I/O-Efficient Multiparty Computation, Formulaic Secret Sharing and NP-Complete Puzzles

Jonas Kölker

Contents

1 Introduction **4**
- 1.1 Complexity Theory: **NP**-Completeness and Puzzles 4
- 1.2 Protocols for Secure Multiparty Computation 6
  - 1.2.1 Secret Sharing 8
- 1.3 Multiparty Computation and I/O Efficiency 9

2 The Magnets Puzzle is NP-Complete **9**
- 2.1 Reduction 10

3 The Kurodoko Puzzle is NP-Complete **11**
- 3.1 Proof of **NP** Membership 13
- 3.2 Overview of the Hardness Proof and Reduction 13
- 3.3 The Reduction 14
  - 3.3.1 Gadgets 14
  - 3.3.2 Manipulation and Combination of Gadgets 16
  - 3.3.3 Components and the Board 17
  - 3.3.4 Post-Processing the Board to Handle $v_?$ 20
- 3.4 Proof of Correctness of the Reduction 21
  - 3.4.1 Proof That Kurodoko Solvability Implies SAT Solvability 22
  - 3.4.2 Proof That SAT Solvability Implies Kurodoko Solvability 22
- 3.5 Discussion of the Reduction and the Result 24

4 Slither Link Variants are NP-Complete **25**
- 4.1 Membership of **NP** 25
- 4.2 The General Approach to Proving Hardness 26
- 4.3 The Dodecahedron Graph Class is Hard 27
- 4.4 The Triangular Graph Class is Hard 28
- 4.5 The Hexagonal Graph Class is Hard 30
- 4.6 The Dual Graph Class is Hard 31
- 4.7 Review and Discussion 32

5 Secret Sharing and Secure Computing From Monotone Formulae **32**
- 5.1 Overview 33
  - 5.1.1 A new Construction of Monotone Boolean Formulae . . . 33
  - 5.1.2 New Secret-Sharing Schemes 33
  - 5.1.3 Black-Box Computation over Non-Abelian Groups 35
  - 5.1.4 A Negative Result for Share Conversion. 36
- 5.2 Formulae for the Majority Function 36
- 5.3 A Linear Secret-Sharing Scheme based on $T(k, m)$ Formulae . . . 38
- 5.4 A Strongly Multiplicative Secret Sharing Scheme 42
- 5.5 Black-Box Computing over Finite Groups 46
  - 5.5.1 The Problem and a Solution for Three Players 46
  - 5.5.2 Construction of a Protocol for n Players 49

5.6 An Approach to Obtaining Active Security 54
5.6.1 An Actively Secure Protocol for 12 Players 54
5.6.2 Construction of a Protocol for n Players. 55
5.7 Local Conversion 56
5.7.1 A More General Lower Bound 57

6 Multiparty computation and I/O efficiency 58
6.1 Overview 59
6.2 Preliminaries 62
6.3 The Main Functionality 63
6.4 The Protocols 64
6.4.1 Passively Secure Implementation of $\mathcal{F}$ 64
6.4.2 Implementation for Malicious Servers and Players 67
6.4.3 A Scalable Way of Handling Malicious Players and Servers 67
6.4.4 Implementation of F_{Rand} 69
6.5 Running Oblivious Algorithms on $\mathcal{F}$ 72
6.6 Example Applications 75

A A python implementation of the Kurodoko reduction 82

B The Kurodoko gadgets 87
B.1 The Variable gadget 87
B.2 The Zero gadget 88
B.3 The Xor gadget 89
B.4 The Choice gadget 90
B.5 The Split gadget 91
B.6 The Sidesplit gadget 92
B.7 The Bend gadget 93
B.8 The Negation gadget 94

C Robust Read and Write Protocols Using Information Theoretic MACs 95

D Security of the Scalable Robust Protocols 98

E A Simulation Proof of Security of the $share^{+}$ Protocol 100

1 Introduction

1.1 Complexity Theory: NP-Completeness and Puzzles

Computational complexity theory is the study of the difficulty inherent in computational problems, defined in terms of resource usage in various computational models, and the relationships between the various classes of difficulty.

The two most important complexity classes are **P**, the set of decision problems for which a deterministic polynomial-time turing machine exists which solves it, and **NP**, the set of decision problems for which a non-deterministic polynomial-time turing machine exists which solves the problem. An equivalent definition of **NP** is the set of decision problems for which every yes-instance has a polynomial length certificate which is verifiable by a deterministic polynomial-time turing machine. For instance, if the problem is to decide whether a boolean circuit has an input for which the output is 1, such an input is (clearly) a certificate that such an input exists.

The reasons these classes are of partiular importance is that **P** is fairly stable across a large set of computational formalisms (e.g. one can translate in polynomial time between turing machines and realistic models of modern computers or other computational formalisms) and is a widely acepted theoretical notion of an algorithm that is efficient in practice. Many practically relevant problems fall in this class, for instance sorting, searching, parsing and linear optimization.

What makes **NP** important is the large number of practical problems in it, having to do with for instance with task scheduling, graph touring and subdivision, logical circuit and formula satisfiability, database normal form violations and many others.

Most important is the relationship between them. It is easily seen that **P** is a subset of **NP**: if we are given a poly-time machine M for solving some decision problem, we can construct a certificate checker which discards the certificate and runs M on the input. An interesting question is whether this inclusion is strict, or put more conventionally, whether $\mathbf{P} = \mathbf{NP}$. This problem was first raised in [Coo71] and remains open. It is one of the six unsolved millenium prize problems, a solution to which the Clay Mathematics Institute will pay one million dollars for (see [Ins]).

To highlight why this question is so important, we shall first discuss the notion of **NP**-completeness. A problem L is **NP**-hard if every other problem L' in **NP** is reducible to L; that is, if the is a polynomial-time computable function $f_{L'}$ such that $x \in L' \iff f(x) \in L$. A problem is **NP**-complete if it is in **NP** and if it is **NP**-hard. If we know of an **NP**-complete problem and we discover a polynomial time algorithm A which solves it, then we can solve every other **NP** problem in polynomial time by running A on $f_{L'}(x)$.

This highlights why the $\mathbf{P} = \mathbf{NP}$ question is so important: if we answer it affirmatively through a constructive proof, we instantly have (at least theoretically) efficient algorithms for an incredibly large of practically relevant problems. As we can encode mathematical conjectures (not least of which any remaining unsolved millenium prize problems) as propositional formulae, it would also be

of immense theoretical interest.

This also highlights why showing problems to be **NP**-complete is useful: it opens up avenues through which $\mathbf{P} = \mathbf{NP}$ can potentially be proven. One of the canonical **NP**-complete problems is 3SAT: given a boolean formula in conjuctive normal form with at most three literals per clause, decide whether there exists a truth assignment which satisfies the formula. The **NP**-completeness of this follows by duality from the first **NP**-completeness result [Coo71] due to Cook.

This result was expanded on by Karp [Kar72], who showed 21 other problems to also be **NP**-complete and developed the theory a bit further: by defining encoding functions one can talk about decision problems on something other than strings, e.g. graphs and sets of integers, and by showing polynomial equivalence of different encodings one can refer to the graphs independently of their encoding as e.g. adjacency lists or adjacency matrices. In addition, k-ary and m-ary integers are equivalent (polynomial-time convertible) for all $k, m > 1$, but unary and binary integers are not.

Algorithms which solve problems in polynomial time when all integers are encoded in unary are called pseudo-polynomial; one such example is subset sum: given a set S and a target sum of W, fill a bit matrix s_{ij} with bits indicating whether some subset of the first i elements sum to j, where $j = 0, \ldots, W$. This can be easily done in time $O(W \cdot |S|)$, which is polynomial in the value W; this is equal to $|enc_{unary}(W)|$ by definition. The corresponding hardness notion, problems remaining **NP**-complete when integers are given in unary, is known as strong **NP**-completeness. The relevance of this is naturally that if we believe that $\mathbf{P} \neq \mathbf{NP}$, then showing a problem to be pseudopolynomial or strongly **NP**-complete is evidence that it doesn't have the other property; and if our belief is wrong and we have a constructive proof, then reductions among strongly **NP**-complete problems will yield solutions to them.

The list of **NP**-complete problems was greatly expanded by a now widely cited textbook on the topic by Garey and Johnson [GJ79]. Around the same time, researchers began studying the computational complexity of puzzles and games—a recent survey paper on the topic [DH09] states that "in one of the first papers on hardness of puzzles and games people play, Robertson and Munro [RM78] proved that this generalized Instant Insanity problem is NP-complete."

The general trend in the field seems to be that many natural puzzles are **NP**-complete. By a quick survey of a collection of 34 puzzle implementations[1] [Tat10], 19 have relevant related problems that are **NP**-complete, 3 have problems in **P** and 13 are unknown to me. The overabundance ($19+3+13 = 35 > 34$) is due to the fact that the generalized version of the 15-puzzle has both an easy problem (solvability) and a hard problem (bounded-length solvability) associated with it. Note that the puzzles have been given names that are local to this collection; we give both this name and the names that are used more widely (and in particular in [DH09]).

The three easy problems are the existence of generalized 15-puzzle solutions,

[1] from which I derive recreational value and to which I have contributed two implementations

embedding of planar graphs in the plane (for untangle; see [Sw93]) and solving matrices over $\mathbb{F}_2$ (for flip; proof sketch: a constant number of for loops brings a matrix on reduced row-echelon form).

The hard problems are determining whether a solution exists, e.g. for Bridges (Hashiwokakero), Filling (Fillomino), Galaxies (Tentai Show), Guess (Master Mind), Keen (KenKen), Light Up (Akari), Loopy (Slither Link), Magnets, Pattern (Nonograms), Pearl (Masyu), Pegs (Peg Solitaire), Range (known as Kurodoko or Kuromasu), Same Game (Clickomania), Singles (Hitori), Solo (Sudoku), Towers (Skyscrapers), and Unequal (Futoshiki). The one outlier is Mines (Minesweeper), for which the **NP**-complete problem is deciding whether a state is consistent (i.e. contains no apparent violation, which doesn't guarantee a solution).

Many references to proofs are in [DH09]. For Keen, Towers and Unequal, the **NP**-completeness follows from the **NP**-completeness of completing partial latin squares [Col84]: for Towers and Unequal a no-change reduction does the trick, and for Keen one must add a single trivially satisfied clue to be well-formed. For Bridges, see [And09]. For Guess, see [SqZ06].

Our contribution consists of showing the **NP**-completeness of Kurodoko, Magnets and several variants of Slither Link. Some completeness results were known for Slither Link, including two of the variants we offer proofs for; the results for Kurodoko and Magnets, as well as the other Slither Link variants we touch on, are all new.

The reductions are reasonably efficient: for Magnets we reduce from monotone exactly-one-in-three-SAT, and produce an instance whose size is the number of variables times the number of clauses. For Kurodoko we reduce from exactly-one-in-three-SAT, and the reduction is of size $O(l(c + \log l))$, where l is the number of literals and c is the largest number of occurences of any one variable (not literal). For Slither Link, the reduction is from the Hamiltonian Cycle problem on hexagonal grids, and the reduction is of size proportional to the bounding box of the source graph. In all cases, this could in the worst case be quadratic, and in the best case linear (the $\log l$ is the running time of a sorting network, which could be $O(1)$ in the best case.)

1.2 Protocols for Secure Multiparty Computation

Now let us turn to multiparty computation (MPC). Loosely speaking, this covers the design and analysis of protocols which enable two or more parties, each with their own private input, to compute some agreed upon function, yielding some output to each party, with the goal of doing so securely.

One of the oldest multiparty computation problems studied is Yao's Millionaries [Yao82], in which two parties each provide an integer input, a and b, say, and the protocol gives the same output to both players: 0 if $a < b$ and 1 otherwise.[2]

[2]The name refers to a scenario in which two millionaires want to compare their fortues; the protocol solves exactly this problem.

This is a somewhat narrow problem, in the sense that there are only two players, and we only have a protocol for computing a single, fixed function. The more general multiparty computation problem, secure function evaluation, consists of finding protocols which allow any number of players to compute an arbitrary function.

An important step in this direction came from Yao [Yao86], showing how two parties can compute any polynomial time function on private inputs with computational security, assuming factoring is hard. (The cited work doesn't include this, but oral presentation of it does [Dam12].)

The following year Goldreich, Micali and Widgerson showed that the same can be done for an arbitrary number of players, assuming that a majority of them are honest and that we have a one-way trap-door function [GMW87]. The same result, but building on the quadratic residuosity assumption this time, was independently discovered by Chaum, Damgård and van de Graaf [CDvdG87].

In 1988 work by Ben-Or, Goldwasser and Widgerson [BOGW88] and independent work by Chaum, Crépau and Damgård [CCD88] showed that function evaluation could be done securely in an information-theoretic sense, against an adversary corrupting less than $\frac{n}{2}$ players passively or $\frac{n}{3}$ players actively. This was later generalized to any Q_2 and Q_3 adversary structure A, respectively, by Hirt and Maurer [HM00], meaning that the adversary can corrupt any one set of players that is in A, and A is Q_k if the set of all players can't be written as a union of k or fewer elements of A. Note that all $\frac{n}{k}$ threshold adversary structures (those containing player sets of size less than $\frac{n}{k}$) are Q_k.

Perhaps surprisingly, the generally accepted security model, Universal Composability, was developed much later; the most important result is by Ran Canetti in 2000 [Can00]. Briefly sketched, functionalities are defined as ideal "black boxes", i.e. collections of relations between inputs and outputs; a protocol is said to securely implement a given functionality against an adversary with a set of capabilities if there exists a simulator which in a sense can pad or patch the ideal world, so that looks indistinguishable in any adversarial environment that only uses the given capabilities. The security is said to be perfect, statistical or computational if the indistinguishability has the same properties.

The most used parameterizations of the adversary's capabilities are which sets of players the adversary can corrupt (the adversary structure), whether the adversary can choose whom to corrupt on the fly (adaptively) or must choose so ahead of time (statically), and whether the adversary only gains the ability to read the corrupted players' sent and received messages (passive corruption) or also gains the ability to choose which messages the corrupted players send (active corruption).

The most important property of this security definition from which the name is derived, and which is Canetti's crucial result, is that if for some set of adversarial capabilities a protocol π securely implements a functionality $\mathcal{F}$ and another protocol π' securely implements another functionality $\mathcal{F}'$ while using $\mathcal{F}$ as a component, and we insert π as a replacement of $\mathcal{F}$ in π', the resulting protocol is seure. In other words, we can decompose the proof of the security of complex protocols into proofs of seurity of components and their combinations.

In yet other words, the components are composable.

As it turns out, the protocols in [BOGW88] and [CCD88] are secure in the strongest sense; that is, against an adaptive adversary, and with the largest possible thresholds for both active and passive corruption.

1.2.1 Secret Sharing

A useful tool for implementing secure function evaluation protocols is secret sharing, by which we understand an access structure (a set of sets of players) and a pair of protocols by which (1) a dealer can input a secret value s to generate a collection of shares $s_1, \ldots, s_n$ which are sent to a collection of players; and (2) any set of players that is a member of the access structure can combine their shares to reconstruct the secret, while any other set of players can't obtain information about the secret (in either a computational, statistical or perfect sense). The classical result in secret sharing is from 1979 and due to Adi Shamir [Sha79], who demonstrated that if the dealer chooses a random polynomial f subject to $f(0) = s$ and $\deg f \leq t$, then sets $s_i = f(i)$, this an information theoretically secure and efficient secret sharing scheme.

Shamir's secret sharing scheme is used in the protocols from [BOGW88]. Research by Cramer, Damgård and Maurer [CDM00] showed how to perform passively (actively) secure multiparty computation based on any multiplicative (strongly multiplicative) linear secret sharing scheme.

While having many useful properties, a limitation of Shamir's scheme is that it is limited to a threshold access structure. A result by Benaloh and Leichter [BL90] shows how one can define monotone access structures in terms of monotone formulae, and how to base secret sharing schemes on such formulae. Since any monotone access structure has a corresponding monotone formula, we can have secret sharing schemes for any monotone access structure, although it may not be trivial how to find (polynomially) small formulae for a given access structure.

Coming full circle, a result by Valiant [Val84] shows the existence of log-depth (and thus polynomial-size) monotone formulae for the majority function, based on **and** and **or** gates. By applying a similar technique, we show how to construct log-depth formulae for the majority function based on majority-of-three gates; one advantage of our resulting scheme is that it is multiplicative, and thus the results of [CDM00] yield an MPC protocol from this.

We also show a construction of a threshold function which yields a strongly multiplicative secret sharing scheme, and thus an MPC protocol that's secure against active corruption.

The majority formulae can also be used to construct a secret sharing scheme suitable for doing computation in non-abelian groups; a problem that has been studied by Cristophe Tartary et al. [DPS+11]. Our result is an improvement over their work in that we show that passively secure protocols with polynomial complexity and no error probability exist. We also illustrate an approach for obtaining active security, an important open problem.

Lastly, we improve on a negative result about local convertibility.

1.3 Multiparty Computation and I/O Efficiency

The computational model of secure function evaluation implicit in the previous section is one of arithmetic circuit evaluation, typically over a finite field. In the protocol from [BOGW88], the players hold secret sharings of the inputs and intermediate values, and open the secret sharings at the end of the computation.

Suppose now that the players want to offload the need for storing intermediate values to a set of storage servers. To do so, they would need a pair of protocols to transfer data from the players to the storage servers and in the other direction. Naturally, we would want this to be secure and efficient.

In the real world, the computing players and the storage servers may have different security characteristics; for instance, the computing players may be home PCs while the storage servers are operated by professional, security-conscious systems administrators. With the participants being heterogeneous, we argue that the adversarial capabilities are best modelled as being assymetric: the adversary can potentially corrupt different proportions of participants in the two different classes, and possibly even in qualitatively different ways (actively vs. passively).

We provide protocols for a broad selection of such combinations of adversarial capabilities and show these to be secure.

We also analyze the efficiency of these protocols, and of the storage of data while in rest. To achieve efficient data storage, we utilize a variation on Shamir's secret sharing scheme whereby we can store multiple secret values in one polynomial (trading off some security to have this). This means that when we retreive data from the storage servers, we always retrieve more than one value. This, combined with the fact that kind of protocol efficiency we can hope to achieve (and do achieve), we both have the desire and potential option to cut down on the number of times we execute the transfer protocols.

How to solve algorithmic problems in a context where multiple data elements are involved in every transfer operation—I/O efficient algorithms—is a problem that has been studied before, e.g. by Aggarwal and Vitter [AV88], though not in any MPC context. We show that problems with algorithms that are both I/O-efficient and efficient in the conventional in-memory sense can be executed efficiently in our I/O-MPC model.

2 The Magnets Puzzle is NP-Complete

In a Magnets puzzle, one must pack magnets in a box subject to polarity and numeric constraints. In this section, we show that deciding solvability of Magnets instances is **NP**-complete.

The puzzle is of unsure origin. Simon Tatham publishes an implementation, [Tat10] citing Janko.at [JJ] as the source, which claims the puzzle to be of unknown pedigree. The puzzle is also unmentioned in the survey paper [DH09] referenced in the introduction. In a Magnets puzzle, one is given a $w \times h$ rectangle, a subdivision of this rectangle into dominos, and $2(w + h)$ integers

denoted r_i^+, r_i^-, c_j^+ and c_j^- for $i = 0, ..., h-1$ and $j = 0, ..., w-1$.

The goal is to assign a value $\{-1, 0, +1\}$ to each square (representing positive or negative magnetic poles, or non-magnetic material) such that (1) the sum of the two values in the same domino is 0; (2) Each pair of horizontally or vertically adjacent numbers (x, y) satisfies $x = y = 0 \vee x \neq y$; and (3) in each row i, $+1$ occurs r_i^+ times and -1 occurs r_i^- times, and similarly for c_j^+ and c_j^- with respect to columns j, that is, the number of positive poles per row (column) equals the positive row (column) constraint, and likewise for negatives.

2.1 Reduction

Theorem 1. *The problem of deciding whether a given Magnets instance has a valid solution is NP-complete.*

Proof. It is easy to see that the problem is in NP: give a solution as a certificate. Its size is proportional to the input (problem) size and its verification is straightforward.

To show hardness, we give a reduction from monotone 1-in-3 boolean satisfiability, which is known to be NP-complete [Sch78]; i.e. given n variables x_i and m clauses C_j, each containing at most three variables, find a set of true variables T such that $|T \cap C_j| = 1$ for every $j = 0, \ldots, m-1$.

The corresponding magnets instance has $w = 4n$ and $h = 2m$, and the rectangle will be divided into $n \cdot m$ gadgets, each of size 4×2.

The intuition of the gadgets are as follows: the leftmost column of each gadget insulates it from its left neighbor, its two rightmost columns connect it to all the other gadgets associated with the same variable, and its fourth column determines the truth value of its corresponding variable in those gadgets whose associated clause contains the gadget's associated variable.

In gadget (i, j), if $x_i \notin C_j$ the two leftmost dominos will be horizontal, otherwise vertical. The rightmost two dominos will always be vertical.

