p_2 broadcast their shares, and all players can locally multiply shares to get the result.

To keep track in the following of the values the players hold, each such value is assigned a variable name. The value stored in variable X is usually called x – to keep the description simple we will usually not change the value in a variable once it is created, instead we create a new variable. A variable X stored by player P_i will be referred to as $P_i.X$, though if it is clear which player we are referring to we sometime only write X. Likewise when P_i executes a multiplication, we will write $P_i.\mathsf{multiply}$.

A very important fact to note for later is the following: for each of the subprotocols above, we can describe the operations each player has to do as a straight-line program consisting of a *constant* number of operations. Furthermore, the only operations needed are: multiply in G, invert in G, sample from G, and send/receive.

We now show how these protocols can be used together with our formula F to construct an n-player protocol for computing $h_1 \cdot ... \cdot h_m$.

Consider the formula F and the $T(2,3)$-gate g_0 computing the output of F. We think of g_0 as the top-most gate and then number the gates consecutively starting from the top level. We assign

[4]Depicted in Fig. 2

to the output wire of each gate g_i a virtual player P_i, and we also assign each input wire to a real player in the natural way according to the construction of the formula. Real players are assigned separate numbers.

Formally speaking, we will now execute the desired computation by giving the inputs $h_1, ..., h_m$ to P_0, have him execute P_0.multiply $m-1$ times and the output the result.

However, each virtual player will not do actually computation himself, instead he will be emulated by three other (virtual or real) players, using the protocols we described above.

Concretely, the virtual player P_i will be emulated by players P_j, P_k, P_l where these are the players assigned to the input wires of g_i. Note that this means that the virtual players assigned to input gates will be emulated by real players.

This leads naturally to a recursive specification of the operations we ask a virtual or real players to do, and procedures for giving input and getting results out. These are syntactically defined as follows:

- P_i.Input(Y, y), executed when another player or an external party wants to give input value y to player P_i, to be stored in variable Y.
- P_i.Output(X), returns the value in X.
- P_i.Sample(X), P_i samples a random group element and stores it in X;
- P_i.Invert(X, X^{-1}), P_i inverts the value in X and stores it in X^{-1}; and
- P_i.Multiply(X, Y, Z), P_i multiplies values in X and Y and stores the result in Z.

We show what the Input and Send operations would look like as examples.

P_i.Input(Y, y)

1. If P_i is a real player, store y in variable Y.
2. Otherwise, let P_j, P_k, P_l be the players emulating P_i. Then do the following:
   (a) Create a random left sharing y_1, y_2 of y.
   (b) Execute P_j.Input$(P_j.Y, y_2)$ and P_k.Input$(P_j.Y, y_1)$.

Note the way of naming values above means that variable name Y refers to shares of the same actual values on all levels of the tree. The next procedure P_i.Send$(P_u, P_i.X, P_u.Y)$ shows how to send the value in variable X from player P_i to player P_v and place result in variable Y. Note that sender and receiver are always on the same level in the formula so they are both virtual or both real.

P_i.Send$(P_u, P_i.X, P_u.Y)$

1. If P_i, P_u are real players, P_i sends the value of $P_i.X$ to P_u who stores it in variable $P_u.Y$.
2. Otherwise, let $P_j P_k, P_l$ be the players emulating P_i, and let P_v, P_w, P_t be the players emulating P_u. Then do the following:
   (a) P_j creates a left sharing of his value in X:
   P_j.Sample$(X_{j,1})$, P_j.Invert$(X_{j,1}, X_{j,1}^{-1})$, P_j.Multiply$(X_{j,1}^{-1}, X, X_{j,2})$.

(b) Send shares to P_v and P_w: $P_j.\mathsf{Send}(P_v, P_j.X_{j,2}, P_v.X_j)$, $P_j.\mathsf{Send}(P_w, P_j.X_{j,1}, P_w.X_j)$.

(c) P_k creates a left sharing of his value in X:
$P_k.\mathsf{Sample}(X_{k,1})$, $P_k.\mathsf{Invert}(X_{k,1}, X_{k,1}^{-1})$, $P_k.\mathsf{Multiply}(X_{k,1}^{-1}, X, X_{k,2})$.