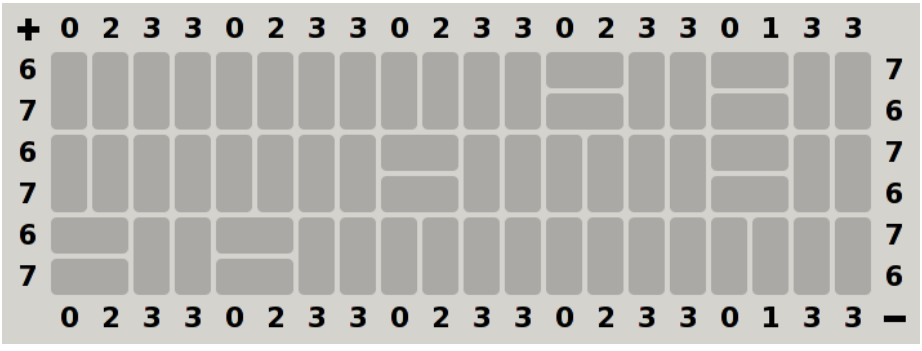


Figure 1: Puzzle for the instance $\{(x_0, x_1, x_2), (x_0, x_1, x_3), (x_2, x_3, x_4)\}$

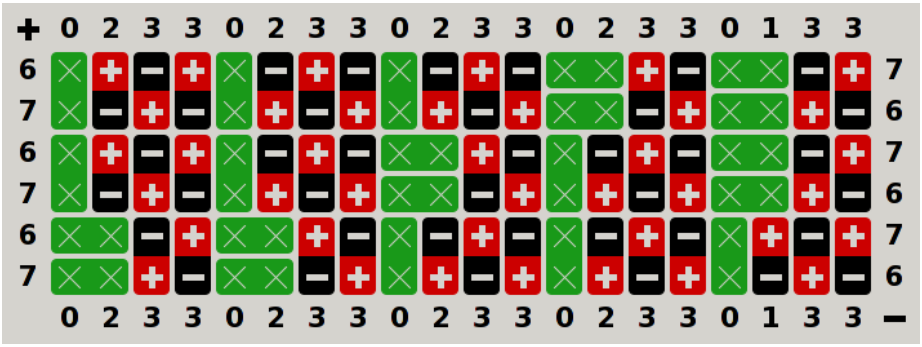


Figure 2: Solution to example puzzle, where crossed green squares represent 0

For $i = 0, \ldots, n-1$, let $c^+_{4i} = c^-_{4i} = 0$, let c^+_{4i+1} be the number of clauses containing x_i, and let $c^-_{4i+1} = c^+_{4i+1}$. Let $c^+_{4i+2} = c^+_{4i+3} = c^-_{4i+2} = c^-_{4i+3} = m$.

For $j = 0, \ldots, m-1$, set $r^+_{2j} = n+1$ and $r^-_{2j} = n + |C_j| - 1$. Let $r^+_{2j+1} = r^-_{2j}$ and $r^-_{2j+1} = r^+_{2j}$.

We claim there is a one-to-one mapping between monotone 1-in-3 SAT solutions and Magnets solutions: $x_i \in T$ iff square $(4i+1, 2j)$ has value $+1$ for every j such that $x_i \in C_j$.

Observe that as $c^+_{4i} = c^-_{4i} = 0$, no leftmost column of any gadget contains a non-zero value. As $c^+_{4i+2} = c^+_{4i+3} = m$, all values in columns $4i+2$ and $4i+3$ are non-zero. This contributes one $(+1)$ and one (-1) to each row.

Thus, for each j, row $2j$ contains n copies of $+1$ plus those in columns $4i+1$, and by r^+_{2j} it must contain $n+1$ copies of $+1$. In other words, exactly one of the squares $(4i+1, 2j)_{i=1}^n$ must contain $+1$ (for each $j = 0, \ldots, m-1$).

Note that iff $x_i \notin C_j$ then square $(4i+1, 2j)$ contains 0: in that case its horizontally adjacent domino buddy contains 0 by the above argument. Note also that in each connected region of non-zero squares, any two squares of even manhattan distance have equal polarity (i.e. value) and squares of odd distance have unequal polarity. Thus, if $(4i+1, 0) \neq 0$, then $(4i+1, 2j) = (4i+1, 0)$ for each $j = 0, \ldots, m-1$.

In other words, any assignment of values to squares is consistent with $x_i \in^? C_j$ according to the defined bijection, and well-defined (either x_i is or isn't in T, but not both), so this bijection maps Magnets solutions to monotone 1-in-3 SAT solutions as claimed.

3 The Kurodoko Puzzle is NP-Complete

In a Kurodoko puzzle, one must colour some squares in a grid black in a way that satisfies non-overlapping, non-adjacency, reachability and numeric constraints specified by the numeric clues in the grid. We show that deciding the solvability of Kurodoko puzzles is **NP**-complete.

According to [Wei] Kurodoko was invented by the Japanese publisher Nikoli [Nik11a]. The name comes from "kuro" meaning black and "doko" meaning where, i.e. "where [are the] black [squares]?". We'll show that solving Kurodoko is **NP**-complete. From a bird's eye view, our proof is very much similar to many other puzzle hardness proofs, in that we produce subpuzzles which capture some part of the problem we reduce from, then combine these subpuzzles in ways that make the global solutions derived from local solutions correspond to solutions to the problem we reduce from.

Kurodoko is played on a rectangular grid of size $w \times h$, $V := \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$. The squares are initially blank, except for a subset of squares $C \subseteq V$ which contain clues, integers given by a function $N \colon C \to \{1, \ldots, w + h - 1\}$. We can think of the rectangular grid as a grid graph $G = (V, E)$, where $(v, v') \in E$ if and only if v and v' are horizontally or vertically adjacent, i.e. $E := \{((r, c), (r', c')) \mid |r - r'| + |c - c'| = 1\}$.

The player's task is to come up with a set of black squares $B \subseteq V$, the rest being white, $W := V \setminus B$, such that the following four rules are satisfied:

1. The clue squares are all white, $C \subseteq W$ (or equivalently, $B \cap C = \emptyset$).

2. None of the squares in B are adjacent to any other square in B. i.e. $(B \times B) \cap E = \emptyset$.

3. All white squares are connected via paths of only white squares, i.e. the induced subgraph on the white squares $G_W = (W, E \cap (W \times W))$ is connected.

4. The number at each clue square equals the number of white squares reachable from that square, going in each of four compass directions and never off the board nor through a black square, i.e. $\forall (r, c) \in V : N(r, c) = hz + vt - 1$ where hz is the length of the longest horizontal run of white squares going through (r, c), i.e. $hz := \max\{k \in \mathbb{N} \mid \exists b : 0 \leq b \leq c \leq b + k - 1 < w \wedge \{(r, b + i)\}_{i=0}^{k} \subseteq W\}$. Similarly, vt is the length of the longest such vertical run. We say that (r, c) *touches* the squares in these runs, including itself.

We show that the Kurodoko Decision Problem, "given w, h, C and N and, is there a set $B \subseteq V$ satisfying the above criteria?", is **NP**-complete. We take w and h to be represented as binary encodings, C as a list of points (vertices) and N as a list of integers; the i'th integer in the list representing N is the function's value at the i'th point in the list representing C.

To help the reader gain an understanding of these rules and their implications we offer a simple observation:

Theorem 2. *If (w, h, C, N) is solvable and $\exists v \in C : N(v) = 1$, then $1 \in \{w, h\}$ and $w + h \leq 4$.*

Proof. Let (w, h, C, N) be given and suppose that $w, h \geq 2$. Let v be given with $N(v) = 1$. Then, since $w \geq 2$, either v has a right or left neighbour, or both; let

us call any one such neighbour v'. Since $h \geq 2$ this neighbour v' has a neighbour below or above, or both. In either case, call such a neighbour v''. By rule 4, all neighbours of v must be black. By rule 2, all those neighbours' neighbours must be white, including v''. But since v is surrounded by black squares, there cannot be a path from v to v'' that steps on only white squares, violating rule 3. Therefore, w and h can't both be at least 2; assume by symmetry that $h < 2$. It cannot be that $h = 0$ or there would be nowhere for v to be found ($V = \emptyset$), so $h = 1$. Assume $w > 3$; then v must either be a corner square with a black neighbour, or have two black neighbours, at least one of which has a white neighbour that isn't v. In either case, there is at least one white square that doesn't have a white-only path to v, which rule 3 requires. So $w \leq 3$, and hence $w + h \leq 4$.

3.1 Proof of NP Membership

We're now ready to state and prove the first part of our **NP**-completeness claim.

Theorem 3. *The Kurodoko Decision Problem is in **NP**.*

Proof. We show there is a polynomial time witness-checking algorithm. The witnesses will be solution candidates, i.e. candidates for B represented as a list of points. Given such a list, we can easily verify rule 1 in time $O(|B||C|)$, by two nested loops. We can also verify rule 2 easily in time $O(|B|^2)$ by testing for every (r, c) and (r', c') in B that $|r - r'| + |c - c'| \neq 1$. Verifying rule 4 is also fairly easy: for each clue square v, find the closest black square in each of the four compass directions. Then, compute $hz + vt - 1$ and check that it equals $N(v)$. This takes time $O(|C||B|)$.

To check rule 3 if $w > 2|B| + 1$ or $h > 2|B| + 1$, compress the grid by merging adjacent rows that don't contain any black squares and do the same for columns. Then $w, h \leq 2|B| + 1$. Find a white square by going through each square until a white square is found. If no white square is found, accept if and only if $w = h = 1$ (if not $w = h = 1$ then rule 2 is violated). If a white square v_w is found, verify by DFS that each white square has a path to v_w (or equivalently, that the number of white squares reachable from v_w plus the number of black squares equals the total number of squares). Accept if and only if this is satisfied.

We note that if w and h were given in unary then the compression step would not be necessary since the size of the board would be polynomial in the length of the input. The rest of this part is devoted to proving the next theorem.

Theorem 4. *The Kurodoko Decision Problem is **NP**-hard.*

3.2 Overview of the Hardness Proof and Reduction

We prove the **NP**-hardness of Kurodoko-solvability, by showing how to compute a reduction from one-in-three-SAT. This problem was shown to be **NP**-complete in [Sch78].

We do the reduction by describing a set of *gadgets*, a set of 17×17 square-of-squares containing clues, which are very amenable to combination and act in circuit-like ways, e.g. as wires, bends, splits, sources, sinks and so forth. These are combined to form three kinds of *components*: one kind acting like SAT variables, a second kind acting like SAT clauses, and a third kind acting like a matching between the components of the two first kinds, such that each clause component gets routed to the components corresponding to exactly those variables referenced in the clause. When taken as a whole we refer to the gadgets as the *board* (as in "printed circuit board").

We will show that in every solution to the Kurodoko instance resulting from the reduction, the gadgets will behave according to fairly simple descriptions which capture their circuit properties (e.g. wires behave like the identity function). This in turn implies that the components each behave according to a description which captures the connection to that SAT problem, such that SAT problem must have a solution.

For the reverse direction, we show how to compute a Kurodoko solution from a given SAT solution, and verify that this Kurodoko solution is indeed a solution, i.e. that it satisfies the four rules which define solutions.

3.3 The Reduction

In this section, we describe in more detail first the gadgets, then how to manipulate and combine them into components. Next, we describe which components we combine gadgets into, how the components are combined into boards, and finally how to handle a deferred issue which requires global information about the board. In Appendix A we provide an implementation of the reduction in order to ensure there is no ambiguity.

3.3.1 Gadgets

The reduction uses nine different gadgets named Wire, Negation, Variable, Zero, Xor, Choice, Split, Sidesplit and Bend. As an example see Figure 3 for an illustration of the Wire gadget.

Diagrams of the remaining gadgets are provided in Appendix B. Note that in all of them, the outermost rows and columns are either empty, or contain the centered sequence {a question mark, no clue, the clue 2, no clue, a question mark}.

Formally, we can think of the Wire gadget as a Kurodoko instance $wire = (17, 17, C, N)$, where C is the set of clue squares and N is the set of clue values. Some squares $v_?$ contain question marks; formally, we give them the value 1 for now. Their true value will be established once the pattern of how gadgets are combined is known, which depends on the SAT instance given to the reduction. When we speak of deductions about the board, we mean with the true value in place.

The eight marked squares ($\{n, w, e, s\}_{\{i,o\}}$) are not elements of C, and are in fact not a part of the gadget. The behavior of the gadgets can be succinctly

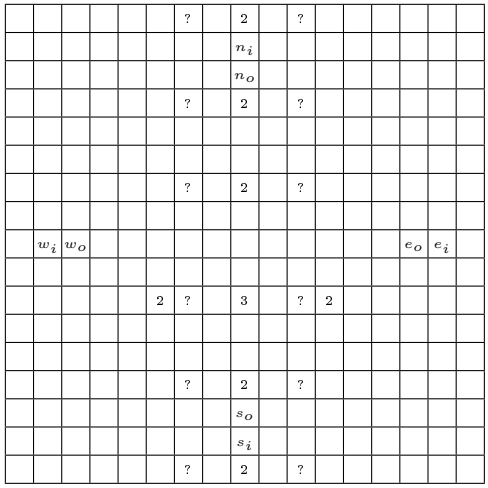


Figure 3: The Wire gadget

described in terms of the colours of these squares. We think of d_i as an input square (for $d = n, w, e, s$) and d_o as an output square. In some sense we want to think of the square in the center row or column immediately outside the gadget as the true output square, but this square will have the same colour as d_o. We treat black as 1 and white as 0. With this, we offer a description of the gadgets:

- Wire: $n_o = s_i$: the north output equals the south input.
- Negation: $n_o = 1 - s_i$: the north output is the opposite of the south input.
- Variable: $n_o \in \{0, 1\}$: the north output is always 0 or 1.
- Zero: $n_o = 0$: the north output is always 0.
- Xor: $n_o = e_i \oplus w_i$, the north output equals the xor of the east and west inputs.
- Choice: $w_i + s_i + e_i = 1$, exactly one input is 1.
- Split: $w_o = e_o = s_i$: the west and east output equals the south input.
- Sidesplit: $n_o = e_o = s_i$: the north and east output equals the south input.
- Bend: $e_o = s_i$: the east output equals the south input.

We claim now—and prove later—that in any instance created by our reduction, these relationships have to hold or the instance doesn't have a solution.

3.3.2 Manipulation and Combination of Gadgets

We note that the gadgets are square and can be mirrored and rotated. For compactness, let us introduce some notation for this: if G is a gadget, then G^+ is G rotated 90° clockwise, G^- is G rotated 90° counterclockwise, G^2 is G rotated 180° and $\overline{G}$ is G mirrored through a vertical line. We call gadgets by their first letter, except for Split which we refer to as T (in Appendix B, it looks like a tee).

Rotating and mirroring gadgets has the expected consequences to their operation. For example, if we mirror and then rotate clockwise a Bend gadget, the result is a gadget that takes an input on the west edge and outputs this on the north edge, i.e. $n_o(\overline{B}^+) = w_i(\overline{B}^+)$. Note that this is different from $\overline{B^+}$, where $s_o = e_i$.

Note that if an input square x_i is adjacent to a clue 2, then its adjacent output square x_o must contain the opposite colour of the input square, $x_i \oplus x_o = 1$, or else rule 2 (two blacks) or rule 4 (two whites) would be violated. This implies a curious property of the wire gadget: $s_o = 1 - s_i = 1 - n_o = n_i$; that is, it works in the other direction too. The same is true for Bend and Negation. For the Xor gadget, we see that $n_o = e_i \oplus w_i \;\Rightarrow\; n_o \oplus e_i = w_i \;\Rightarrow\; (1 \oplus n_o) \oplus e_i = 1 \oplus w_i \;\Rightarrow\; n_i \oplus e_i = w_o$.

This is why there is no Side-Xor: for X^+ we have $n_o = s_i \oplus e_i$, i.e. X^+ works as a hypothetical Side-Xor would. Likewise, X^- behaves as a $\overline{\text{Side-Xor}}$, in that $n_o = s_i \oplus w_i$.

Let us next define what it means to combine multiple gadgets. Let's say we have a set $K = \{(g, r, c)_i\}_{i=1}^k$ of k gadgets, the i'th gadget $g = (w_g, h_g, C_g, N_g)$ having coordinates $(r, c) \in \mathbb{N}^2$ on the combined board. Then, in the combined instance,

$$\begin{aligned} h_K &= \max\{16r + w_g \mid ((w_g, h_g, C_g, N_g), r, c) \in K\} \\ w_K &= \max\{16c + h_g \mid ((w_g, h_g, C_g, N_g), r, c) \in K\} \\ C_K &= \bigcup_{(g,r,c)\in K} \{(16r + i, 16c + j) \mid (i, j) \in C_g\} \end{aligned}$$

Furthermore, for each $(r, c) \in C_K$ where $r = 16i + r'$ and $c = 16j + c'$ and $0 \le r', c' \le 16$ and $(g, i, j) \in K$ we have $N_K(r, c) = N_g(r', c')$.

In other words the gadgets are placed next to one another, such that the rightmost column of each gadget overlaps the leftmost column of its right neighbour, and similarly in each other compass direction. The set of clues in the instance (w_K, h_K, C_K, N_K) is the union of the clues in each of the gadgets, suitably translated. For this to be well-defined, the clue values in the overlap areas must agree between the two overlapping gadgets.

If two overlap areas are empty (in both gadgets), this clearly isn't an issue. If one gadget is non-empty in the overlap, it simply "overwrites" the (overlap-)empty gadget, although this will never happen. If they are both non-empty, then they are in fact equal in the overlap area, per our previous remark about their outermost row and column structure. So this will always be well-defined.

3.3.3 Components and the Board

We have just seen how to combine a collection of gadgets K into an instance (w_K, h_K, C_K, N_K). As it happens, components are just such instances. Let us consider as an example the clause component. It has several variants; we show first the variant corresponding to a clause (x, y, z):

B	W^+	C	W^-	$\overline{B}$
W		W		W

Next the variant for a clause $(x, \neg y)$:

B	W^+	C	W^-	$\overline{B}$
W		N		W

Finally the same clause, with variables gadgets added to show the component in a context:

B	W^+	C	W^-	$\overline{B}$
W		N		W
V_x		V_y		Z

Formally speaking, the first clause component can be described as a combination (by the above rule) of $\{(B,0,0)$, $(W^+,0,1)$, $(C,0,2)$, $(W^-,0,3)$, $(\overline{B},0,4)$, $(W,1,0)$, $(W,1,2)$, $(W,1,4)\}$.

Hopefully it is clear how these work. Assuming the W- and N-gadgets have neighbours to the south which provide input (as in the right hand example), this input goes through the W or N and either directly into the C gadget, or through a bend that points it towards the C which it reaches after going through either W^+ or W^-. The pattern is this: the top row is identical in all variants of the clause component. In the bottom row, columns one and three are always empty. For a clause with k variables where $k < 3$, the rightmost $3-k$ even-indexed columns contain W-gadgets (these will be connected to a Z input elsewhere). The remaining columns contain W or N, depending on whether the corresponding variable in the clause is negated or not. For instance, the central N is the middle component is there (rather than a W as in the left component) because y is negated. The rightmost W is connected to a Z since the clause only has two variables.

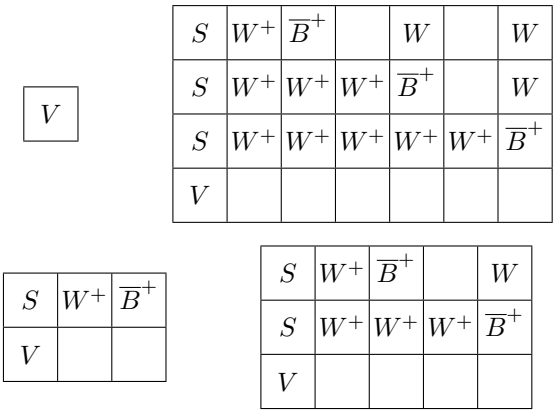
Next we show the zero component. This one is rather simple:

Z

As we will see later, instances of this will be connected to clause components for those clauses that refer to less than three variables.

Next, let us consider the variable component. Its structure depends on how many times the variable is used. We show the structure for variables used once,

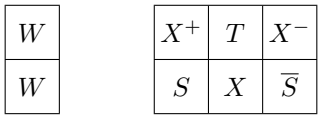
twice, thrice and four times:



Once again, it should hopefully be clear what happens. The output produced by V is repeatedly split to go both up and to the right. The copies continue to the right until they can be bent upwards and continue up, such that the variable component produces k equal outputs with a non-outputting square between each output.

Also, the structural pattern should be fairly clear: a variable used k times produces a component of height k and width $2k-1$. The bottom row always has a V in its leftmost square—only this, and nothing more. Other than this, row i from the top has (from left to right) an S, then $2i+1$ times W^+, then $\overline{B}^+$, then $k-i-1$ times {nothing, followed by W}.