(d) Send shares to P_v and P_w: $P_k.\mathsf{Send}(P_v, P_k.X_{k,2}, P_v.X_k)$, $P_k.\mathsf{Send}(P_w, P_k.X_{k,1}, P_w.X_k)$.

(e) Note that we now have a situation equivalent to P_u holding variables X_j, X_k, and these contain the shares in X that P_j and P_k hold. We therefore execute: $P_u.\mathsf{Multiply}(X_k, X_j, Y)$.

The sample, inverse and multiply operations can be specified in exactly the same fashion. This is straightforward to derive from the specification of the three player protocols above, but very long and tedious, so the reader will be spared the details.

We can now finally define the n-player protocol which we call π_F to emphasize that it is built from the formula F:

Protocol π_F

1. For $i = 1..m$, the real player holding h_i executes $P_0.\mathsf{Input}(H_i, h_i)$.
2. Do $P_0.\mathsf{Multiply}(H_1, H_2, T_2)$. Then, for $j = 3..m$, do $P_0.\mathsf{Multiply}(T_{j-1}, H_j, T_j)$.
3. Do $P_0.\mathsf{Output}(T_m)$, this causes real players to broadcast all their shares in T_m. Each real player multiplies shares as appropriate and returns the result.

We now have

Theorem 5. *Assume the formula F has logarithmic depth and computes the majority function on n inputs. Then the protocol π_F runs in time polynomial in n and m, it always computes correct results and there exists an efficient simulator S, such that if $t < n/2$ players are corrupted, then when given the corrupted players' inputs and the output, S produces a transcript with the same distribution as seen by the adversary by running the protocol.*

Proof. The running time is clear from the fact that F has logarithmic depth and that in each subprotocol we use for player emulation, each emulating player only executes a constant number of basic instructions. This means that the total number of operations done is $m \cdot c^{O(\log n)}$ for some constant c.

Correctness is clear from correctness of the subprotocols we use for player emulation.

We therefore turn to describing the simulator S:

If at most one honest player provides input to the protocol, the adversary and simulator can know all the inputs. In this case the simulator computes the input of the honest player (if any), runs the protocol on all inputs, and outputs a trace of this protocol run. Clearly this has the same distribution as a normal protocol execution. Informally, the adversary already knew everything so the protocol messages don't reveal any extra information.

Otherwise, the simulator works as follows: choose the neutral element as input for each honest player and run the protocol. Concretely, this means that the adversary plays for the corrupt players and the simulator plays for the honest ones following the protocol (but on dummy inputs). We stop this when we reach the output stage. In the output stage we construct and broadcast shares for the honest players leading to the correct result; below we describe how this is done.

To see that this simulator works as desired, we first argue that the simulation up to the output stage is perfect.

Consider first the input phase. In it, the simulator generates a sharing of the neutral element for each honest player and sends the appropriate shares to the adversary. If the underlying formula has height one and a player receiving a share is corrupt, that share is uniformly random. If the formula is taller, at most one virtual player who gets a share is corrupt; that player sees a uniformly random value, and by induction the corrupt physical players who participate in emulating the honest virtual players see only uniformly random values. This doesn't depend on the fact that element we're sharing is the neutral element; in other words, both in the simulated case and the real case, the adversary sees uniformly random elements in the input phase.

To analyse the computation phase, we first need some terminology: Let str_A be the characteristic vector for the corrupted set A of real players, i.e., str_A is an n-bit string with 1's corresponding to players in A and 0's elsewhere. We say that a virtual player P_i is corrupt if the sub-formula of F below gate g_i accepts str_A. So P_0 is always honest, and at most 1 of the players immediately below (namely P_1, P_2, P_3) is corrupt.

We now want to say that if P_i is corrupt, then the adversary knows the values of all variables held by P_i. This will be the case if he knows enough shares held by real corrupt players to reconstruct those values. But since our sharing among the three players emulating P_i only gives shares to the first two players, the adversary does not necessarily learn the value (namely if the last two players are corrupt). For the sake of this argument, however, we will give those values to the adversary for free. It is clear that this does not give the adversary more information about any of the values shared initially. We want to show that his view is independent of the honest players' inputs. We will show this even given the extra information, so the same is also true without it.

Now back to the computation phase. By inspection of the protocol, we see that the only time we execute a physical send operation between real players is in the (virtual) send and multiply operations. In both cases, a sender creates a fresh random multiplicative sharing of a group element and sends shares to two or three other players. If the sender is corrupt, the adversary already knows the value, so we will assume the sender is honest.

If at most one of the receivers is corrupt, the adversary sees nothing or a uniformly distributed group element.

Otherwise, if the send occurs as part of emulating a multiplication, the sender is one of the receivers, so the two other players may be corrupt. But then the virtual player we are emulating is corrupt, so the adversary already knows the values we are multiplying. Since the values sent only depends on those values, shares of them and fresh randomness, they are independent of the honest inputs.

The remaining case is if the send occurs to emulate one virtual player sending to another. In this case there are always two receivers, and if both are corrupt, the adversary learns the value sent, and we also know that the receiving virtual player is corrupt. Of course, if also the sending virtual player is corrupt, the adversary already knows the value sent, so we only need to consider the case where a virtual honest player sends to a corrupt one. This we can handle by reusing exactly the same argument as we just gave on the next higher level of the formula. This argument will stop when we reach the top-most group of three players since here there is no communication on a higher level between two different virtual players.

We conclude that everything the adversary sees in the input and computation phases is independent of the honest players' input, so the simulation of those phases is perfect. In particular, the distribution of the adversary's shares in the output is the same in simulation as in real life.

Only the output phase is left. We're to describe what the simulator does and why that works.

In the output phase, the players hold a sharing of the product $h_1 \cdot \ldots \cdot h_m$ and reconstruct this value by each player broadcasting their share. The simulator is to output messages which have the same distribution, given the global output and the shares the adversary already knows. This is sufficient by what we argued above.

Let us first analyse what the distribution is in the real world. Consider the root level: if a *shareholder* (player 1 or 2) is corrupt, then since the output is known, the adversary can compute the value of the other share. Inductively, the same happens to the players who implement the honest shareholder: their "output" (share value) is now externally specified, and so if one is corrupt the other's value is uniquely determined. If at any level both shareholders are honest, one share is chosen uniformly at random and the other is computed to match the output value (this is readily seen by examining the protocol).

What the simulator does is simply generate a sharing which has this distribution, by executing the algorithm which immediately follows from the above recursive description of the distribution. This is possible because the simulator knows the shares of corrupt players and the output.

6 An Approach to Obtaining Active Security

As mentioned earlier, an actively secure protocol for 4 players tolerating 1 actively corrupted player could be used together with our formulas built from $T(2,4)$-gates to get an n-player protocol with active security against $\frac{1}{6}(5-\sqrt{13})n$ players. At time of writing, however, we do not have such a 4-player protocol. On the other hand, we do know how to use the passive 3-player protocols as a "black-box" towards obtaining a 12-player protocol with security against 1 actively corrupted player.

Combining this with formulae built from $T(2,12)$-gates will lead to an n-player polynomial-time protocol with security against a constant fraction of actively corrupted players, and this was not known to be possible before, for any constant fraction. Applying a similar analysis to formulas with $T(2,12)$-gates as we did earlier for $T(2,4)$-gates, we find that such formulas of logarithmic depth exist that compute a threshold function with threshold approximately $0.017n$, so this is the number of actively corrupt players we can tolerate with this construction.

6.1 An Actively Secure Protocol for 12 Players

We now sketch the idea for building the 12-player protocol: we will call the players $P_1, .., P_{12}$. We will let π denote the collection of 3-player protocols we have seen in the previous section, for multiplication, inverse, sampling etc.. on shared values.