Finally, we have the routing component. This is built up of multiple copies of two kinds of subcomponents, wires and swaps:



Clearly the (northern) output of the wire component is the southern input. Let a denote the input to S and b the input to $\overline{S}$. Then the output of X is $a \oplus b$. This is fed through T as input to both X^+ and X^-; the input to X^+ is a (from S) and $a \oplus b$ (from T), and so the northern output must be $a \oplus (a \oplus b) = b$. Similarly, the northern output of X^- is a.

In order to best motivate the structure of the routing component, we need to look at the structure of the Kurodoko instance produced by the reduction. In the top, there will be one clause component for every clause in the SAT instance; each pair of adjacent components has one column of "air", unused space, between them. At the bottom, there is one variable component for every variable that occurs in a clause somewhere (with the appropriate number of repetitions built into the variable component), as well as k zero components, where k is the

amount of "extra space" in the clauses, $k = 3m - \sum_{j=1}^{m} |C_j|$, three times the number of clauses minus the total number of literals.

What the routing component does is essentially label each output of the bottom part with the index of its goal column and then sort the signals.

Internally, create an array A (initially $3m$ copies of $\perp$) corresponding to the outputs of the bottom part of the board. For $i = 1, \ldots, 3m$, if the $\lfloor \frac{i}{3} \rfloor$'th clause has at least $(i \bmod 3) + 1$ variables, store i in the leftmost non-$\perp$ entry of A which corresponds to the $(1 \bmod 3+1)$'th variable; if not, store i in the leftmost non-$\perp$ entry of A which corresponds to a zero component.

Then, run some comparison-based sorting algorithm which only compares adjacent elements[3]. For $t = 1, \ldots, t_{max}$, place a row of swap and wire subcomponents on top of the bottom part of the board, putting swap subcomponents between entries that are swapped at time t and wire subcomponents at entries that are left unchanged at time t. Here, t_{max} is the smallest number such that the algorithm never makes more than t_{max} layers of swaps. In the case of the odd-even transposition sorting network, t_{max} is $3m$, the number of elements to be sorted. Note that once A is sorted we are free to stop early (i.e. the height of the routing component may depend on more features of the SAT instance than just its size).

These are the components. They are combined into a board by the same rule used to combine gadgets into components. To summarise the description of the structure of the board, given a SAT instance: on top there's one clause component for every clause in the SAT instance with at least one variable. At the bottom, there's a variable component for every variable contained it at least one clause, and at the bottom right k zero components for every clause with $3 - k$ variables ($k = 1, 2$). In the middle, there's a routing component. As an example, consider the SAT instance $(\neg v_2, v_1)$. The corresponding Kurodoko instance looks like this:

B	W^+	C	W^-	$\overline{B}$
N		W		W
X^+	T	X^-		W
S	X	$\overline{S}$		W
V		V		Z

Note that every gadget with one or more outputs is placed adjacent to other gadgets that have these outputs as their inputs (and vice versa). In other words, in the areas where gadgets overlap, either both (all) gadgets contain no clues in the overlap, or they do contain clues and the clues match up. Also, no gadget contains clues in the outermost rows or columns of the board.

[3]Examples include insertion sort, bubble sort and the odd-even transposition sorting network

3.3.4 Post-Processing the Board to Handle $v_?$

In Appendix B we show the gadgets. For each of the gadgets, we can make deductions about the colour of some of the squares, under the assumption that instance containing the square has a solution. In other words, some squares are necessarily the same colour in all solutions. In this section, *white (black) squares* refers to squares that are white (black) in all solutions.

For some squares containing a question mark in three directions going out from that square there are black squares within the question mark's gadget (whose blackness can be deduced independent of the value in the square with the question mark), while in a fourth *open* direction there are no black squares within the gadget. We call these squares *end squares* (e.g., there are six of those in S). Suppose that we draw a line from each end square in the open direction and its opposite until we hit the edge of the board or a black square. We call these lines *rays*. Then, we want to show:

Lemma 1. *Every square containing a question mark lies on a ray.*

Proof. Observe that some question marks can be seen to lie on a ray just by examining the gadget containing the question mark. Call the rest of the question marks *non-obvious*. Observe (by inspection) that in every gadget G which contains a non-obvious question mark, one can draw one or two lines which are 17 squares long and don't contain any black squares such that every non-obvious question mark lies on one of these lines. It can be seen (again by inspection) that every such line contains two non-obvious question marks which lie in the outermost rows or columns of the gadget, and thus also in the gadgets adjacent to G.

If the question marks can be directly observed to lie on a ray, then the question marks in G must lie on the same ray (observe that the rays point in the right directions for this to follow, i.e. they aren't completely contained within gadget overlap areas). If not, continue into the next neighbour. As argued earlier, this cannot stop by running out of neighbours or running off the board, so this must stop in a gadget containing an end square, with its ray fully containing these 17 square long lines, and thus the non-obvious question marks in G (at least on one of the lines; repeat this argument for the other line, and for all other gadgets).

As a consequence of this two-directional inductive argument, every ray contains at least (in fact exactly) two end squares. Extend the ray from one end square; it will eventually hit an end square. This means that in each of the four directions out from the end squares, we have found a square that is certainly black. If we assume that every square on the ray is white, then for every end square v and every value of $N(v)$ except one, rule 4 must be violated. Assign to $N(v)$ the remaining value, such that every square on a ray must be white.

Observe that every non-end square $v_?$ with a question mark is adjacent to a black square (in a non-ray direction). As this square is on a ray (by lemma 1), it is surrounded by two end squares and thus also two black squares in the ray

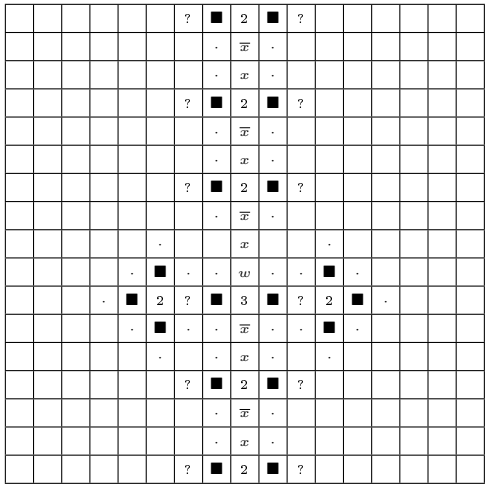


Figure 4: The Wire gadget

direction. There might be a black square in the last direction inside the gadget containing $v_?$. If there is, let $N(v_?)$ be the value such that the other squares in this last direction must be white. If there isn't, let $N(v_?)$ have the unique value such that $v_?$ must touch zero squares in this last direction.

3.4 Proof of Correctness of the Reduction

We have described how to produce a Kurodoko given a SAT instance. We want to show that the Kurodoko instance has a solution if and only if the SAT instance has a solution. To do this, we first make some observations about the set of solutions to the Kurodoko instance and the properties of its elements.

Consider again the Wire gadget (Figure 3). In every Kurodoko instance containing it that has a solution, the squares between 2 and ? must be black, or else rule 4 is violated. Their neighbouring squares will have to be white, or rule 2 is violated. Similar deductions can be made around the 2s adjacent to the ?s. This means that in any solution, the wire gadget must look like in Figure 4. The ■ represent squares that are black in all solutions, and · represent squares that are white in all solutions. Not all possible deductions are made—some squares are necessarily white because of the ray property, but these are not marked as such, as this information is not necessary to make the deductions we need. The square marked w must be white, or else the 3 would touch either too few or too many squares and violate rule 4. The squares marked x can be seen to all have the same value, likewise for $\overline{x}$, and the two values must be different. Otherwise, either rule 4 or rule 2 would be violated.

Also, the square north (in the board) of the northernmost 2 in the Wire gadget ought to be marked x, as it must be consistent with its value. Partial solutions for the remaining gadgets are to be found in Appendix B, along with

arguments that they work as described previously.

3.4.1 Proof That Kurodoko Solvability Implies SAT Solvability

We are now ready to prove the following:

Theorem 5. *Let a SAT instance be given, and let $K = (w, h, C, N)$ be a Kurodoko instance produced by the reduction described previously. If K has a solution, then the SAT instance also has a solution.*

Proof. As we just stated, if K has a solution, then the gadgets work as claimed. It is clear from the structure of the components that if the gadgets work as claimed, the components do as well.

Let a solution be given. The reductions specifies an obvious bijection between SAT variables and variable components in K: assign to each SAT variable the value of x in its component (i.e. v_1 is true iff x is black in the leftmost variable component).

Exactly one input to each clause gadget must be black (i.e. true), which by the structure of clause components implies that there is exactly one "good" input to the gadget's containing component, one that is either true (black) and non-negated or false (white) and negated.

The routing component matches the clause components with variable components in the same way clauses are matched with variables in the SAT instance. Since clause components each have one good input, the variable assignment ensures that each SAT clause have exactly one literal that is satisfied. That is, the assignment is a solution of the SAT instance.

3.4.2 Proof That SAT Solvability Implies Kurodoko Solvability

Next, we want to establish that if the SAT instance that produces a given Kurodoko instance K has a solution, then K has a solution as well. We will do this by suggesting a solution candidate and then showing that none of the four rules are violated.

Theorem 6. *Let a SAT instance be given, and let K be the Kurodoko instance produced by the reduction when run on the given SAT instance. If the SAT instance has a solution, then K has as solution as well.*

Proof. Let $v \in \{0,1\}^n$ be a variable assignment which satisfies the SAT constraints. For every live variable i in the SAT, there is one corresponding variable gadget in K; in each such gadget, let x be black if v_i is 1 and white if v_i is 0.

Also, let every square be black if it contains a ■ in its partial solution in Appendix B. Let a square be black if the consistency of x and $\overline{x}$ requires it, and let squares be black according to the description of the gadgets (i.e. if two inputs x and y to an Xor are black, let the squares corresponding to xy and $\overline{z}$ be black). For purposes of (opposite) consistency, consider the black square in the next-to-top row in the Zero gadget to be a named square (x, $\overline{x}$, y, etc.), and cross the gadget boundary: if an outermost x of a gadget is white, the square on

the other side of the 2 is black and vice versa. Finally, some squares $v_?$ marked ? are not end squares and aren't flanked by two intra-gadget black squares in the partial solutions in Appendix B. One of the two squares adjacent to $v_?$ in the non-ray direction is marked ■ in the partial solutions; let the other be black. Let the remaining squares be white.

First, rule 1 is clearly satisfied: one can see by inspection that no square in a gadget marked ■ nor any named square also contains a clue.

Secondly rule 2: no squares marked ■ are adjacent and no square marked ■ is adjacent to a named square. The set of squares adjacent to named (black) squares that could potentially be black are the negations of said named squares, which obviously don't pose a problem, and one square in Xor marked xy: it contains the value $x \wedge y$ and is adjacent to $z = x \oplus y$. Note that if $x \oplus y = x \wedge y$ then $x = y = 0$, so this doesn't violate rule 2 either. So there is no pair of adjacent black squares.

Connectedness is required by rule 3. If we can establish that every square is connected to the top left square(s) of the gadget(s) containing it, we can conclude the rule can't be violated, as due to symmetry and transitivity of the connectedness relation each square in a gadget is connected to every other square in that gadget. In particular, each square is connected to the edge squares, which are connected to every square in the neighbouring gadget. But then we can reach any target square, starting at any other square: go through neighbouring gadgets to the gadget containing the target square, and then (staying inside that gadget, going via the top left corner) go to the target square.

It can be seen by inspection[4] that every square in a gadget has such a path to the top left corner of the gadget, except for one class of square: a 2 in an outermost row or column, with three adjacent black squares—either three squares marked as black, in the case of Zero, or two such squares and one named square. In both cases, assuming this "trapped" square is on the north edge, the next square north of 2 is white by the oppositional consistency of $(x, \overline{x})$, having considered the "trapping" black square from Zero as a named square. If there was no such trapping black square, not only would the 2 have an intra-gadget path to the top left square, it would also have a five square path to its closest two ?s. When there is such a trapping black square, the 2 therefore has a five square path to its nearby ?s through the adjacent gadget. From there, it then has a path to the top left square of both its containing gadgets.

Lastly, rule 4 requires each clue to touch a number of (white) squares denominated by that clue. The clue squares can be divided into three sets, namely those marked ?, those which are flanked by a named square and its negation (possibly with some squares marked w adjacent on one side), and finally the rest, which (loosely speaking) serve to necessitate the flanking of the squares in the middle group by two black squares.

The last group can be seen to all obey rule 4 by inspecting the partial solutions in Appendix B, with one exception: the topmost 2 in Zero, but this

[4]The inspection may in some cases be easier if one cuts the gadgets into quadrants and see each quadrant to be connected, then realises that the dividing lines can be crossed by paths that only contain white squares.

clue is fulfilled since we treat the black square on the south as named with respect to oppositional consistency—in other words, as every Z has a gadget to the north of it, the square to the north of this 2 is white, and the square to the north of that is black.

It should be obvious from how the values of $N(v_?)$ are chosen in the reduction that the first group, the squares marked ?, also obey rule 4: the values are chosen such that if every square on a ray is white (which it is), the squares orthogonal to the ray going out from the end squares are white (which they are), and each $v_?$ that isn't "framed" by four necessarily black squares are given black neighbours to frame them (which they are), the clues are satisfied.

Lastly, the clues extended by w and flanked by named squares. These can all fairly straightforwardly be seen to be satisfied by the oppositional consistency of the named squares (and their black flanks). Two notable exceptions are the center squares of every Choice and Xor gadget, respectively. A simple case analysis shows that every choice of x, y and their implied values of xy and z will satisfy the central 3.

Lastly, since v is a solution to the SAT instance, each clause has one true literal. This implies by the structure and combination of the gadgets and components that every choice gadget has one black input. But then it is clear that exactly one of the central $\overline{x}$, $\overline{y}$, $\overline{z}$ are white, satisfying the central clue.

Thus, under the assumption that the given SAT instance has a solution, so does the Kurodoko instance K produced by the reduction.

3.5 Discussion of the Reduction and the Result

We have seen (with proof) a mapping from SAT instances to Kurodoko instance which preserves solvability. In fact, we have given a map from SAT solutions to Kurodoko solutions. Note, however, that for every SAT solution there are multiple Kurodoko solutions: every clause component contains gadget-free board positions. In such a "null gadget", one can freely choose the colour of the center square (and one in fact has many more degrees of freedom).

This means that the map from SAT solutions to Kurodoko solutions (given our choice of polynomial time witness checking turing machines) isn't injective. One might as future work try to find a reduction from other problems to Kurodoko where there is an injective solution map.

Also, the use of ? is somewhat unsatisfactory: this makes the atomic parts of the reduction depend on how they're combined. It would make for a simpler and more easily understood reduction if this requirement was eliminated.

However: note that each ? value, and in fact each other integer in the representation of the Kurodoko instance produced by the reduction, is at most linear in the size of the SAT instance (and also the Kurodoko instance). In other words, the Kurodoko solvability problem is in fact *strongly* **NP**-complete.

One can do better than linear, though. If one adds kinks to the wire subcomponents (connect four Bends so as to act the same) and adds layers of wire subcomponents between each sorting layer, each ? is on a ray of length $O(1)$. The details of proving this are left as an exercise to the reader.

4 Slither Link Variants are NP-Complete

In a Slither Link puzzle, the player must draw a cycle in a planar graph, such that the number of edges incident to a set of clue faces equals the set of given clue values. We show that for a number of commonly played graph classes, the Slither Link puzzle is **NP**-complete.

Slither Link is a popular puzzle, published both in book form [Nik11b], as online static games [Bum11] and as downloadable applications [Tat10]. In its most general form, the rules can be formalized as follows.

Let $G = (V, E)$ be a plane graph[5] and F_c a set of faces in G, the *clues*, and let $v\colon F_c \to \mathbb{N}$ be a function giving the *clue values*. The Slither Link decision problem is this: decide whether there is a set of edges $C \subseteq E$ such that C is a cycle and such that for all faces $F \in F_c$ the number of edges in C incident to F equals $v(F)$.

The original and most popular form of this puzzle is where V is a rectagular subset of $\mathbb{Z}^2$, e.g. $V = \{1, \ldots, w\} \times \{1, \ldots, h\}$, and E is the set of vertex pairs with Euclidean distance 1. This was shown to be **NP**-complete in [YS03]; a proof is readily available online in [Yat03]. We show that for a small handful of other graph classes $\mathcal{G}$, the Slither Link decision problem restricted to $\mathcal{G}$ is also **NP**-complete.

For two of the graph classes, the triangular and hexagonal graphs, the **NP**-completeness is already known [NU09]. In fact, the authors show the problem to be ASP-complete on those graphs: given a solution, it is **NP**-hard to find another solution. These solution and hardness concepts are studied in [Yat03]. One benefit of our reductions on those same graphs are their simplicity, which they achieve by relying on a result published after [NU09].

4.1 Membership of NP

First, we want to show that in the general form, i.e. for the class of all planar graphs (given as adjacency lists with clue faces and values given as edge lists and binary integers), the Slither Link decision problem is in **NP**. The argument is simple. Let C be given and consider the graph $G' = (V_C, C)$ where V_C is the set of endpoints of edges in C. It can easily be checked that each vertex v has degree 2 in G' and that G' is connected, and therefore a cycle. Checking that the number of edges incident to a face F of G equals $v(F)$ is likewise easy: compute a planar embedding in linear time [BM04]; for each given clue, check that the given edge list actually is the boundary of a face, and that C overlaps that face F at exactly $v(F)$ edges.

For more specific Slither Link decision problems, i.e. for narrower graph classes $\mathcal{G}$, their **NP**-membership can be seen by the above proof and that testing membership of $\mathcal{G}$ is easy, if this is indeed the case. We will typically represent members of $\mathcal{G}$ not by an adjacency list but by parameters (similar to how w and h characterise rectangular grid graphs) from which an adjacency list can

[5] A planar graph which not only can be but also *is* embedded in the plane.

efficiently be computed. Then membership of $\mathcal{G}$ is given by construction and the above proof still applies.

4.2 The General Approach to Proving Hardness

It is known that the Hamiltonian Cycle problem is **NP**-complete on hexagon grids [IMR+07]: that is, given a set of points P on a hexagonal grid with a set of edges induced by unit Euclidean distances, the problem of determining whether a cycle exists that goes through every point exactly once is **NP**-complete. This is true even if the largest Euclidean distance between two points is required to be polynomial in the number of vertices in the graph; this is readily apparent from the proof in the citation. Thus, we can take the coordinates of hex grid points to be given in unary.

We will reduce this problem (Unary Hex Grid Hamiltonian Cycle) to each of our concrete Slither Link variants by showing how to construct vertex and non-vertex gadgets corresponding to points in P and not in P, respectively, and how edge gadgets are formed when vertex gadgets are combined. Lastly, we'll prove that this construction actually captures the behavior of vertices and edges in a hex grid hamiltonian cycle problem, such that local solutions produce global solutions to Slither Link, which match hamiltonian cycle solutions.

Let us be specific about hexagon grid graphs: let $x = (1, 0) \in \mathbb{R}^2$ and $w = (\cos 120°, \sin 120°) \in \mathbb{R}^2$, and let $T = \{ax + bw \mid a, b \in \mathbb{Z}\}$ be the *triangle grid* or lattice formed by integer combinations. The reader may recognize this as the Eisenstein integers. A *triangle grid graph* is a graph where V is a finite subset of T and E is those vertex pairs with Euclidean distance 1. The *hexagon grid* is the set $T_h = \{ax + bw \in T \mid a + b \not\equiv 0 \pmod 3\}$. Note that (x, w) is a basis for $\mathbb{R}^2$ so (a, b) is uniquely determined for every point in T, and so any property of e.g. $a + b$ is a well-defined property of points in T. If one draws line segments between adjacent points in T_h, one will have drawn a hexagon tesselation of the plane. A *hexagon grid graph* is a graph induced by a finite set $V \subseteq T_h$ and unit distance edges.

Let $G_r = (T, E_1)$ be an infinite grid graph, e.g. T or T_h, and $G = (P, E)$ be a finite graph on this grid, i.e. $P \subseteq T$ and $E = (P \times P) \cap E_1$. Then we define $\text{Neigh}(P) := \{v \in T \mid \exists v' \in P \colon (v, v') \in E_1\}$ and $\text{Rim}(P) = \text{Neigh}(P) \setminus P$. Note that $P \cup \text{Rim}(P)$ has height and width only $O(1)$ larger than P, and can easily be computed.

The motivation for this definition is that if non-vertex gadgets are placed in a Slither Link graph, corresponding to points in $\text{Rim}(P)$, any candidate solution C which is partially contained in P will be unable to cross $\text{Rim}(P)$ and so will be completely confined to P: it can't be on the outside as long as vertex gadgets contain a clue face with a positive value, because then C wouldn't satisfy that clue.

4.3 The Dodecahedron Graph Class is Hard

The dodecahedron grid[6] looks like this: draw the hex grid and replace each vertex with an equilaretal triangle, turning each hexagon into a regular dodecahedron. Equivalently, draw a row of dodecahedrons next to one another with overlapping vertical edges; then, at each edge at five-o'-clock, draw a dodecahedron with the same edge in the eleven-o'-clock position. If the first row is w dodecahedrons wide and this row pair is repeated $\frac{h}{2}$ times vertically translated (plus a copy of the top row if h is odd), this is the *dodecahedron grid graph.*[7] See also Figure 5. They are represented by a pair (w, h) in unary, plus a clue/value list.

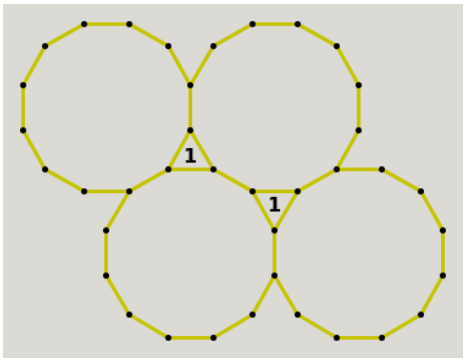


Figure 5: 2 by 2 dodecahedron grid, showing two vertex gadgets

The Dodecahedronal Slither Link decision problem is the Slither Link decision problem, limited to this class of graphs. Now, let us consider a reduction from Unary Hex Grid Hamiltonian Cycle to Dodecahedronal Slither Link.

Let a Unary Hex Grid Hamiltonian Cycle problem $G = (P, E)$ be given. Note that we can biject the vertices in P onto triangular faces F_s on the dodecahedronal grid such that $(s, t) \in E$ if and only if F_s and F_t have an edge between them (i.e. between a corner in F_s and one in F_t). Let $v(F_s) = 1$ for $s \in P$ and $v(F_s) = 0$ for $s \in \mathrm{Rim}(P)$, and undefined elsewhere (i.e. the set of clues is $P \cup \mathrm{Rim}(P)$). With the hex grid coordinates in P given in unary, we can output w and h within polynomial time if we just choose them no larger than necessary to contain all the faces in this bijection, which is easy to do. If G has only a single vertex, we don't do the above. Instead, we output a particular yes-instance, the $w = h = 1$ puzzle with a single face, containing the clue 12.

The triangular faces with clue value 1 are the vertex gadgets; those with clue value 0 are the non-vertex gadgets, and their connecting edges are edge gadgets.

[6] More commonly known as the truncated hexagonal tiling; we hope the reader will accept our idiosyncratic terminology.

[7] We focus on tesselations of the plane by dodecahedrons and triangles in this particular grid structure, because this is the kind of graph produced by at least one popular Slither Link implementation.

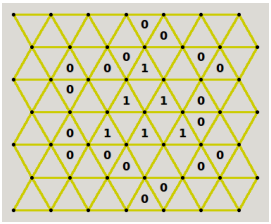


Figure 6: Triangular vertex gadget

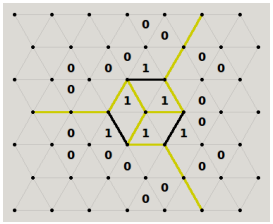


Figure 7: Partially solved vertex gadget

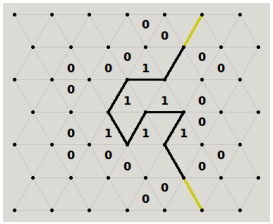


Figure 8: Completely solved vertex gadget

Theorem 7. *The Dodecahedronal Slither Link decision problem is **NP**-complete.*

Proof. The problem is clearly in **NP**—checking dodecahedron class membership is done by construction, so the previous proof applies. Thus, we only need to show that the reduction works.

If G has only a single vertex, it has a hamiltonian cycle and clearly the reduction outputs a yes-instance, so assume that G has multiple vertices.

The local solution to a vertex gadget is that C contains one edge of the triangular face and two edge gadgets adjacent to that face. Each face has three solutions, rotationally symmetric, corresponding to each choice of edge pairs.

If there is a hamiltonian cycle H in G, for every edge $e \in H$, include its corresponding edge gadget in C. This is consistent with exactly one local solution to every vertex gadget. Include the edges from the local solutions to vertex gadgets in C as well, and no more edges. Then C is a cycle because H is, and C satisfies all clue value constraints by construction.

On the other hand, any solution C will have to go through the gadget of every vertex in P (or it would violate a clue), and each vertex gadget will have two adjacent edge gadgets contained in C. This corresponds exactly to a hamiltonian cycle in G.

The result of the reduction is polynomial in size—the input has unary coordinates—and is easily (polynomial time) computable.

4.4 The Triangular Graph Class is Hard

Recall the above definition of triangular grid graphs. We want to focus on a subset of these which has a very regular row/column structure. A row of length w is $2w$ triangles, alternating between pointing down and up, overlapping in the non-horizontal edges. In a graph with h rows, $h > 1$, each row is the horizontal mirror image of its predecessor. See Figure 6 for a $w = h = 6$ graph. We represent these graphs by unary encodings of w and h, as before, plus clue faces and values. We want to reduce Unary Hex Grid Hamiltonian Cycle to Triangular Slither Link.

Vertex gadgets look as in Figure 6, or as in Figure 7 when partially solved—light grey edges are not a part of any solution, black edges are part of all solutions, and yellow edges are ambiguous.

Note that the displayed gadget corresponds to a hex grid vertex with edges going west, northeast and southeast. Mirror images of this gadget do the opposite. Non-vertex gadgets are as vertex gadgets except with all 1-valued clues replaced by 0-valued clues. Vertex gadgets are joined together as in Figure 9. Note how the partial (and full) solutions are mirror images of one another.

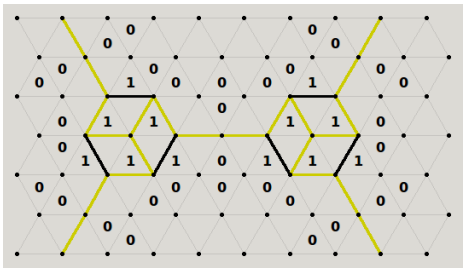


Figure 9: Combined triangular vertex gadgets

To reduce a Unary Hex Grid Hamiltonian Cycle instance $G = (P, E)$ to Triangular Slither Link, put a vertex gadget in a triangle grid for each $v \in P$ and a non-vertex gadget for each $u \in \mathrm{Rim}(P)$, joined as shown. Put these with as small coordinates as possible, so as to minimise w and h.

Theorem 8. *The Triangular Slither Link decision problem is **NP**-complete.*

Proof. We can establish **NP**-membership by the above observation: checking the graph in adjacency list form is easy, and it's easy to create that representation. Any pair of (w, h) is a member of our special class of triangle grid graphs.

Clearly the reduction is polynomial time: the gadgets are small and easily manipulable, and $w + h$ is small even in unary.

The local solution to a vertex gadget is shown in Figure 8; this solution has two rotationally symmetric variants not shown, and each of the three corresponds to a choice of two selected edges incident to the gadget's corresponding vertex. There is also a fourth solution, a closed hexagonal loop, which corresponds to no edges being chosen. The local solution to a non-vertex gadget is one with no edges in the cycle.

If $|P| = 1$, then G has a trivial hamiltonian cycle, and the closed loop local solution is also a global solution.

If $|P| > 1$ then the fourth solution can't occur as part of any (global) solution: let C_4 be the edges in the hexagonal loop of a particular vertex v. Then any superset of C_4 can't be a cycle, and C_4 fails to satisfy the clues in the gadgets of $v' \in P$, $v' \neq v$.

Let a cycle H in G be given. Construct a solution C to the Triangular Slither Link instance by choosing local solutions to vertex gadgets corresponding to the choice of incident edges in H. This violates no clue, and the local solutions are consistent in their overlap (i.e. no edge is both a member and non-member of C)

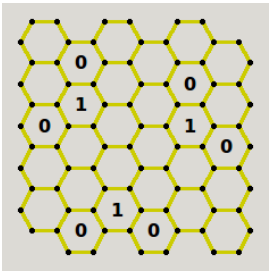


Figure 10: Hexagonal vertex gadget

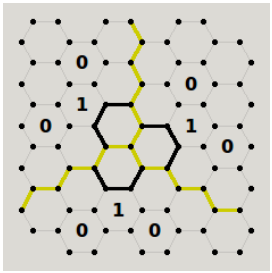


Figure 11: Partially solved vertex gadget

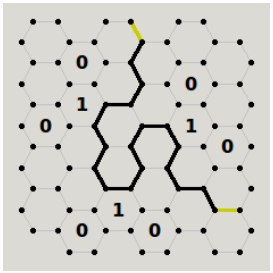


Figure 12: Completely solved vertex gadget

by construction. Globally, C is guaranteed to be a cycle (and not a multi-cycle cover) because H is a cycle.

Now, let a solution C to the Triangular Slither Link instance be given. Since the vertex gadgets only have local solutions corresponding to consistent choices of edges, and these edges globally form a cycle, we can produce a cycle H in G.

4.5 The Hexagonal Graph Class is Hard

Recall the definition of the hexagonal grid. Again we look at a particular shape: we have w columns of h hexagons, each column's starting hexagon alternatingly to the southeast or northeast of its left hand neighbour. See Figure 11 for an example with $w = 6$ and $h = 5$. We want to show that the Hexagonal Slither Link problem is **NP**-complete.

The vertex gadgets look like in Figure 10 and its counterpart in Figure 13. The non-vertex gadgets are the same, except with all clue values replaced by 0. The gadgets join as in Figure 13. The reduction is as previous: put vertex gadgets at P-members and non-vertex gadgets at $\mathrm{Rim}(P)$, making w and h as small as possible. Note though, that we have to rotate our input graph 90°, essentially flipping the two axis, as our gadgets have edge gadgets going north/south rather than east/west.

Theorem 9. *The Hexagonal Slither Link decision problem is **NP**-complete.*

Proof. Just like previously, the gadgets are small and easily manipulable, so the reduction is polynomial time.

The local solution is as in Figure 12, with three rotations in total, allowing each choice of two out of three edges, and a fourth solution—a loop around a cluster of three hexagons—which is only valid for one-vertex graphs, in which case it's again a global solution. And again, the local non-vertex solution allows no edges. Each (global) solution is consistent with a hamiltonian cycle of the given Unary Hex Grid Hamiltonian Cycle problem, and corresponding to each hamiltonian cycle is a solution to the Slither Link problem.

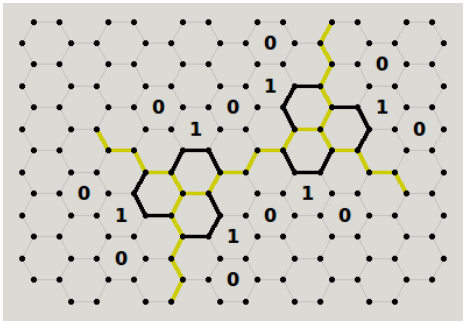


Figure 13: Combined hexagonal vertex gadgets

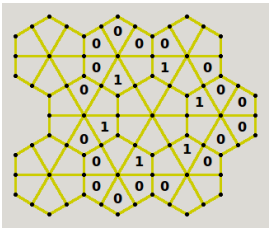


Figure 14: Dual grid vertex gadget

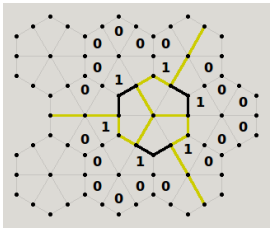


Figure 15: Partially solved vertex gadget

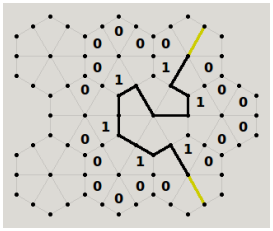


Figure 16: Completely solved vertex gadget

4.6 The Dual Graph Class is Hard

The triangular and hexagonal grids are each others' planar duals. The dual graph class[8] arises from superimposing them on one another; that is, by adding vertices in the center of hexagons and connecting them to the centers of neighbouring hexagons, also adding a vertex when the connecting line intersects the edge between the connected hexagons. If we scale the unit down by 2, the hexagon centers have edges to all vertices in distance 2, while all other vertices have only unit distance edges. The particular structure we look at is a grid of h rows, each containing w hexagons—similar to the hexagonal grid, but rotated 90°.

The vertex gadget looks as in Figure 14.[9] Non-vertex gadgets are vertex gadgets with all clues replaced by 0. Once again, put vertex gadgets at P and non-vertex gadgets at $\mathrm{Rim}(P)$, packed together to make w and h as small as possible; the combination looks as in Figure 17.

Theorem 10. *The Dual Slither Link decision problem is* ***NP****-complete.*

[8] More commonly known as the deltoidal trihexagonal tiling; again we hope the reader will accept our idiosyncracy.

[9] The two leftmost empty hexagons are not part of the gadget proper, but an artifact of the software used to generate the illustration.

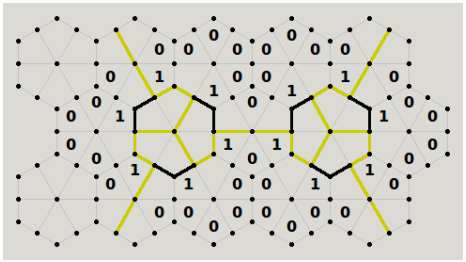


Figure 17: Combined dual vertex gadgets

Proof. Like above: the local solutions combine into a global solution, which matches hamiltonian cycles. Again, the single-vertex case has a special fourth local solution to vertex gadgets which is a hexagonal loop. This local solution is forbidden in global solutions to problems which the reduction yields on multi-vertex graphs.

4.7 Review and Discussion

We have shown how four Slither Link variants on particular graph classes are **NP**-complete. Note also that the reductions are all parsimonious—the number of solutions is preserved. This implies that the Slither Link variants are not only #**P**-complete if Unary Hex Grid Hamiltonian Cycle is, but also ASP-complete. To establish that Unary Hex Grid Hamiltonian Cycle is ASP-complete, one could try to find a chain of parsimonious reductions from one of the ASP-complete problems given in [Yat03], for instance a SAT variety. The chain of known reductions showing the **NP**-completeness of Unary Hex Grid Hamiltonian Cycle might be parsimonious at all steps.

If one defines the Hamiltonian Cycle problem such that single-vertex graphs are no-instances, one can modify the presented reductions so they test for this and output a no-instance (for instance, give a face a positive clue value and all its neighbours the clue value 0).

Clearly Slither Link isn't difficult for all graph classes—it's rather easy to determine if G has a clue-satisfying cycle if G is a cycle. If one wants a deeper understanding of the hardness of Slither Link than we provide, one might try to find graph classes with non-obvious hardness characteristics.

5 Secret Sharing and Secure Computing From Monotone Formulae

We present a new construction of log-depth formulae for the majority function based on 2-out-of-3 threshold gates. 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. Moreover, we obtain strongly multiplicative schemes from a variant of this construction based 2-out-of-4 threshold gates. 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 actively secure and efficient, a long standing open problem. Finally, we also show some positive results on local conversion of shares in our scheme to a class of other schemes and a negative result on usage of our scheme for pseudorandom secret sharing as defined by Cramer, Damgård and Ishai.

5.1 Overview

5.1.1 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 [Val84] using a probabilistic method. In this paper we show that the probabilistic approach can also be used to get log-depth formulae for the 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. This significantly simplifies Valiant's argument, and at the same time a slight extension of the argument shows that one can efficiently construct such a formula using a procedure that succeeds with overwhelming probability.

These new formulae can be used to construct new secret sharing schemes and protocols, as we explain below. We stress that 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 [HM00], however that result leads to exponential size formulae.

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. It is therefore important that our method for constructing formulae extends to other basic gates than $T(2,3)$.

5.1.2 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[BL90], 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 [CDI05]). These are not efficient, however.

5.1.3 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 [HM00]. 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 [HM00], 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 [Bar89].

The black-box computing problem was considered in [DPS+11], where a passively secure protocol was obtained, secure against corruption of $t < n/2$ parties. The construction of the general protocol is higly 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. We only need to extend slightly the three-player solution from [DPS+11] to make it fit into the construction.

At the same time, this observation opens the door to obtaining a protocol with active security, which was a main open problem in [DPS+11]: we build a a protocol 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 protocol for 4 players that is secure against 1 actively corrupted player, but so far we were not able to do this. On the other hand, we stress that the problem was open for any constant fraction of corrupt players.

5.1.4 A Negative Result for Share Conversion.

In the final part of the paper we consider the notion of pseudorandom secret sharing (PRSS) introduced in [CDI05], 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 [CDI05] 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 [CDI05]. 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 [CDI05] to cover any secret-sharing scheme.

5.2 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 2. *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 now the function $f(x) = 3x^2 - 2x^3$, and consider 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

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

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.

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

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 11. *$F(\mathcal{S}_{2/3})$ 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 $\alpha_{i,j} = 1$.

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 12. *$F(\mathcal{S}_{2/3})$ 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

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

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 [BL90], 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.

5.4 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 13. *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 14. *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.[10]

Thus, we can write rs as a linear combination of linear combinations of shares held by the players in R. 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 3. *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 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

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

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, $x_i < c$.[11] 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 $p_{j+k} = p^{(j+k)}(p_0) < 2^{-k}c$.

[11] 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 $0 \leq x \leq t_0$.

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.

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 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.

5.5 Black-Box Computing over Finite Groups

5.5.1 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 [DPS+11] 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 [HM00]. 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 [DPS+11] 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 18 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 [DPS+11].

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 18, 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 $x \cdot y$.

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 [DPS+11], 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 [DPS+11] refers to as *compatibility*. These three properties are the ones from which the existence

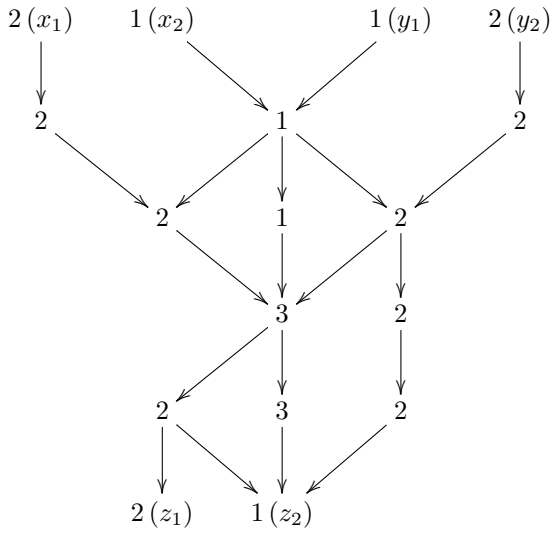


Figure 18: Multiplication protocol (left output)

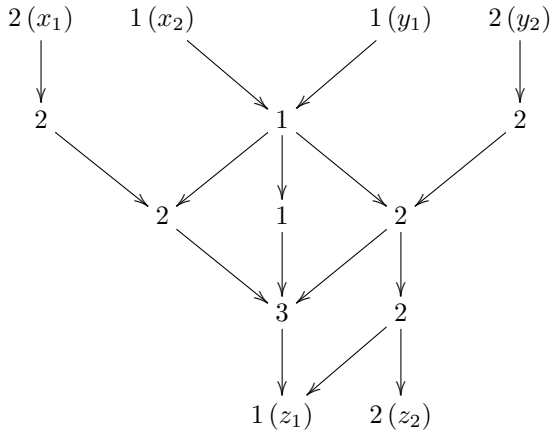


Figure 19: Multiplication protocol (right output)

of a simulator ultimately follows.

5.5.2 Construction of a Protocol for n Players

Following [HM00], 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[12] 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.

[12] Depicted in Fig. 19

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 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 $P_0.\mathsf{multiply}$ $m-1$ 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:

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

$P_i.\mathsf{Input}(Y, y)$

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 $P_j.\mathsf{Input}(P_j.Y, y_2)$ and $P_k.\mathsf{Input}(P_j.Y, y_1)$.

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 $P_i.\mathsf{Send}(P_u, P_i.X, P_u.Y)$ 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.

$P_i.\mathsf{Send}(P_u, P_i.X, P_u.Y)$

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:
   $P_j.\mathsf{Sample}(X_{j,1})$, $P_j.\mathsf{Invert}(X_{j,1}, X_{j,1}^{-1})$, $P_j.\mathsf{Multiply}(X_{j,1}^{-1}, X, X_{j,2})$.

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

   (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: $P_k.\mathsf{Send}(P_v, P_k.X_{k,2}, P_v.X_k)$ followed by $P_k.\mathsf{Send}(P_w, P_k.X_{k,1}, P_w.X_k)$.

   (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 do $P_0.\mathsf{Multiply}(T_{j-1}, H_j, T_j)$ for all $j = 3, \ldots, m$.

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 15. *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.

5.6 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.

5.6.1 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.

5.6.2 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.

5.7 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 [CDI05]. 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 16. *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.

5.7.1 A More General Lower Bound

In [CDI05], 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 [CDI05], 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 [CDI05], 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 [CDI05] 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 [CDI05]. However, we are about to look at negative results which are only made stronger this way. The following result is shown in [CDI05]:

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 17. *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.