In essence, we will execute 4 instances of π. These instances will be executed by the subsets $T_1 = \{P_1, P_2, P_3\}, T_2 = \{P_4, P_5, P_6\}, T_3 = \{P_7, P_8, P_9\}$ and $T_3 = \{P_{10}, P_{11}, P_{12}\}$, respectively. All 4 instances of π will be executed on the *same input* and with the *same randomness*. We obtain this by also dividing the players in sets in a different way, $S_0 = \{P_1, P_4, P_7, P_{10}\}, S_1 = \{P_2, P_5, P_8, P_{11}\}$ and $S_2 = \{P_3, P_6, P_9, P_{12}\}$, and have players in each S_i coordinate their actions. Note that this makes sense because all players in S_1, for instance, play the role of the first player in π.

Note also that there is at most 1 actively corrupted player in a set and since each set has 4 players, we can do broadcast and Byzantine agreement inside each set using standard protocols. This can be used to make sure that the inputs are shared consistently, i.e., that all players in a set S_i hold the same share of each input. To select random group elements, we simply let the first

player in each set S_i do the selection and broadcast his choice inside the set. Of course, this player may be corrupt and may not do a random choice. The net effect of this is that π is executed in such a way that the one corrupt player it tolerates may not select his randomness with the correct distribution (but otherwise follows the protocol). However, the important observation is that π is secure even in this case, and in fact the same simulation will work to prove this.

We now execute all 4 instances of π in parallel. Since all inputs and randomness is the same, whenever π instructs a player to send a message, what we expect actually happens is that all players in some set S_i sends the same message to a corresponding player in S_j. Of course, this may not actually be true since a corrupt player is acting in one of the instances. Therefore the players in S_j compare what they have received and take majority decision. This ensures that all honest players are in a consistent state throughout, and we will therefore get correct results in the end.

6.2 Construction of a Protocol for n Players.

Finally, we need to consider whether the above 12-player protocol can be used with the player emulation approach we described earlier. More specifically, we need to check that the local computation a player is supposed to do can be emulated efficiently by a set of players (12 in our case).

First, we fix a way to represent values held by the player we are emulating. We derive this directly from the emulation of π, described in the previous subsection: to represent an element x we make a left sharing (x_1, x_2) and give x_1 to all players in S_1 and x_2 to all players in S_2.

Now, to execute the 12-player protocol, a player needs to do multiplication, inversion, sampling and sending (to execute π), and the only additional operation is to compare group elements.

To emulate the first 4 operations, we use the same idea as we used for constructing the 12-player protocol from π: the 12 emulating players run 4 instances of the emulated command from π, inside sets $T_1, ..., T_4$, and whenever something is sent (from set S_i to S_j), we compare the received values inside the receiving set as described above.

To emulate the comparison, we assume that the two elements to compare, g_1, g_2 are secret shared among the 12 players in the way we just described. We now compute and open $g_1^{-1}g_2$ which can be done by the operations we already have, and return "equal" if the result is the neutral element e and "not equal" otherwise. Note that this is not actually a secure comparison: the result of the comparison is leaked and if the values are not equal, information about g_1, g_2 is leaked. However, by the way in which comparisons are used in the 12-player protocol, this is actually sufficient: recall that whenever something is sent as part of the 4 instances of π that we execute, a player in the receiving set S_i will compare what he received to what the others got. If all senders and receivers were honest all comparisons will return equality (the adversary already knows this), and no further information is leaked. If the actively corrupted player was involved in sending or receiving, the adversary knows what the correct message is and how he (may have) modified it. Therefore again he knows when the comparison will return unequal and also knows both group elements that are compared.

The only other place where comparison is used is as part of the broadcast protocol that is used inside each set S_i. This can be done a simple deterministic protocol, and it is easy to construct it such that if all players are honest, then all comparisons return "equal". If one player is corrupt, the adversary learns in any case what the broadcasted value is and can predict the result of all local comparisons.

As a result, we can use the 12-player protocol together with a formula built from $T(2, 12)$ gates to get a protocol for n players that is secure against active corruption of a constant fraction of

players.