6 Multiparty computation and I/O efficiency

We consider a setting where a set of n players use a set of m servers to store a large, private data set. Later the players decide on one or more functions they want to compute on the data without the servers needing to know which

computation is done, while the computation should be secure against a malicious adversary corrupting a constant fraction of the players and servers. Using packed secret sharing, the data can be stored in a compact way but will only be accessible in a block-wise fashion. We explore the possibility of using I/O-efficient algorithms to nevertheless compute on the data as efficiently as if random access was possible. We show that for sorting, priority queues and data mining, can indeed be done. Even if a constant fraction of servers and players are malicious, we can keep the complexity of our protocols within a constant factor of the passively secure solution. As a technical contribution towards this goal, we develop techniques for generating values of form r, g^r for random secret-shared $r \in \mathbb{Z}_q$ and g^r in a group of order q. This costs a constant number of exponentiation per player per value generated, even if less than $n/3$ players are malicious. This can be used for efficient distributed computing of Schnorr signatures. Specifically, we further develop the technique so we can sign secret data in a distributed fashion at essentially the same cost.

6.1 Overview

In this section, we consider a setting where a set of n players $P_1, \ldots, P_n$ with limited memory use m remote servers $D_1, \ldots, D_m$ to store a large data set securely, and later wish to do secure computation on these data.

As a motivating example, think of a set of authorities in the public sector, each of which initially possesses a database with personal information on various citizens. Suppose they wish to compute results requiring access to all databases, to gain some administrative advantage, for instance, or to do data mining. Allowing a single entity access to everything raises some obvious privacy concerns and is in fact forbidden by law in several countries. A standard solution is to store all data in secret shared form and compute results by multiparty computation. But this would require every player to store an amount of data corresponding to all the original databases put together. It may be a more economic and flexible solution to buy storage "in the cloud", i.e., involve some remote servers whose main role is to store the data, and thus we arrive at the situation described above.

An important property of the model is that we do not assume that the computations we want on the database are given in advance. Instead players will decide dynamically which computations to do as a result of input received from the environment – indeed this seems to us to be a more realistic model. This is the main reason why we want the servers to act as storage devices only, rather than have them do computation for us. If the servers had to be involved in the computation, we would have to pay for the communication needed to specify each new computation to all servers. In practice we may also face the problem of installing new software on the servers. In our model, all a server has to do is to supply storage and do a fixed and simple computation on demand, namely whatever is required to read or write a block of data.

We stress that although cloud computing is one motivation for our work, our results can also be applied in a case where no servers are physically present.

Namely the players can choose to play the role of the servers themselves. Even in this case, our result offers a way to save memory at little or no extra cost to do the computation. We give more details on this below.

We will assume that our data can be represented as N elements in some finite field, and the computation is specified as a program $\mathcal{P}$ using arithmetic operations in the field, comparison and branching on public data. In addition, the program may read from or write to a "disk", modeling the storage supplied by the servers. Of course, we cannot allow the servers to see any data in cleartext, and since we want to do secure computation on the data later, the computationally most efficient approach is to use secret sharing[13]. A first naive approach is to use standard Shamir secret-sharing [Sha79]; this will require $\Omega(Nm)$ space. A well-known improvement is to use packed secret sharing, suggested by Franklin and Yung [FY92]. This is a variant of Shamir's scheme that stores data in large blocks (of size $\Theta(m)$ in our case). This will allow us to tolerate a constant fraction of the servers being corrupted, while requiring only storage $O(N)$.[14] However, this creates a new problem: the usage of packed secret-sharing format means that we can only read or write a large block of data at a time. This means we will be wasting resources if we only use a small part of each block we access.

Our first observation in this paper is that we can handle this issue by making use of so called I/O-efficient algorithms [AV88]. In the I/O model, we assume we have a fast computing device (a CPU) with small internal memory, that connects to a large, but much slower memory such as a disk, which only allows you to read or write data in blocks of a given length. The goal of an I/O-efficient algorithm is to solve some computational problem while minimizing the number of times blocks have to be transferred between CPU and external memory. There is a large body of research in this area, and the I/O-complexity of several basic computational problems is known.

Now, if we think of the n players as the CPU and the m servers as the memory device, it is clear that there should be hope that if the players follow an I/O-efficient algorithm, we can minimize the number of transfers and in this way use only $O(N)$ storage while still being able to perform the computation essentially as efficiently as if random access was possible – thus enabling us to use any available technique for optimizing the computation. This idea of combining MPC and I/O-efficient algorithms (which to the best of our knowledge, we are the first to explore) is not completely trivial to capitalize on, however: we need protocols for reading from and writing to the servers' memory. This involves converting between secret sharing among the servers and secret sharing among the players, and must be efficient enough to not dominate the rest of the computation. The program must also be oblivious in the sense that its memory

[13] Using fully homomorphic encryption is an option as well, but would incur a huge computational overhead with current state of the art

[14] Packed secret sharing is an instance of the more general concept of a Ramp Scheme[BM85], in which a secret is shared such that sets of at least t_0 players can learn nothing, sets of at least t_1 players can reconstruct the secret, and sets in between can learn partial information about the secret.

access pattern should only depend on the data known to the adversary[15]. This, however, is an issue in all known multiparty computation protocols.

In this paper, we give efficient protocols for reading from and writing to the memory formed by the servers, tolerating a constant fraction of malicious players and servers. We build from this a "compiler" that transforms any program that is oblivious (in the sense explained above) into a protocol the runs the program securely in our model. We furthermore show that if the program is also I/O-efficient, then the overhead from the format conversion is insignificant. Here, I/O efficiency means that, up to logarithmic factors, the program needs $O(N/\ell)$ I/O operations, where ℓ is the size of blocks that are transferred. We show that our method applies to a rich class of computational problems by giving oblivious I/O-efficient programs for sorting, priority queues and some forms of datamining.

The table below shows a comparison between running an oblivious and I/O efficient program $\mathcal{P}$ using our approach and "Servers" which is the naive method where we use standard individual Shamir sharing of all N values on the servers and standard protocols for reading and writing. Here, storage and communication is the total number of field elements while computation is the total number of field operations. $Comp_{\mathcal{P}}(N)$ is the total number of operations done by $\mathcal{P}$ on input size N while $Comm_{\mathcal{P}}(N)$ counts only operations that require communication for a secure implementation, such as multiplication and comparison. We have simplified the table by ignoring logarithmic factors; all details can be found in Section 6.5.

	storage	communication	computation
Servers	$O(mN)$	$\Omega(nmN) + Comm_{\mathcal{P}}(N)n$	$\Omega(nmN) + Comp_{\mathcal{P}}(N)n$
No servers	$O(nN)$	$Comm_{\mathcal{P}}(N)n$	$Comp_{\mathcal{P}}(N)n$
Our results	$O(N)$	$O(nN) + Comm_{\mathcal{P}}(N)n$	$O(nmN) + Comp_{\mathcal{P}}(N)n$

We see that our technique is in all respects as efficient or better than the (standard shamir with) "Servers" approach. The "No servers" is a scenario where the players store all the data themselves using standard Shamir sharing and protocols. In this case, the comparison to our results shows that even if players have massive amounts of memory themselves, our approach still offers a trade-off between memory usage and computational work: as soon as $Comm_{\mathcal{P}}(N)$ is $\Omega(N)$, our result offers smaller storage, similar or smaller communication complexity, and perhaps a factor m larger computational complexity. We say perhaps because this factor may or may not be significant, depending on how large $Comp_{\mathcal{P}}(N)$ is. For instance, the players could choose to play the role of the servers themselves (so that $m = n$). Then, if $Comp_{\mathcal{P}}(N)$ dominates nN, our results allow saving a factor n memory at no essential extra computation or communication cost.

To achieve our results, we develop some techniques of independent interest: we show how a set of players can compute several values of form $g^r \in G$, where

[15]Otherwise the construction would be insecure: in a protocol based on secret-sharing, players must agree on which shared values are being used at any given time, so we cannot hide the access pattern.

G is a group of prime order, and r is random and secret shared among the players. The cost of this is a constant number of exponentiations per player per instance produced, even if less than $n/3$ players are malicious. To the best of our knowledge, the most efficient solution following from previous work would be for each player to do a Feldman-style VSS [Fel87] of a random value and then combine these shared values to a single one. This would require $\Omega(n)$ exponentiations per player. Our technique can be used, for instance, to do threshold Schnorr signatures very efficiently in an off-line/on-line fashion: by preparing many g^r-values in preprocessing (or in idle time), a signature can be prepared using only cheap linear operations once the message becomes known. While this on-line/off-line property of Schnorr signatures is well-known, it is new that the *total* computational cost per player in a distributed implementation can be essentially the same as what a single player would need to prepare a signature locally. We extend this technique so that the message to be signed can be secret-shared among the players and remains secret, at a constant factor extra cost.

On the technical side, we start from techniques for verifiable secret sharing based on hyperinvertible matrices from [BTH08]. While in [BTH08] information theoretically secure protocols were considered, in our case the secret r is only computationally protected as g^r becomes public. This requires some modification to the protocol but more importantly, some non-trivial issues must be handled in the security proof. We show that our protocol implements an ideal functionality that chooses a random r, secret shares it and publishes g^r. This fixes the value of r, yet the simulator must make a view that is consistent with g^r without knowing r. Fortunately, the homomorphic properties of the function $r \mapsto g^r$ allows us to solve the problem. Finally, we are concerned with the computational efficiency, and a naive extension of the VSS from [BTH08] would cost a factor n more exponentiations than we can afford. To solve this, we devise a technique that allows players to distribute the exponentiations required among themselves while still being able to verify the work of each player efficiently.

6.2 Preliminaries

As in classical MPC, we have n players $P_1, \ldots, P_n$ who want to compute the value of an arithmetic circuit C over a finite field $\mathbb{F}_q \cong \mathbb{Z}_q$ for a prime q. At most t_p players may be statically corrupted by the adversary, who may be either active or passive. We also have m disk servers $D_1, \ldots, D_m$. They take part in the protocol, but their role is to store data rather than to compute on it. We assume the adversary can adaptively corrupt at most t_s of these. We assume synchronous communication and secure point to point channels between any pair of players. We will use the UC model for formalizing security of protocols.

Recall that Shamir's secret sharing scheme, when sharing one secret among n players, where t of them may be corrupt, uses a polynomial f of degree at most t, and that given $t+1$ points we can reconstruct the polynomial and recompute the secret value $f(0)$. We can think of this as "spending" t points on corrupt players and a single point on storing secrets. If we instead use more points to

store secrets and fewer points to handle corrupt players, we can store our data more densely, as observed by Franklin and Yung [FY92].

Definition 3. *A block is a vector of secrets* $(x_1, \ldots, x_\ell)$ *over the field* $\mathbb{F}$. *A block sharing among* m *servers with threshold* t_s *is a vector of shares* $(f(1), \ldots, f(m))$, *where* f *is a random polynomial of degree at most* $d = t_s + \ell - 1 < m$, *such that* $f(-k) = x_k$ *for* $k = 1, \ldots, \ell$.

Given a block $(x_1, \ldots, x_\ell)$ of known values it is easy to generate a suitable polynomial f for a block sharing: let $f(-k) = x_k$ for $k = 1, \ldots, \ell$. Next, choose random values for $f(i)$, for $i = 1, \ldots, t_s + 1$. Use Lagrange interpolation to compute the coefficients of f. Then f can easily be evaluated in $1, \ldots, m$. It is easy to see that any t_s shares reveal no information on the secret block, while any $d + 1$ shares allow reconstruction of the entire block. Of course there's nothing special about $-1, \ldots, -\ell$ or $1, \ldots, m$, except the two sets are disjoint and notationally conventient. In the following, we will choose both t_s and ℓ to be linear in m; exact choices will depend, e.g., on whether adversary is passive or not, as detailed later.

In the following, $[x]_f$ will denote a Shamir sharing of value x among the n players using polynomial f, of degree at most t_p. Likewise, for $\mathbf{x} = (x_1, \ldots, x_\ell)$, we let $[[\mathbf{x}]]_g = [[(x_1, \ldots, x_\ell)]]_g$ denote a block sharing among the m servers using polynomial g, of degree at most d as defined above. Depending on the context, we will sometimes omit f and g from the notation. Both types of sharings are linear; in particular, for blocks $\mathbf{x}, \mathbf{y}$ we have $[[\mathbf{x}]]_f + [[\mathbf{y}]]_g = [[\mathbf{x} + \mathbf{y}]]_{f+g}$ where addition of blocks and vectors of shares is done component-wise.

A word on Broadcast and Byzantine Agreement. When players and servers may be actively corrupted, we sometimes need broadcast among the players for a string of length $\Theta(m)$ field elements. In [FH06], it is shown how to do this with communication complexity $mn + poly(n)$. Since our protocols communicate at least $O(nm)$ field elements anyway, the cost of this will not dominate if m is sufficiently large compared to n (actually it will always be sufficient if m is cubic in n). However, we emphasize that in practice it will be a much better solution to use protocols without the $poly(n)$ overhead that do not guarantee termination (see [GL05]), and take some out-of-band action if the protocol blocks. In this case there is no demand on how m compares to n.

6.3 The Main Functionality

We will use the UC framework of Canetti [Can00] to argue security of our protocols. In this framework, one defines what a protocol is supposed to do by specifying an ideal functionality, and then shows that using the protocol is equivalent to using the functionality. The functionality $\mathcal{F}$ that we implement is basically a black-box computer for doing secure arithmetic in $\mathbb{F}$. In addition to these operations, it has a "main memory" and a "disk", and commands for writing to and reading from the disk. A command is executed on request by all honest players.

Functionality $\mathcal{F}$

input(i, v) Get a value x from P_i and store it in main memory at address v.

open(v) Send value stored at location v in main memory to all players and the adversary.

operation$(\diamond, v_1, v_2, v_3)$ Here, $\diamond$ can be $+, -, *$ or $\leq$. Let $val(v_1), val(v_2)$ be the values stored at locations v_1, v_2 in main memory. Compute $val(v_1) \diamond val(v_2)$ (0 or 1 in case of $\leq$), and store the result in location v_3.

const(v, x) Store the constant $x \in \mathbb{Z}_q$ at memory location v.

random(v) Sample a uniformly random $r \in \mathbb{Z}_q$ and store at the memory location v.

write$(addrs, blockaddr)$ Here, *addrs* is a tuple of ℓ distinct addresses in main memory. Write the values stored there as a block on disk, at location *blockaddr*.

read$(addrs, blockaddr)$ As above, *addrs* is a tuple of ℓ distinct addresses in main memory, Read the block from disk at location *blockaddr* and stores the ℓ values obtained in the locations specified in *addrs*.

Figure 20: The Functionality we implement

6.4 The Protocols

Standard techniques can be used to implement the input, open and arithmetic (including *const* and *random*) commands of $\mathcal{F}$, so we focus on the read and write commands. These commands reflect that we want the players to store blocks of data on the servers and be able to read the blocks back. As a warm-up we first do this assuming passive corruption, and then show methods for handling malicious players and servers. The write protocol converts ℓ secrets, shared independently, into a block sharing which is given to the servers. The read protocol converts the block sharing back into ℓ separate sharings. For a passive adversary, this is relatively straightforward. The idea is related to the results from [CDH07], except that here we convert between shares held by separate sets of players.

6.4.1 Passively Secure Implementation of $\mathcal{F}$

We can use the standard technique from [BOGW88] to implement the input, output, addition and multiplication commands. This simply amounts to representing a value x stored in main memory by $\mathcal{F}$ as a secret sharing $[x]$. Given this, one can build a (constant round) implementation of the comparison [DFK+06].

Hence, to implement the write command, we may assume that the players $P_1, \ldots, P_n$ hold $[x_1], \ldots, [x_\ell]$, that is, sharings of secrets x_1 through x_ℓ to be written. We then implement write by converting this to a block sharing held by the servers. The following lemma is useful

Lemma 4. *For $i = 1, \ldots, m$ and $k = 1, \ldots, d+1$, there exist $\lambda_k^i \in \mathbb{F}$, such that the following holds: For any $x_1, \ldots, x_\ell, r_1, \ldots, r_{d-\ell+1} \in \mathbb{F}$, let f be the*

polynomial with $f(-i) = x_i, f(j) = r_j$. In other words, f defines a block-sharing $[[(x_1, \ldots, x_\ell)]]_f$. Then each share in this block sharing can be computed as a linear combination of the x_i's and r_j's. More precisely,

$$f(i) = \sum_{k=1}^{\ell} \lambda_k^i x_k + \sum_{k=\ell+1}^{d+1} \lambda_k^i r_{k-\ell}$$

Proof. If we set $f(-i) = x_i$ and $f(j) = r_j$ (for all sensible i and j), this uniquely defines a polynomial f of degree at most d. This can be computed through Lagrange interpolation, which is a linear computation. More precisely, Let $y = (x_1, \ldots, x_\ell, r_1, \ldots, r_{d-\ell+1})^\top$. Writing $f(x) = c_0 x^0 + \ldots + c_d x^d$, let $c = (c_0, \ldots, c_d)^\top$, and let V be a $(d+1) \times (d+1)$ Vandermonde matrix over the points $-\ell, \ldots, -1, 1, \ldots, d-\ell+1$. Then $V \cdot c = y$ and thus $V^{-1} \cdot y = c$. We can then compute $f(x)$ as $(x^0, \ldots, x^d) \cdot c$. Since each entry in c is a linear combination of entries in y, so is $(x^0, \ldots, x^d) \cdot c$ and this defines the λ_k^i's we promised.

In the protocol we assumes that players can generate shares of random values unknown to the adversary. Several techniques exist for this, and for now, we simply assume access to a functionality F_{Rand}, defined below, and discuss later how to implement it.

Functionality F_{Rand}

Share(*num*) For $i = 1, \ldots, num$, choose random $s_i \in \mathbb{F}$, form $[s_i]_{f_i}$, where f_i is random subject to $f_i(0) = s_i$, $deg(f_i) \leq t_p$, and in each point owned by a corrupt player, f_i evaluates to a share chosen by the adversary[a]. Finally, send shares of all *num* values to all honest players.

[a] We need to allow the adversary to choose shares for corrupted players, in order to be able to implement the functionality.

Figure 21: The functionality delivering randomness

The idea behind the definition of F_{Rand} is that we can use known techniques to implement it for large values of the *num* parameter, with low amortized cost per sharing generated.

Next, let us consider a passively secure protocol for reading a block. Each server i has a share from the sharing of the block to be read, $[(x_1, \ldots, x_\ell)]$. We implement read by converting this to individual sharings held by the players, $[x_1], \ldots, [x_\ell]$. For this, we use the following lemma which is easy to show in a way similar to Lemma 4:

Lemma 5. *There exist constants δ_j^i such that for any block sharing $[x_1, \ldots, x_\ell]_f$, we have $x_i = \sum_{j=1}^{m} \delta_j^i f(j)$.*

Protocol $Write(addrs, blockaddr)$

1. Call $Share(d-\ell+1)$ to get sharings of random values $[r_1], \ldots, [r_{d-\ell+1}]$.
2. For $i = 1, \ldots, m$ compute

$$[f(i)] = \sum_{k=1}^{\ell} \lambda_k^i [x_k] + \sum_{k=\ell+1}^{d+1} \lambda_k^i [r_{k-\ell}]$$

   where $\{[x_k]\}$ is a set of individually shared values pointed to by $addrs$ and f is the polynomial from Lemma 4. Note that this only requires local computation.
3. For $i = 1, \ldots, m$ each player sends "write *blockaddr*" and his share of $[f(i)]$ to server D_i.
4. For $i = 1, \ldots, m$ D_i uses the shares received to reconstruct and store $f(i)$ under $blockaddr$.

Figure 22: Protocol for writing a block

Protocol $Read(addrs, blockaddr)$

1. each player sends "read *blockaddr*" to each server.
2. Each server D_i retrieves $f(i)$ using $blockaddr$, forms a set of shares $[f(i)]$ from $f(i)$ and sends a share to each player.
3. For $k = 1, \ldots, \ell$, the players locally compute $[x_k] = \sum_{i=1}^{m} \delta_i^k [f(i)]$ and associate the results with the addresses in $addrs$.

Figure 23: Protocol for reading a block.

Security and efficiency of the read and write protocols We need to show how to simulate the adversary's view of the read and write protocols, without knowing any of the actual data. As this is straightforward, we give only a brief sketch:

In the write protocol, only step 3 involves communication that must be simulated. To do this for a corrupt server D_i, the simulator chooses a random value to play the role of $f(i)$, and interpolate a set of shares $[f(i)]$ consistent with $f(i)$ and what the adversary already knows. This is possible as the adversary only has a non-qualified set of shares of $f(i)$, and it matches exactly the distribution the adversary would see in real life since he has no information on $r_1, \ldots, r_{d-\ell+1}$.

For the read protocol, only step 1 involves communication. To simulate what corrupt players receive from honest servers, the simulator simply chooses random values. This simulates perfectly, as the adversary only sees an unqualified part of each $[f(i)]$ when D_i is honest.

Finally, note that that Lemmas 4, 5 clearly imply that all involved secret values are correctly written and read when executing the protocols. Therefore, if any such value is ever opened, it will be correct in the ideal as well as in the real process.

We conclude that these simulations, together with the standard techniques for simulating the input, open and arithmetic operations show that we have

an implementation of $\mathcal{F}$ that is perfectly secure against a passive adversary corrupting at most $t_p < n/2$ players and $t_s \in O(m)$ servers.

Finally, by inspection, it is easy to see that each call to *Read* or $Write$ involves communication of $O(nm)$ field elements and $O(nm^2)$ local field operations.

6.4.2 Implementation for Malicious Servers and Players

In this section we show how to handle the case where a constant fraction of the servers and players are corrupted by a malicious adversary. A first easy observation is that when less than $t_p < \frac{n}{3}$ players are actively corrupted, we can implement the input, open and arithmetic operations of $\mathcal{F}$ using standard techniques from [BOGW88] such that ordinary Shamir secret sharing is still the way values are represented. Moreover, when an honest server receives shares of some value from the players, it can use standard error correction techniques to reconstruct the right value. We need, of course, an implementation of $\mathcal{F}_{Rand}$ with active security and we show how to do this later.

The main problem is that malicious servers may return incorrect shares in read operations. While an obvious solution is to authenticate each share held by a server, this is not trivial to do efficiently: since shares must be kept private, the players have to do a secure computation of the authentication value. We present two solutions that work along these lines. The first uses information theoretically secure macs, leading to protocols *RobustRead* and $RobustWrite$ found in appendix C. We get the following result:

Theorem 18. *Together with the standard techniques for implementing the input, open and arithmetic operations, RobustRead and $RobustWrite$ form a statistically secure implementation of $\mathcal{F}$ in the F_{Rand}-hybrid model for an adversary corrupting at most $t_p < n/4$ players and $t_s \in \Theta(m)$ servers actively, or an adversary corrupting at most $t_p < n/2$ players passively and $t_s \in \Theta(m)$ servers actively. Each call to RobustRead or $RobustWrite$ involves communication of $O(nm)$ field elements and $O(nm^2)$ local field operations.*

This solution is relatively simple but needs extra functionality from F_{Rand} that we only know how to implement for passively corrupted players or a small number of actively corrupted players. In the next section, we present a second solution that scales better with the number of players.

6.4.3 A Scalable Way of Handling Malicious Players and Servers

In this section, we show how to handle actively corrupted players and servers in a way that scales well with n. The solution based on Schnorr signatures [Sch91] living in a group G of order q where q is the size of $\mathbb{F}$, the field we compute in, and where we assume q to be prime. We will define protocols $ScalableRobustWrite$ and *ScalableRobustRead* and show the following:

Theorem 19. *Assuming that Schnorr signatures in G are secure against chosen message attacks, then, together with the standard techniques for implementing*

the input, open and arithmetic operations, the protocols ScalableRobustRead and ScalableRobustWrite form a computationally secure implementation of $\mathcal{F}$ in the F_{Rand}-hybrid model for an adversary corrupting at most $t_p < n/3$ players and $t_s \in \Theta(m)$ servers actively. ScalableRobustRead and ScalableRobustWrite each involve communication of $O(nm)$ field elements and $O(nm^2)$ local field operations plus $O(nm)$ exponentiations in G per invocation.

The $O(nm)$ exponentiations in G we require are equivalent to $O(\log(|\mathbb{F}|)nm)$ multiplications in G. To see if this extra cost is going to dominate, consider that if we use a state of the art implementation of G based on elliptic curves, a multiplication in G takes time comparable to a multiplication in $\mathbb{F}$. So comparing $O(nm^2)$ field operations to $O(\log(|\mathbb{F}|)nm)$ multiplications in G boils down to comparing m and $\log(|\mathbb{F}|)$. These parameters are not really comparable in general, but since the idea of our model is to consider a moderate number of players and many servers, it does not seem unreasonable to consider m and $\log(|\mathbb{F}|)$ to be the same order of magnitude. Under this assumption, the solution has the same complexity as the passively secure protocol up to a constant factor. We add that in many settings, the communication cost is the main limiting factor, in which case our solution will be satisfactory regardless of other factors.

Some words on the basic ideas of our protocols: the main problem we face is that the servers may send incorrect data in the read protocol. To protect against this, the players will sign each server's share using a shared signing key. However, this share must not become publicly known and we know of no signature scheme where we can sign efficiently when both the signing key and the message are secret-shared. To overcome this, we instead sign a commitment to the server's share since this commitment can indeed be publicly known. We also have the problem of the servers replaying old signed values, but this can be solved easily by maintaining a sequence number $c_{blockaddr}$ (initally 0) counting how many times we wrote to *blockaddr*, signing this, and having the servers remember it.

Next we describe the protocol in more detail. First, some setup: let G be a group of order q where random elements $g, h \in G$ have been chosen to be used as public key for Pedersen commitments in G. We also assume that a verification key α, β for our Schnorr signature scheme has been set up, with $\beta = \alpha^a$ for some $a \in \mathbb{Z}_q$ where the players hold a sharing $[a]$ of a. We assume that nobody knows (or can compute) the discrete logarithm of h base g, or of β base α.

Recall that in Schnorr's scheme, a signature on m is a pair (γ, δ) satisfying $\gamma = \alpha^\delta \cdot \beta^{H(\gamma,m)}$ for some collision intractable hash function H. Observe that signatures can be easily verified, and computed knowing a by setting $\gamma = \alpha^r$ and $\delta = r - a \cdot H(\gamma, m)$ for a random $r \in \mathbb{Z}_q$. We will need functions from F_{Rand} in addition to those we defined earlier. The new functions are defined here:

Our protocol as well as some subprotocols described later uses player elimination where some players are disqualified underway, so that only a subset of the players actually participate at any given time. For ease of exposition, we suppress this fact in our description below, but emphasize that an actual implementation must keep track of who participates. Furthermore, in the protocol

Additional functions for F_{Rand}

$share^+(num)$ Same as *share*, generate *num* random shared values $[r_i]$, but in addition send α^{r_i} to all players.

$share^{++}(num)$ Generate *num* pairs of random shared values $[u_i], [v_i]$ in the same way as in *share*, but in addition, send $g^{u_i}h^{v_i}$ to all players.

Figure 24:

below for reading and writing, we use a global variable u that is remembered between calls to the read/write protocols. It points to a player and is initially 1. P_u is a player that does computational work on behalf of all players, to save resources. If P_u is found to be corrupt, he is replaced by P_{u+1}. In such a case, our protocols usually do an amount of work that is much larger than in normal operation, but since this can only happen t_p times, it only incurs an additive overhead that is independent of the size of the computation we do. Therefore it does not affect the asymptotic complexity of our solution.

Due to space limitations a more detailed argument for security of these protocols is deferred to appendix D. The most important observation in the write protocol is that in step 5, players effectively evaluate a polynomial with coefficients $s_i - u_i - \tau_i$ in a random point x. If there is an error, i.e., $\tau_i \neq s_i - u_i$ for some i, the polynomial is not zero and can have at most m roots. Hence the check is OK with probability at most $m/|\mathbb{F}|$ which is negligible. Note also that in the read protocol, an honest P_u will always get good data from the honest servers and hence can make S_u be of size at least $m - t_s$—this also means P_u must be corrupt if S_u is too small. On the other hand, even if P_u is corrupt, the commitments and signatures are checked by everyone and hence the last step is always successful if S_u is large enough.

6.4.4 Implementation of F_{Rand}

To implement the *Share* command, we can use a protocol from [BTH08], which is based on so-called hyperinvertible matrices. A matrix is hyperinvertible if any square submatrix is invertible.

We sketch the idea here and refer to [BTH08] for details. Using an n by n hyperinvertible matrix M, players can generate $\Theta(n)$ random shared values at cost $O(n)$ communication per value: each player P_i simply constructs $[r_i]$ for random r_i and sends shares to the other players. Now each player P_j collects a vector $(r_{1,j}, r_{2,j}, \ldots, r_{n,j})$ of shares he received, multiplies the vector by M to obtain a new vector $(s_{1,j}, s_{2,j}, \ldots, s_{n,j})$, and outputs $(s_{1,j}, s_{2,j}, \ldots, s_{n-t_p,j})$. Clearly this forms shares of the first $n - t_p$ entries in $M \cdot (r_1, \ldots, r_n)$. By the hyperinvertible property, these entries are in 1-1 correspondence with the $n - t_p$ values chosen by honest players, and are therefore completely unknown to the adversary. We get the required efficiency, as long as t_p is any constant fraction of n, and [BTH08] shows how to add checks to get security also for $t_p < n/3$ actively corrupted players.

We now go on to describe a protocol for the $share^+$ command that produces

Protocol $ScalableRobustWrite(addrs, blockaddr)$

1. Send "begin write at *blockaddr*" to the servers. Each server returns its value of $c_{blockaddr}$, players decide the correct value by majority, and increment the value.
2. Call $Share(d-\ell+1)$ to get sharings of random values $[r_1], \ldots, [r_{d-\ell+1}]$.
3. For $i = 1, \ldots, m$ compute

$$[f(i)] = \sum_{k=1}^{\ell} \lambda_k^i [x_k] + \sum_{k=\ell+1}^{d+1} \lambda_k^i [r_{k-\ell}]$$

   where $\{[x_k]\}$ are a set of individually shared values pointed to by *addrs* and f is the polynomial from Lemma 4. Note that this only requires local computation. In the following, we set $s_i = f(i)$.
4. Call subprotocol $share^{++}(m)$ to obtain $c'_i = g^{u_i} h^{v_i}$ and sharings $[u_i], [v_i]$ for $i = 1 \ldots m$. Also call $share(1)$ to get $[x]$.
5. We now want to adjust c'_i so it becomes a commitment to s_i. Players compute locally shares in $[s_i - u_i]$ and send them to P_u who computes $\tau_i = s_i - u_i$ and broadcasts all the τ_i's. To verify P_u's work, players open x by sending all shares to everyone and compute locally $\sum_i x^i([s_i] - [u_i] - \tau_i) = [\sum_i x^i(s_i - u_i - \tau_i)]$. This value is opened by each player sending his share to all players. If the resulting value is 0 continue to the next step. Otherwise, P_u is disqualified, we set $u = u+1$, and all players exchange shares of $s_i - u_i$ and compute the correct values τ_i.
6. All players compute $c_i = g^{\tau_i} c'_i$. We now want to sign c_i (and some more data).
7. Call $share^{+}(m)$ to obtain a sharing $[r_i]$ of some random value $r_i \in \mathbb{Z}_q$, and $\gamma_i = \alpha^{r_i}$, for $i = 1 \ldots m$.
8. Using hash function H, each player computes their share $[\delta_i]_j = [r_i] - [a]H(\gamma_i, c_i, c_{blockaddr})$ of δ_i.
9. The players all send "complete write at *blockaddr* using $([s_i], [v_i], [\delta_i], \gamma_i)$" to server i.
10. Each server i reconstructs s_i, v_i, δ_i using error correction to handle incorrect shares from corrupt players, computes γ_i (taking majority decision on the values received), increments $c_{blockaddr}$ and stores all values.

Figure 25: Scalable protocol for writing a block.

$\Theta(nm)$ values of form $[r], \alpha^r$ where the r's are randomly chosen and the only information released to the adversary on r is α^r. If the call asks for a smaller number of values, we simply buffer excess values until the next call.

We then describe the subprotocol required to compute the results in step 3 above efficiently:

A protocol for the $share^{++}$ command is obtained by a trivial extension of the $share^{+}$ protocol. Due to space constraints, we leave the details to the reader.

As for security of the protocols, correctness of the checks in steps 2e and 3

Protocol *ScalableRobustRead*$(addrs, blockaddr)$

1. All players send "read at $blockaddr$ towards P_u" to all the servers. The servers each compute u by majority decision.
2. Each server D_i sends γ_i, δ_i, c_i to player u and a share from $[s'_i], [v'_i]$ to each player, with the implied claim that $(s'_i, v'_i) = (s_i, v_i)$ as last stored by the read protocol, and that $c_i = g^{s_i} h^{v_i}$. The current value of $c_{blockaddr}$ is also sent, and players decide on reception on the correct value of $c_{blockaddr}$ by majority.
3. Since we cannot rely on servers to supply consistent shares of s'_i, v'_i, players generate their own consistent shares and adjust them to what the (honest) servers sent: Call $share^{++}(m)$ to obtain sharings $[b_{x,i}]$ and $[b_{y,i}]$ of random elements of $\mathbb{F}$, for $i = 1, \ldots, m$ as well as a commitment $g^{b_{x,i}} h^{b_{y,i}}$ to $b_{x,i}$, using $b_{y,i}$ as the randomness.
4. For $i = 1, \ldots, m$, everybody sends their shares in $[s'_i - b_{x,i}]$ and $[v'_i - b_{y,i}]$ to P_u. P_u attempts to reconstruct $x_i = s'_i - b_{x,i}$ and $y_i = v'_i - b_{y,i}$. He verifies for all i that (γ_i, δ_i) is a valid signature on $c_i, c_{blockaddr}$ and that $c_i = g^{x_i} h^{y_i} \cdot g^{b_{x,i}} h^{b_{y,i}} (= g^{x_i + b_{x,i}} h^{y_i + b_{y,i}} = g^{s'_i} h^{v'_i})$. This may fail for some i's, in which case P_u sets $x_i = y_i = \perp$.
5. P_u broadcasts γ_i, δ_i, c_i and (x_i, y_i), for $i = 1, \ldots, m$ to the players with the implied claim that $x_i = s'_i - b_{x,i}$ and $y_i = v'_i - b_{y,i}$.
6. Each player verifies for all i that (γ_i, δ_i) is a valid signature on $c_i, c_{blockaddr}$ and that $c_i = g^{x_i} h^{y_i} \cdot g^{b_{x,i}} h^{b_{y,i}}$. Let S_u be the set of i for which this holds. If S_u is smaller than $m - t_s$, disqualify P_u, set $u = u + 1$ and restart the read protocol. Otherwise, the players compute $[b_{x,i}] + x_i = [b_{x,i} + x_i]$ for $i \in S_u$. Note that unless the signature scheme or commitment scheme has been broken, $b_{x,i} + x_i = s'_i = s_i$.
7. For $k = 1, \ldots, \ell$, the players compute locally

$$[x'_k] = \sum_{i=1}^{m} \delta_i'^k [b_{x,i} + x_i]$$

and associates the results with the addresses in $addrs$. Here $\delta_i'^k$ is a set of interpolation coefficients such that $\delta_i'^k = 0$ if i is not in S_u, and the rest is set such that the correct value for x'_k is computed. This is possible because we set the parameters such that there are enough honest servers to reconstruct only from their data. This still allows t_s to be in $\Theta(m)$.

Figure 26: Scalable protocol for reading a block.

can be argued in a way similar to the similar check in the *ScalableRobustRead* protocol. Security then follows from this and the hyper-invertible property of M. See appendix E for a simulation proof.

Protocol *share*$^+$:

1. Use the protocol from [BTH08] to generate random shared values $[r_b^a]$ and $[x_a]$ for $a = 1..n$, $b = 0..m$.
2. In parallel, for $a = 1, \ldots, n$ do:
   (a) Players send their shares in $[r_b^a]$ to P_a, for $b = 0 \ldots m$.
   (b) P_a reconstructs the r_b^a's, computes $\chi_b^a = \alpha^{r_b^a}$ and broadcasts these values, for $b = 0 \ldots m$.
   (c) Now we want to verify that P_a has computed correctly, so we open x_a by broadcasting all its shares.
   (d) Locally compute $[y_a] = [\sum_{b=0}^m {x_a}^b r_b^a] = [r_0^a] + [\sum_{b=1}^m {x_a}^b r_b^a]$ and open y_a to everyone.
   (e) All players check that $\alpha^{y_a} = \prod_{b=0}^m (\chi_b^a)^{{x_a}^b}$. If this is not satisfied, P_a is disqualified, open r_b^a for $b = 0, \ldots, m$, and all players compute the correct values of $\alpha^{r_b^a}$. Hence, in any case, we can assume we continue with correct values of $\chi_b^a = \alpha^{r_b^a}$, for $a = 1 \ldots n$, $b = 0 \ldots m$.
3. We form column vectors $V_b, b = 1 \ldots m$, with n entries, where entry a is of form $([r_b^a], \alpha^{r_b^a})$. Players then compute a new column vector $M \cdot V_b$, formed as follows: let $\gamma_1, \ldots, \gamma_n$ be the k'th row of M. Then the k'th entry of $M \cdot V_b$ is $([\sum_a \gamma_a r_b^a], \alpha^{\sum_a \gamma_a r_b^a} = \prod_a (\alpha^{r_b^a})^{\gamma_a})$. This can be done by local operations only, but to get the required efficiency, it is necessary that the players share the work of doing the exponentiations. We do this using subprotocol *AmortizedExp* described below.
4. Output the first $n - t_p$ entries of $M \cdot V_b$, for $b = 1 \ldots m$.

Figure 27: Protocol for sharing random exponents

Protocol *AmortizedExp*:

1. Each player P_k computes a part of the result, namely $\beta_b^k = \prod_{a=1}^n (\alpha^{r_b^a})^{\gamma_a}$, for $b = 1 \ldots m$, where $(\gamma_1, \ldots, \gamma_n)$ is the k'th row of M. P_k broadcasts his results.
2. Generate and open a random value x in the same way as in *share*$^+$.
3. All players compute $(\delta_1, \ldots, \delta_n) = (x^0, \ldots, x^{n-1}) \cdot M$, i.e., we take a linear combination of the rows of M. Players check that, for $b = 1, \ldots, m$, we have:

$$\prod_{k=1}^n (\beta_b^k)^{x^{k-1}} = \prod_{a=1}^n (\alpha^{r_b^a})^{\delta_a}$$

   If this does not hold, at least one player output incorrect results. All players then compute all the β_b^k, find the player(s) who cheated and disqualify them. In any case, output the β_b^k's.

Figure 28: Protocol for exponentiating many values.

6.5 Running Oblivious Algorithms on $\mathcal{F}$

So far we've discussed how to implement a "black box" functionality called $\mathcal{F}$. In this section, we show how to execute oblivious programs that use $\mathcal{F}$ as a

library of subroutines, and analyze the time and communication cost of doing so.

First, we need to define what a program for $\mathcal{F}$ is. We shall take it to mean a finite sequence of instructions $\mathcal{C}$, each of which is either one of $\mathcal{F}$'s commands with suitable parameters, or a control command $(\texttt{goto}, i)$ which is described below.

These programs are executed on a machine with a memory $\mathcal{M}$, the configurations of which are $\in \mathbb{Z}_q^M$. The machine also has a program counter register, $pc \in \mathbb{Z}_q$ and a finite disk $\mathcal{D}$ with configurations in $(\mathbb{Z}_q^\ell)^D$ where ℓ is the block size. All $\mathbb{Z}_q$-elements are initially set to 0. We assume that $q \geq M, D, (|\mathcal{C}|+1)$[16].

To execute a program, repeat this main loop: Leak pc. If $pc \geq |\mathcal{C}|$, halt; else if $\mathcal{C}[pc] = (\texttt{goto}, v)$, set $pc \leftarrow \mathcal{M}[v]$; else act according to $\mathcal{C}[pc]$ as described by $\mathcal{F}$ and subsequently increment pc. Then repeat.

With these instructions, a fixed-address jump instruction "goto p" can be expressed as the sequence $(\texttt{const}, 0, p)$, $(\texttt{goto}, 0)$. A conditional jump "if $\mathcal{M}[v_1] \diamond \mathcal{M}[v_2]$ goto x else goto y" where $\diamond$ is one of the two comparisons, can be expressed as the sequence $(\diamond, 0, v_1, v_2)$, $(\texttt{const}, 1, x - y)$, $(*, 0, 1, 0)$, $(\texttt{const}, 1, y)$, $(+, 0, 1, 0)$, $(\texttt{goto}, 0)$. Out of these, control structures like **if-then-else**, **while** and **for** can be built.

A program $\mathcal{P}$ that makes random choices can be viewed as a function mapping inputs to distributions of streams of leakage. A program is said to be *perfectly oblivious* if the output reveals this distribution for each input that can produce that output. Formally speaking, $\mathcal{P}$ is oblivious if there exists a polynomial time randomized turing machine T such that

$$\forall o \forall i \colon (\exists r \colon \mathcal{P}(i, r) = o) \Rightarrow T_{\mathcal{P}}(o) \sim^p traces(\mathcal{P}, i).$$

That is to say, for each output o and for each input i that can make $\mathcal{P}$ produce o as the output, the distribution of instructions executed by $\mathcal{P}$ (over $\mathcal{P}$'s coin flips r) can be sampled by T using just the output. Computational and statistical obliviousness is defined similarly, where T samples suitably indistinguishable distributions. If a program is perfectly oblivious, we say it is *oblivious*.