7 Local Conversion

A secret sharing scheme $\mathcal{S}$ is locally convertible to secret sharing scheme $\mathcal{S}'$ if for any set of shares $(S_1, \ldots, S_n)$ of a secret s computed according to $\mathcal{S}$, there exist functions $f_1, \ldots, f_n$ such that $(f_1(S_1), \ldots, f_n(S_n))$ is a set of shares that consistently determine s according to $\mathcal{S}'$. This notion was introduced in [3]. We expand this definition to also cover conversion between sharings of members of different sets. If $\mathcal{S}$ is a scheme for sharing secret values from a set A, and $\mathcal{S}'$ from B, we say there is a local coversion with respect to $f\colon A \to B$ if there exists functions $f_1, \ldots, f_n$ such that for any set of values $S_1, \ldots, S_n$ forming a sharing of $s \in A$ according to $\mathcal{S}$, the values $(f_1(S_1), \ldots, f_n(S_n))$ form a sharing of $f(s) \in B$. We can think of the narrower concept of local conversion as the special case where $A = B$ and f is the identity function.

Consider now the scheme we defined earlier: $F(\mathcal{S}_{2/3})$ for a formula F. Consider also the secret sharing scheme $\mathcal{S}^R_{2/3}$ defined over a ring R, as follows: given the secret $s \in R$, choose s_1, s_2 uniformly at random in R and set $s_3 = s - s_1 - s_2$. Finally, give $(s_2, s_3), (s_1, s_3), (s_1, s_2)$ to players 1, 2 and 3, respectively. When $R = \mathbb{Z}_p$ this scheme is clearly a perfect secret sharing scheme where sets of 2 or more players are qualified.

On local conversion in general, it is known that shares in the *replicated secret sharing scheme* (RSS) over any field K can be locally converted to any linear scheme over K. $\mathcal{S}^{\mathbb{Z}_p}_{2/3}$ is actually RSS for three players. RSS, however, is not efficient for a large number of players. It is also known that the Shamir scheme cannot be locally converted to RSS, but other than this, virtually nothing is known on local conversion. For our scheme, we have

Theorem 6. *For any monotone formula F built from threshold gates and any two rings R and R' between which there exists a ring homomorphism $f\colon R \to R'$, the players can locally convert from the secret sharing scheme induced by F over R, to the scheme induced by F over R' with respect to f. If F is multiplicative or strongly multiplicative, this is preserved through the conversion.*

Proof. The local conversion consists of applying f to each individual share component. Reconstructing the secret is done by doing several additions. Since f is a homomorphism, f applied to each sum equals the sum of f applied to the parts. Multiplicativity rests on a property of products which is similarly preserved by ring homomorphisms.

7.1 A more general lower bound

In [3], the concept of a generic conversion scheme for n players is defined, as follows:

Definition 2. *A generic conversion scheme for secret sharing scheme $\mathcal{S}$ with access structure Γ consists of a set of random variables $R_1, \ldots, R_m$, an assignment of a subset B_j of these to each player P_j, and local conversion functions g_j such that if each P_j applies g_j to the variables in B_j, we obtain values $(s_1, \ldots, s_n)$ forming consistent $\mathcal{S}$-shares of some secret s. Furthermore, given the information accessible to any unqualified set of Γ, the uncertainty of s is non-zero. Finally, for every R_i, there exists some qualified set A, such that the value of the secret s determined by shares of players in A depends on the value of R_i. More precisely: the uncertainty of s, given all variables known to A, except R_i, is non-zero.*

The last condition in the definition was not specified in the definition in [3], but was assumed in their proofs. It is clearly necessary to avoid redundant schemes: if a variable R_i makes no difference to any qualified set, it can be eliminated.

The interesting point about this concept is that a generic conversion scheme can be used to build so called pseudorandom secret sharing: by predistributing m keys to a pseudorandom function, players can locally generate values that are indistingusihable from the R_i's, and then convert these values to shares in a secret sharing scheme without communicating. Thus we have a way to generate random shared secrets with no communication. This is of course a very useful tool. Concrete constructions of this were given in [3], but they do not scale well with the number of players. Unfortunately, this cannot be avoided, due to a lower bound that was shown in [3] and which we strengthen here.