A note about this definition: a more commonly used definition of obliviousness is that the distribution of traces is computable knowing only the program and not the output. Here, what we want to capture is really that the program is oblivious from the adversary's point of view; that is to say, given what the adversary is allowed to know in the ideal world (the output), he could himself have computed what he sees in the real world. Using this definition, we are now ready to state the following theorem:

Theorem 20. *Given an oblivious program $\mathcal{P}$ for $\mathcal{F}$, we can implement using $\mathcal{F}$ a UC-secure functionality $\mathcal{F}_{\mathcal{P}}$ that has only* `input` *and* `output` *commands, such that if a set of inputs is given to $\mathcal{P}$ and the same inputs are given to $\mathcal{F}_{\mathcal{P}}$, the outputs produced by $\mathcal{F}_{\mathcal{P}}$ equal those produced by $\mathcal{P}$ when run on $\mathcal{F}$.*

[16] Otherwise, we could express things in terms of an implicit "bignum" library, but this would be unwieldy.

Proof. To implement $\mathcal{F}_{\mathcal{P}}$, simply run $\mathcal{P}$ on $\mathcal{F}$ as described above. This clearly creates the desired relationship between inputs and outputs. A UC simulator takes the output from running $\mathcal{F}$ on $\mathcal{P}$ and feeds it to the turing machine T that exists because $\mathcal{P}$ is oblivious. This yields a simulated trace of $\mathcal{P}$ that is suitably (i.e. perfectly, statistically or computationally) indistinguishable from the real world.

We also want to study the efficiency of the resulting implementation. To this end, let $IO_{\mathcal{P}}(N)$ be the number of read and write operations that $\mathcal{P}$ executes on input of size N. It then follows from our previous results that the total cost of executing these using our protocols is $IO_{\mathcal{P}}(N)O(mn)$ transfers of field elements, and $IO_{\mathcal{P}}(N)O(m^2n)$ local field operations.

As for the operations that do computation, it is well known from the standard protocols for field arithmetic that addition and subtraction can be done with no communication, but interaction is needed for the non-linear operations. Let $Comp_{\mathcal{P}}(N)$ be the total number of computation commands done by $\mathcal{P}$ on input of size N, while $Comm_{\mathcal{P}}(N)$ is the number of commands that require communication. We will (conservatively) assume that each computation command requires each player to do one field operation, and—for commands that require communication—to send one field element. This means that the total cost of executing $\mathcal{P}$ using our protocols is $IO_{\mathcal{P}}(N)O(mn) + Comm_{\mathcal{P}}(N)n$ communication and $IO_{\mathcal{P}}(N)O(m^2n) + Comp_{\mathcal{P}}(N)n$ computation. We will assume that $\mathcal{P}$ implements an I/O-efficient algorithm, which typically means $IO_{\mathcal{P}}(N)$ is $O(N/\ell \cdot \log^c(N))$, where c is a constant that may vary depending on the concrete problem, and ℓ is the blocklength. We will let ℓ be $\Theta(m)$, and so for I/O-efficient programs, we get $O(nN\log^c(N)) + Comm_{\mathcal{P}}(N)n$ communication and $O(mnN\log^c(N)) + Comp_{\mathcal{P}}(N)n$ computation.

We want to compare this to executing $\mathcal{P}$ in two alternative scenarios. First, we consider a case where players have memory enough to store each field element individually using standard Shamir-sharing, in particular we don't even need the servers. This requires more storage than in our case, namely a total of $\Theta(Nn)$ field elements as opposed to $\Theta(N)$. On the other hand, players now only have to do the computation required by $\mathcal{P}$, implying $Comm_{\mathcal{P}}(N)n$ communication and $Comp_{\mathcal{P}}(N)n$ computation. Since $Comp_{\mathcal{P}}(N) \geq Comm_{\mathcal{P}}(N)$, it is now clear that if $Comm_{\mathcal{P}}(N)$ is $\Omega(N\log^c(N))$, then the communication cost of our solution is the same within a constant factor, while our computation is a factor m larger. Thus we get a tradeoff between memory and computation. Note, however, that by being conservative in estimating the cost of the actual computation, we have taken care not to give ourselves an unfair advantage. Depending on $\mathcal{P}$ and the precise cost of operations, our solution may be the same or better in all respects. Of course, one could object that if we do not have to care about I/O-efficiency, players could execute a different and perhaps more efficient program solving the same problem. However, there are many examples (we show a few below) where the demand for I/O-efficiency does not incur a significant increase in computational complexity.

This first comparison is actually not completely fair because we are allowing

the players a large amount of memory that we do not assume in our model. After all, the motivation for our work in the first place is to consider cases where the players cannot or do not want to store the entire database themselves. We therefore also compare to a second case where the database is stored on the servers, but again each field element is shared invidually using Shamir sharing. To read/write a field element in the straightforward way in this scenario costs $\Theta(nm)$ communication and computation. It is now not necessary to ensure that the memory access pattern is I/O-efficient, but is certainly reasonable to assume that one needs to look at all the inputs. We therefore get an I/O cost in this case of $\Omega(Nnm)$ communication and computation. This should be compared to our cost of $O(nN\log^c(N))$ communication and $O(mnN\log^c(N))$ computation, i.e., up to log factors the same computation cost and a factor m smaller communication. Furthermore, our memory usage is a factor $\Theta(m)$ smaller.

6.6 Example Applications

There are plenty of example applications for the previous theorem. Oblivious algorithms have already been studied in literature; sorting networks are the prime example. Another source of examples is MPC protocols for specific tasks. Oblivious algorithms are often used as the basis for protocols, where two or more parties wish to query or process some secret dataset. The present setting is very suited for such examples: The data is often processed during one or more scans of the entire dataset.

Secure Datamining Secure datamining was introduced by Lindell and Pinkas [LP00] and k-means clustering is one example. Given N points p_j in a d-dimensional space, group these points into k *clusters*, i.e., find trends in the data. The k-means clustering protocol of Jagannathan and Wright [JW05] is easily generalized to consider a secret shared database held by a number of disk-servers.

The overall idea is to start with a set of k initial *cluster centers*. These are then updated: 1) assign each p_j to the closest center, 2) replace each center with the average of its points. The process is repeated until some (revealed) termination condition is fulfilled, e.g., the clusters move sufficiently little. The details are seen below; the d-dimensional points are stored as d secret shared coordinates. To improve readability, we refer to shared points as a single secret sharing, writing $[p_j]$ rather than $([p_{j,1}], \ldots, [p_{j,d}])$.

- Init cluster centers, $[c_i]$
- `While` Termination condition not reached `do`
  - Init accumulators ($[N_i]$: number of points assigned to $[c_i]$; $[\sigma_i]$: sum of points assigned to $[c_i]$)
  - `For` each point $[p_j]$
    - Determine index, $[k]$, of closest cluster

    * `For` each cluster center $[c_i]$: Add $\left([k] =^? i\right)$ to $[N_i]$ and $\left([k] =^? i\right) \cdot [p_i]$ to $[\sigma_i]$
  - `For` each cluster center $[c_i]$: $[c_i] \leftarrow [\sigma_i] \div [N_i]$

We ignore many details, notably computing distances and averages, and determining the closest center. Our goal is *not* to provide a k-means clustering protocol, but to demonstrate that the task can be performed with most of the data residing "on disk.". Access is oblivious and I/O-efficient: the parties hold the $[c_i]$, $[\sigma_i]$, and $[N_i]$ in memory, while the dataset (the $[p_j]$) is accessed by linear scans.

Sorting As a second example, consider a Shell sorting (SS) network, sorting N elements: for each value $\delta = 2^j 3^k < N$ in decreasing order, for $i = 1, \ldots, N - \delta$, compare and exchange elements i and $i + \delta$. Clearly that's $O(N \log_2 N \log_3 N)$ comparisons, and they're organised in $O(\log^2 N)$ pairs of parallel forward scans. So SS does $IO_{SS}(N) = \frac{N}{\ell} \log^2 N$ I/O, and $Comp_{SS}$ is $\Theta(N \log^2 N)$: it falls in the class of I/O-efficient oblivious programs we analyzed in the previous section. For proof of correctness, see [Pra72].

There are many alternatives to the SS network. The odd-even merge sort [Bat68] provides the same result, but better round complexity. Randomized SS [Goo10] seems to require many I/O operations as it (recursively) compares elements in a randomized complete matching between the left and right halves of the data. Using an I/O-efficient oblivious sorting algorithm (e.g., the odd-even merge sort adaptation in [GM10]) appears to be a good strategy, but even if we can get to $O(\frac{N}{\ell} \log_{\frac{M}{\ell}} \frac{N}{\ell})$ I/O ([GM10] gets to $O(\frac{N}{\ell} \log^2_{\frac{M}{\ell}} \frac{N}{\ell})$), we still need $\Omega(N \log N)$ comparisons, each of which requires computation and communication.

A priority queue Toft's oblivious priority queue (PQ) [Tof11] is an oblivious datastructure allowing two operations: `Insert` $([x], [p])$ which adds element $[x]$ with priority $[p]$; and `GetMin` $()$, which removes and returns the element with minimal priority. The solution keeps track of a list that is sorted lazily, through buffering. Extraction of the minimal pair is trivial, except for occasional buffer flushing.

The construction is based entirely on oblivious merging as in Batcher's odd-even merge sort [Bat68], which is I/O-efficient. Combining the techniques of this paper with those of [Tof11] implies a PQ with the bulk of the data held on disk. This makes perfect sense: the bulk of the data is very rarely touched, hence storing it in a compact fashion ensures a very low memory overhead with little impact on efficiency.

References

[And09] Daniel Andersson. Hashiwokakero is np-complete. *Information Processing Letters*, 109(19):1145, 2009.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[Bar89] 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.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BL90] J. Benaloh and J. Leichter. Generalized secret sharing and monotone functions. In *Proceedings on Advances in cryptology*, CRYPTO '88, pages 27–35, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[BM85] G. Blakley and Catherine Meadows. Security of ramp schemes. In George Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 242–268. Springer Berlin / Heidelberg, 1985.

[BM04] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Proceedings of the 5th conference on Theory of cryptography*, TCC'08, pages 213–230, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bum11] Jim Bumgardner. Free slitherlink puzzles from krazydad, 2008–2011.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.

[CDH07] Ronald Cramer, Ivan Damgård, and Robbert Haan. Atomic secure multi-party multiplication with low communication. In *Proceedings of the 26th annual international conference on Advances in Cryptology*, EUROCRYPT '07, pages 329–346, Berlin, Heidelberg, 2007. Springer-Verlag.

[CDI05] 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.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *Proceedings of the 19th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'00, pages 316–334, Berlin, Heidelberg, 2000. Springer-Verlag.

[CDvdG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *CRYPTO*, pages 87–119, 1987.

[Col84] Charles J. Colbourn. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Dam12] Ivan Damgård. Personal communication, 2012.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[DH09] Erik D. Demaine and Robert A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory. In Michael H. Albert and Richard J. Nowakowski, editors, *Games of No Chance 3*, volume 56 of *Mathematical Sciences Research Institute Publications*, pages 3–56. Cambridge University Press, 2009.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Proceedings of the 27th annual international cryptology conference on Advances in cryptology*, CRYPTO'07, pages 572–590, Berlin, Heidelberg, 2007. Springer-Verlag.

[DPS+11] 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.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–437, 1987.

[FH06] Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, pages 163–168, New York, NY, USA, 2006. ACM.

[FY92] Matthew Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC '92, pages 699–710, New York, NY, USA, 1992. ACM.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18(3):247–287, 2005.

[GM10] Michael T. Goodrich and Michael Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious ram simulations. *CoRR*, abs/1007.1259, 2010.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[Goo10] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

[HM00] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.

[IMR+07] Kamrul Islam, Henk Meijer, Yurai Núñez Rodríguez, David Rappaport, and Henry Xiao. Hamilton circuits in hexagonal grid graphs. In Prosenjit Bose, editor, *CCCG*, pages 85–88. Carleton University, Ottawa, Canada, 2007.

[Ins] The Clay Mathematics Institute. The p versus np question. http://www.claymath.org/millennium/P_vs_NP/.

[JJ] Angela Janko and Otto Janko. Beispiel mit ausführlicher lösung.

[JW05] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 593–599, New York, NY, USA, 2005. ACM.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[LP00] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *CRYPTO*, pages 36–54, 2000.

[Nik11a] Nikoli. Rules of kurodoko, 2001–2011.

[Nik11b] Nikoli. Rules of slither link, 2001–2011.

[NU09] Kosuke Nukui and Akihiro Uejima. Asp-completeness of the slither link puzzle on several grids. *IPSJ SIG Notes*, 2007(23):129–136, 2007-03-09.

[Pra72] Vaughan R. Pratt. *Shellsort and Sorting Networks*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1972. See also `http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/shell/shellen.htm`.

[Sch78] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[SqZ06] Jeff Stuckman and Guo qiang Zhang. Mastermind is np-complete. *INFOCOMP J. COMPUT. SCI*, 5:25–28, 2006.

[Sw93] Hermann Stamm-wilbrandt. A simple linear time algorithm for embedding maximal planar graphs, 1993.

[Tat10] Simon Tatham. Portable puzzle collection. `http://www.chiark.greenend.org.uk/~sgtatham/puzzles/`, 2004–2010.

[Tof11] T. Toft. Secure datastructures based on multiparty computation. Cryptology ePrint Archive, Report 2011/081, 2011. `http://eprint.iacr.org/`.

[Val84] Leslie G. Valiant. Short monotone formulae for the majority function. *J. Algorithms*, 5(3):363–366, 1984.

[Wei] Seth Weiss. Kuromasu. http://www.lsrhs.net/faculty/seth/Puzzles/kuromasu/kuromasu.html.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167, oct. 1986.

[Yat03] Takayuki Yato. Complexity and completeness of finding another solution and its application to puzzles. Master's thesis, Graduate School of Science, University of Tokyo, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[YS03] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundamentals*, E86-A:1052–1060, 2003.

A A python implementation of the Kurodoko reduction

We have attempted to give a semi-formal, unambiguous description of the reduction in sufficient detail. However, nothing can be quite as unambigous and sufficiently detailed as an implementation, so we give one.

```
1  #!/usr/bin/env python
2
3  from sys import argv, exit
4  from itertools import chain
5
6  ####################
7
8  null = None
9  gadget_size = 17
10
11 def unpack(gadget_str):
12     k = gadget_size
13     gadget = [null for _ in range(k**2)]
14     dim, stream = gadget_str.split(':')
15     assert dim == '17x17'
16     ptr = 0
17     for (i, c) in enumerate(stream):
18         if c in '_': pass
19         elif c.islower(): ptr += 1 + ord(c) - ord('a')
20         elif c.isdigit() or c in '?!':
21             if c.isdigit(): assert not stream[i+1].isdigit()
22             gadget[ptr] = c
23             ptr += 1
24         else: assert False
25     assert ptr == len(gadget)
26     return [gadget[i:i+k]
27             for i in range(0, len(gadget), k)]
28
29 def rotate(old):
30     k = gadget_size
31     new = [[null for _ in range(k)] for _ in range(k)]
32     for r in range(k):
33         for c in range(k):
34             new[k - 1 - c][r] = old[r][c]
35     return new
36
37 def flip(old):
38     k = gadget_size
39     new = [[null for _ in range(k)] for _ in range(k)]
40     for r in range(k):
41         for c in range(k):
42             new[r][k - 1 - c] = old[r][c]
43     return new
44
45 ####################
46
47 wire, neg, var, zero, xor, choice, \
48       tee, ltee, bend, ngadgets = range(10)
49
50 # exclamation marks indicate where 'rays' end.
51 gadgets = [
52     # wire
53     "17x17:e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e qqq "
54     "d!2?!3!?2!d qq e!?!2!?!e qq e!?!2!?!e",
55     # neg
56     "17x17:e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e qqq "
57     "e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e",
58     # var
59     "17x17:e!?!2!?!e qq d3!?!2!?!3d d3a!c!a3d"
```

```
60        "qqqqqqqqqqqq",
61        # zero
62        "17x17:e!?!2!?! e qq d3!?!2!?!3 d d3a!c!a3d h7h"
63        "qqqqqqqqqqq",
64        # xor
65        "17x17:qqqq c!d!d!c !b2d2d2b! ?b?d?d?b? !b!d!d!b!"
66        "2b4d3d4b2 !b!i!b! ?b?!g!?b? !b4_4a!c!a4_4b! c!i!c"
67        "d!2?!3!?2! d qq e!?!2!?! e",
68        # choice
69        "17x17:qqqq c!i!c !b2d!d2b! ?b?d?d?b? !b!d!d!b!"
70        "2b4d2d4b2 !b!i!b! ?b?!g!?b? !b4_4a!c!a4_4b!"
71        "c!i!c d!2?!4!?2! d qq e!?!2!?! e",
72        # tee
73        "17x17:qqqqq !b!b!c!b!b! ?b?b?c?b?b?"
74        "!b!b!c!b!b! 2b2b2c2b2b2 !b!d4d!b! ?b?!g!?b?"
75        "!b4_4a!c!a4_4b! c!i!c" "d!2?!3!?2! d qq e!?!2!?! e",
76        # ltee
77        "17x17:qqqq c!f!f !b2b4c3b!b! ?b?a!2!b?b?b?"
78        "!b!b!c!b!b! 2b2b3c2b2b2 !b!d4d!b! ?b?!g!?b?"
79        "!b4_4a!c!a4_4b! c!i!c d!2?!2!?2! d qq e!?!2!?! e",
80        # bend
81        "17x17:qqqq j!f j2b!b! e2!c?b?b? j!b!b! j2b2b2"
82        "e4b4a!b!b! l!?b? d3a!c!a4_4b! m!c d!2?!3!?2!d"
83        "qq e!?!2!?! e",
84        ]
85    gadgets = [s.replace(' ', '') for s in gadgets]
86    assert len(gadgets) == ngadgets
87
88    bend_dr = unpack(gadgets[bend])
89    bend_dl = flip(bend_dr)
90    bend_lu = rotate(rotate(bend_dr))
91
92    tee_d = unpack(gadgets[tee])
93    tee_l = rotate(flip(unpack(gadgets[ltee])))
94    tee_r = flip(tee_l)
95
96    xor_r = rotate(unpack(gadgets[xor]))
97    xor_d = rotate(xor_r)
98    xor_l = rotate(xor_d)
99
100   wire_up = unpack(gadgets[wire])
101   wire_right = rotate(wire_up)
102   wire_left = flip(wire_right)
103
104   gchoice = unpack(gadgets[choice])
105   gwnot = unpack(gadgets[neg])
106   gzero = unpack(gadgets[zero])
107   variable = unpack(gadgets[var])
108
109   xgadgets = [bend_dr, bend_dl, bend_lu, tee_d, tee_l, tee_r, xor_r,
110               xor_d, xor_l, wire_up, wire_right, wire_left, gchoice,
111               gwnot, gzero, variable]
112
113   def main(argv):
114       sat_instance = parse(argv)
115       kurodoko_instance = reduction(sat_instance)
116       encoding = encode(kurodoko_instance)
117       print encoding # a la 'Range' in Simon Tatham's puzzle collection
118
119   def parse(argv):
120       sets = []
121       for s in argv:
122           v = map(int, s.split('.'))
123           if len(v) > 3:
124               raise SystemExit("bad arg: %s (bigger than 3)" % s)
125           sets.append(v)
126       return sets
127
```

```
128 def reduction(clauses):
129     counts = count_vars(clauses)
130     vars = sorted(counts.keys())
131     sorting_network = make_sorting_network(vars, counts, clauses)
132     (w, h) = compute_gadget_count(counts, sorting_network)
133     ww, hh = (gadget_size - 1) * w + 1, (gadget_size - 1) * h + 1
134     grid = [[null for c in range(ww)] for r in range(hh)]
135 
136     add_variables(grid, w, h, vars, counts)
137     add_sorting_network(grid, w, h, counts, sorting_network)
138     add_clauses(grid, w, h, clauses)
139     print_grid(grid)
140     fix_deferred_work(grid)
141 
142     return grid
143 
144 def print_grid(grid):
145     for line in grid:
146         buf = []
147         for x in line:
148             if x != None: buf.append(str(x))
149             else: buf.append(' ')
150         print ''.join(buf)
151 
152 ####################
153 
154 def add_variables(grid, w, h, vars, counts):
155     k = max(counts.values())
156 
157     c = 0
158     for vidx in range(len(vars)):
159         n = counts[vars[vidx]]
160         r = h - k
161 
162         for i in range(n - 1):
163             solder(grid, w, h, r+i, c, tee_l)
164             solder(grid, w, h, r+i, c+1, wire_right)
165             for j in range(1, i+1):
166                 solder(grid, w, h, r+i, c+2*j+0, wire_right)
167                 solder(grid, w, h, r+i, c+2*j+1, wire_right)
168             solder(grid, w, h, r+i, c+2*i+2, bend_lu)
169             for j in range(i+2, n):
170                 solder(grid, w, h, r+i, c+2*j, wire_up)
171         solder(grid, w, h, r + (n-1), c, variable)
172 
173         c += 2*n
174     assert c == w + 1
175 
176 def add_sorting_network(grid, w, h, counts, sorting_network):
177     base = h - 1 - max(counts.values())
178     def swap(r, c):
179         br, bc = base - 2*r, 2*c
180         solder(grid, w, h, br - 0, bc + 0, tee_l)
181         solder(grid, w, h, br - 0, bc + 1, xor_d)
182         solder(grid, w, h, br - 0, bc + 2, tee_r)
183         solder(grid, w, h, br - 1, bc + 0, xor_r)
184         solder(grid, w, h, br - 1, bc + 1, tee_d)
185         solder(grid, w, h, br - 1, bc + 2, xor_l)
186 
187     def mkwire(r, c):
188         for dr in range(2):
189             solder(grid, w, h, base - (2*r + dr), 2*c,
190                    wire_up)
191 
192     for i in range(len(sorting_network)):
193         for j in range(len(sorting_network[i])):
194             if sorting_network[i][j]: swap(i, j)
195             elif j == 0 or not sorting_network[i][j-1]:
```

```
196                     mkwire(i, j)
197
198 def add_clauses(grid, w, h, clauses):
199     for i in range(len(clauses)):
200         solder(grid, w, h, 0, 6*i + 0, bend_dr)
201         solder(grid, w, h, 0, 6*i + 1, wire_right)
202         solder(grid, w, h, 0, 6*i + 2, gchoice)
203         solder(grid, w, h, 0, 6*i + 3, wire_left)
204         solder(grid, w, h, 0, 6*i + 4, bend_dl)
205
206         clause = clauses[i]
207         assert len(clause) <= 3
208         for j in range(len(clause)):
209             v = clause[j]
210             if v < 0: solder(grid, w, h, 1, 6*i + 2*j,
211                              gwnot)
212             else:     solder(grid, w, h, 1, 6*i + 2*j,
213                              wire_up)
214         for j in range(len(clause), 3):
215             solder(grid, w, h, 1, 6*i + 2*j, gzero)
216
217 def fix_deferred_work(grid):
218     for (r, row) in enumerate(grid):
219         for (c, elt) in enumerate(row):
220             if elt != '?': continue
221             n = 1
222             for (dx, dy) in [(-1, 0), (1, 0),
223                              (0, -1), (0, 1)]:
224                 y, x = r + dy, c + dx
225                 while grid[y][x] != '!':
226                     n += 1
227                     x += dx
228                     y += dy
229             grid[r][c] = n
230     for (r, row) in enumerate(grid):
231         for (c, elt) in enumerate(row):
232             if elt in (None, '!'): grid[r][c] = 0
233             else: grid[r][c] = int(elt)
234
235 ####################
236
237 def solder(grid, w, h, r, c, gadget):
238     w, h = (gadget_size - 1) * w + 1, (gadget_size - 1) * h + 1
239     base_y = (gadget_size - 1) * r
240     base_x = (gadget_size - 1) * c
241     for y in range(gadget_size):
242         for x in range(gadget_size):
243             gy, gx = base_y + y, base_x + x
244             if gy < 0 or gx < 0 or gy >= h or gx >= w: continue
245             assert grid[gy][gx] in (null, gadget[y][x])
246             grid[gy][gx] = gadget[y][x]
247
248 ####################
249
250 def count_vars(clauses):
251     counts = {}
252     for clause in clauses:
253         for v in clause:
254             counts[v] = 1 + counts.get(v, 0)
255     return counts
256
257 def make_sorting_network(vars, counts, clauses):
258     clauses = [[abs(v) for v in clause] for clause in clauses]
259     terminals, goals = sum(clauses, []), dict()
260     for (i, v) in enumerate(terminals): goals.setdefault(v, []).append(i)
261     wires = []
262     for v in vars:
263         for j in range(counts[v]):
```