Note that neither $\mathcal{S}$ nor the conversion functions g_j are assumed to be linear. Also note that the convertibility requirement formulated above is weaker than the default requirement defined in [3]. However, we are about to look at negative results which are only made stronger this way. The following result is shown in [3]:

Proposition 1. *For any generic conversion scheme for $\mathcal{S}$ where $R_1, \ldots, R_m$ are independent and $\mathcal{S}$ has the property that a qualified set of players can reconstruct not only the secret, but also the shares of all players, it holds that m is at least the number of maximal unqualified sets.*

For a threshold secret sharing scheme where the threshold is a constant fraction of n, the number of qualified sets grows exponentially with n, so this result rules out efficient generic conversion schemes in many cases. If we want to somehow circumvent this lower bound, it is clear that we should either consider cases where the R_i are not independent, or cases where $\mathcal{S}$ does not have the property specified in the proposition. Since indeed our secret sharing schemes $F(\mathcal{S}_{2/3})$ do not have that property, one might hope that these schemes could lead to pseudorandom secret sharing with better complexity. This will not work, however. We show that the lower bound holds without the assumption on $\mathcal{S}$:

Theorem 7. *For any generic conversion scheme for $\mathcal{S}$ where $R_1, \ldots, R_m$ are independent, it holds that m is at least the number of maximal unqualified sets.*

Proof. First, we claim that for any qualified set A, it must be the case that all variables R_i are known to players in A.

Namely, assume this is not the case for some A, and consider some variable R_k which is not given to any player in A. However, there must be some other qualified set A' that does know R_k, and where R_k is needed for A' to determine the secret. Let $R_{-k}^{A'}$ be the set of R_i's known to A', except for R_k, and let R_{-k} be the set of all R_i except R_k. Finally for each qualified A we define a random variable S_A taking the value of the secret determined by players in A. The condition that R_k is necessary for players in A' means $H(S_{A'}|\ R_{-k}^{A'}\} > 0$. Since the R_i are independent, we even have $H(S_{A'}|\ R_{-k}\} > 0$.

On the other hand, the sharing computed by the players must consistently determine one secret s. More precisely, the demand is that $S_A = S_{A'}$ always, and for any A, A'. Obviously, we have $H(S_A|\ R_{-k}\} = 0$, but then we have a contradiction: since $S_{A'}$ is not fixed given R_{-k}, we cannot have $S_A = S_{A'}$ with probability 1.

To finalize the proof, we will construct a secret sharing scheme $\mathcal{S}'$ as follows: shares are defined to be any set of values that players can obtain in the generic conversion scheme we are given.

Reconstruction of the secret is done by converting these values to shares in $\mathcal{S}$ using the functions g_j, and doing reconstruction in $\mathcal{S}$. Clearly, there is a trivial generic conversion scheme from $R_1, .., R_m$ to $\mathcal{S}'$, namely where all local conversion functions are the identity. Moreover, we have just shown that $\mathcal{S}'$ has the property that any qualified set can reconstruct all other player's shares. Hence, by Proposition 1, m must be at least the number of maximal unqualified sets.

References

[1] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[2] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In Shafi Goldwasser, editor, *CRYPTO*, volume 403 of *Lecture Notes in Computer Science*, pages 27–35. Springer, 1988.

[3] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.

[4] Yvo Desmedt, Josef Pieprzyk, and Ron Steinfeld. Active security in multiparty computation over black-box groups. In Ivan Visconti and Roberto De Prisco, editors, *SCN*, volume 7485 of *Lecture Notes in Computer Science*, pages 503–521. Springer, 2012.

[5] Yvo Desmedt, Josef Pieprzyk, Ron Steinfeld, Xiaming Sun, Christophe Tartary, and Andrew Chi-Chih Yao. Graph coloring applied to secure computation in non-abelian groups. *J. Cryptology*, 13(1):31–60, 2011.

[6] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.

[7] Leslie G. Valiant. Short monotone formulae for the majority function. *J. Algorithms*, 5(3):363–366, 1984.