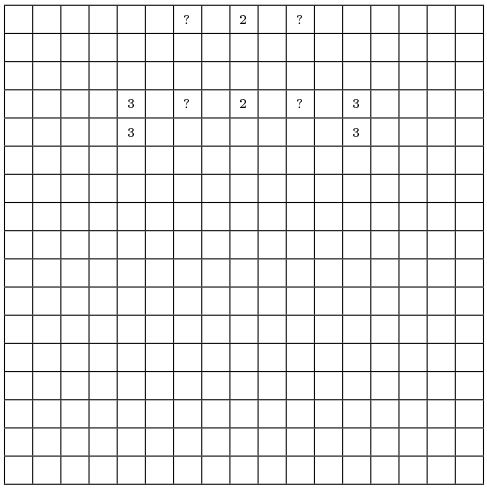
```
264                 wires.append(goals[v][j])
265         n = len(wires)
266 
267         net = []
268         for i in range(n):
269             if wires == sorted(wires): break
270             row = []
271             if i % 2: row.append(0)
272             for j in range(i % 2, n - 1, 2):
273                 if wires[j] > wires[j+1]:
274                     row.extend([1, 0])
275                     wires[j], wires[j+1] = wires[j+1], wires[j]
276                 else: row.extend([0, 0])
277             if i % 2 != n % 2: row.append(0)
278             net.append(row)
279         return net
280 
281     def compute_gadget_count(counts, sorting_network):
282         vals = counts.values()
283         n, k = sum(vals), max(vals)
284         sorting_network = 2 * len(sorting_network)
285         variable_branching = k
286         clauses = 2
287         return (2*n - 1, variable_branching + sorting_network + clauses)
288 
289     def encode(instance):
290         stream = sum(instance, [])
291         buf = []
292         runlength = 0
293         for elt in stream:
294             if runlength == 26 or elt != 0 and runlength > 0:
295                 buf.append(chr(ord('a') - 1 + runlength))
296                 runlength = 0
297                 if elt != 0: buf.append(str(elt))
298             elif elt == 0: runlength += 1
299             else: buf.append('_%d' % elt)
300         return ''.join(buf)
301 
302     if __name__ == '__main__': main(argv[1:])
```

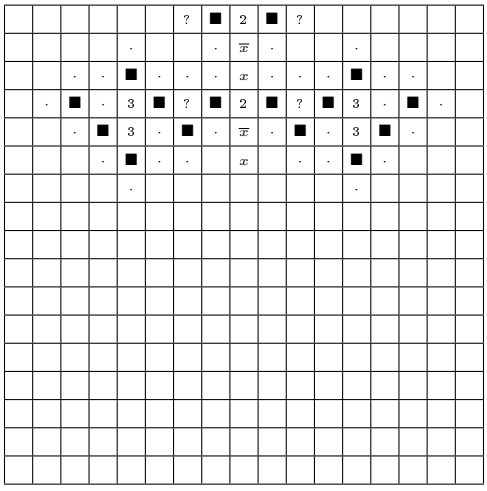
B The Kurodoko gadgets

Here we display all the gadgets (except Wire, which is shown in Figure 3 and Figure 4), along with color deductions that are true in every solution. As earlier, ■, ·, w are black, white, white; all xs are equal, and all $\overline{x}$s are unequal to the xs (easily seen by rule 4).

B.1 The Variable gadget

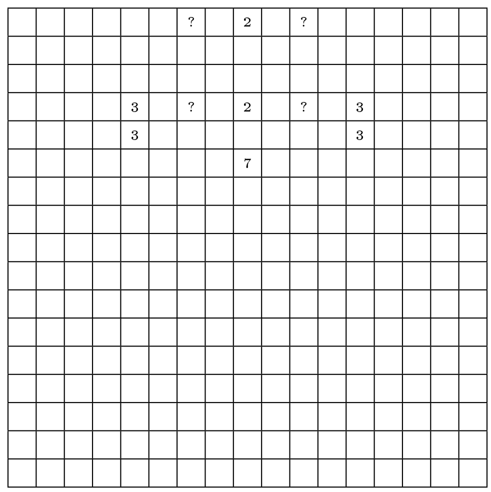
The deductions about black and white squares might be easiest to see if one looks at squares horizontally adjacent to a ? and considers rule 4—it is typically violated if such a square is white.

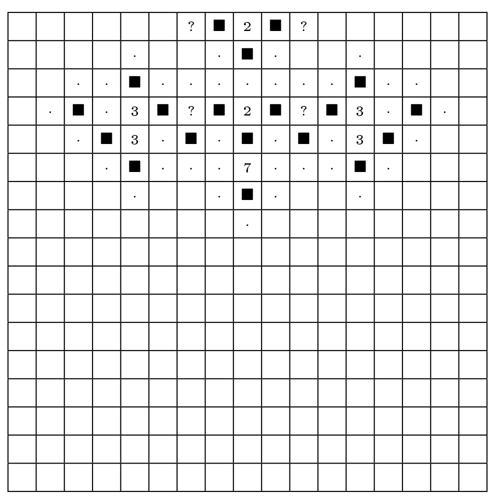




B.2 The Zero gadget

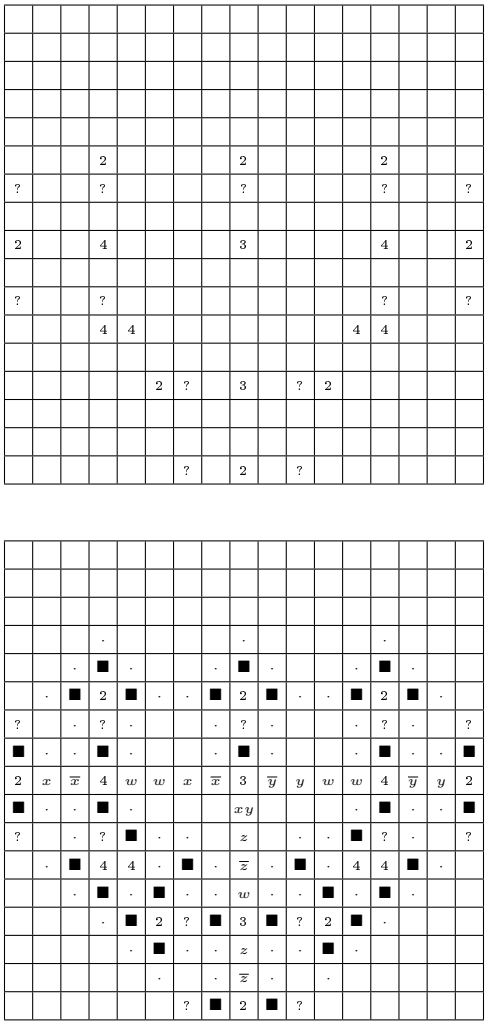
Note the similarity with Variable. We simply force x to be white by putting a clue in its square. As a learning exercise, the reader is encouraged to attempt designing a One gadget.





B.3 The Xor gadget

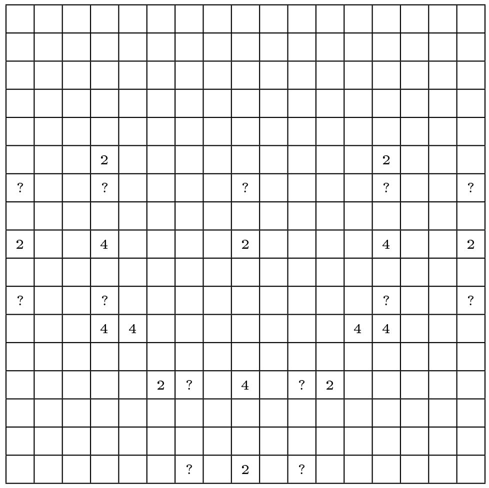
We present the Xor gadget rotated 180° (X^2). Making deductions around the 2s from rule 4 often enables deductions around the 4s.

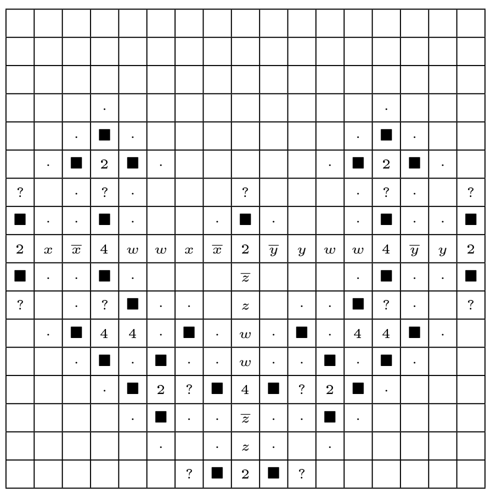


Like $x, \overline{x}$, squares marked $y, \overline{y}$ and $z, \overline{z}$ form consistent opposite pairs. The square marked xy is black if and only if x and y are both black. By rule 4 and by considering the four cases of blackness of $\overline{x}$ and $\overline{y}$ adjacent to the central 3, one can see that $z = x \oplus y$.

B.4 The Choice gadget

Note the similarity with the Xor gadget.

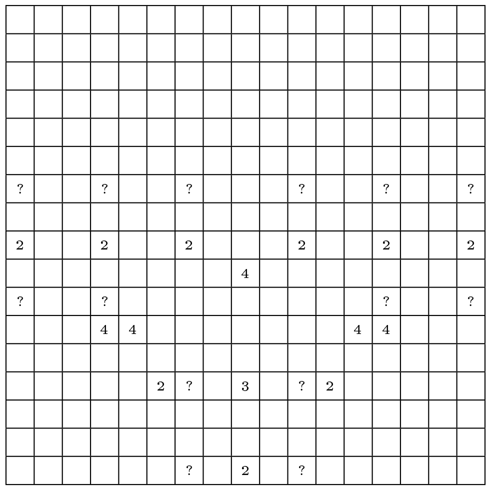


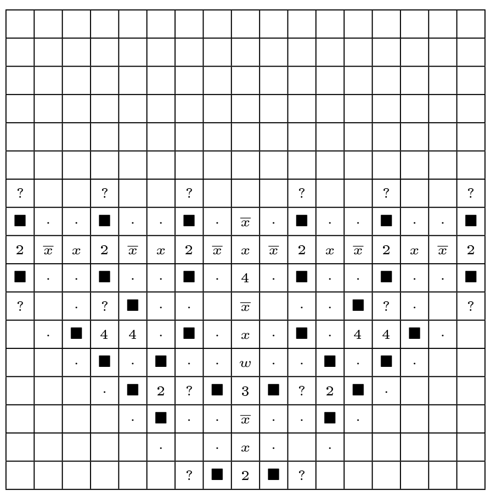


Consider the center clue 2. If (say) x is black then $\overline{x}$ is white. By rule 4 $\overline{y}$ and $\overline{z}$ must be black, and so y and z must be white. This is symmetric, so at most one is black. They can't all be white, or rule 3 is violated as the center clue is trapped.

B.5 The Split gadget

Note the similarity with N^+ in the upper half, and with Xor and Choice in the lower half.

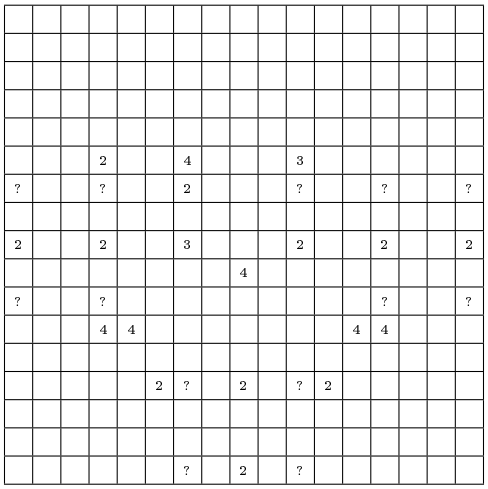


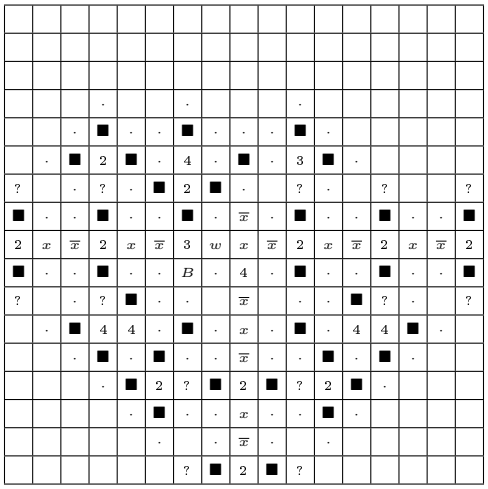


Note that the 3 is flanked by two black squares. If the 2s weren't adjacent to the ?s, connecting the 3 to a ? would not violate rule 4, so the 2s can't immediately be dispensed with.

B.6 The Sidesplit gadget

We present the Sidesplit gadget rotated 90° clockwise (S^+), to make comparison with the Split gadget easier. Note in particular how the 3 has moved.

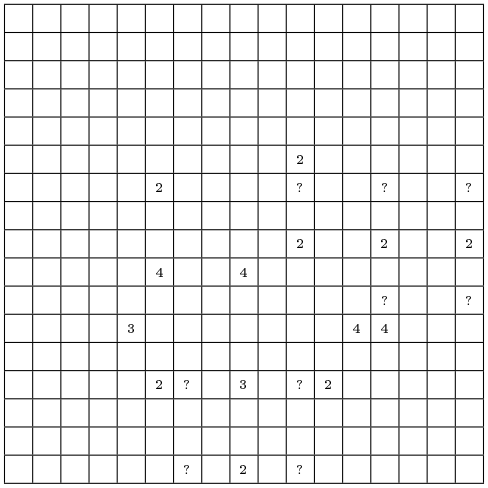


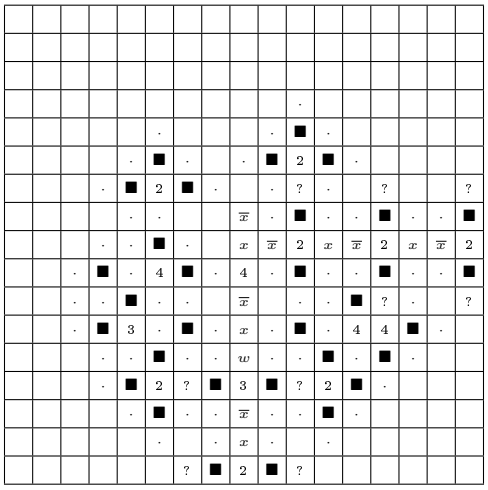


Note that the whiteness of the square below B can be derived independent of B's color. If B is white, then its adjacent 3 has black horizontal neighbours and B's horizontal neighbours are white. This connects the central 4 to more than four white squares, violating rule 4. Thus, B is black.

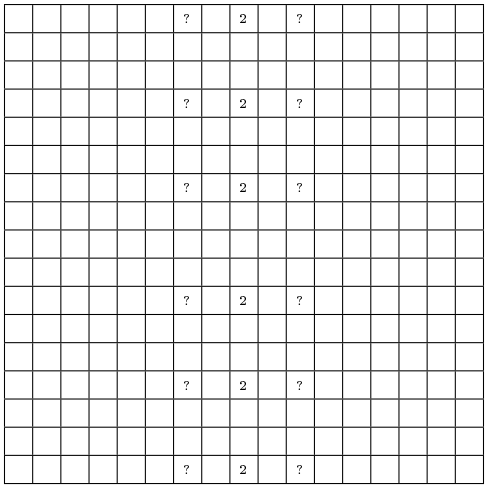
B.7 The Bend gadget

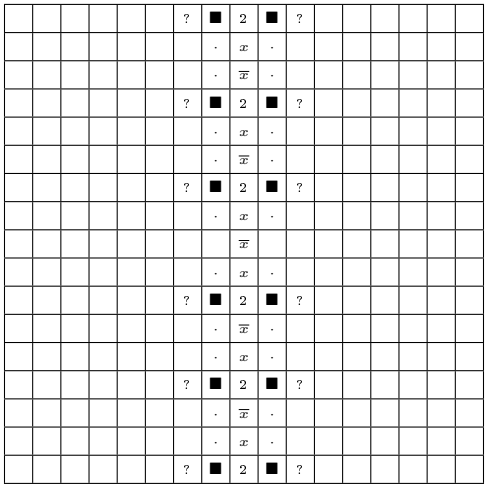
Note the similarity to the Split gadget in the lower and right hand parts.





B.8 The Negation gadget





C Robust Read and Write Protocols Using Information Theoretic MACs

We first specify two extra commands that the protocols in this section will need to assume is available from F_{Rand}:

ShareLabel$(num, label)$ If *ShareLabel* has not called before with *label* as last parameter, for $i = 1, \ldots, num$, choose random $s_i \in \mathbb{F}$, form $[s_i]_{f_i}$, where f_i is random subject to $f_i(0) = s_i$, $deg(f_i) \leq t_p$, and in each point owned by a corrupt player, f_i evaluates to a share chosen by the adversary[17]. Finally, send shares of all *num* values to all honest players, and store $\{s_i\}$ together with *label*.

ShareZero(num, d) For $i = 1, \ldots, num$, form $[0]_{f_i}$, where f_i is random subject to $f_i(0) = 0$, $deg(f_i) \leq d$, and in each point owned by a corrupt player, f_i evaluates to a share chosen by the adversary. Send shares from all *num* sharings to the honest players.

If *ShareLabel* has been called before with *label* as last parameter, retrieve the set of values $\{s_i\}$ stored under label, and secret share these values as described above.

The idea behind the labels is to allow an implementation to generate the shared values pseudorandomly based on the the label and some keys stored by the players.

Implementing *ShareLabel* is harder than for the *Share* command since we have to guarantee that the same shared values are generated when a label is reused. For passively corrupted players, we can use the same protocol described earlier for *Share* if we let players generate the r_i's using a pseudorandom function and based on a locally stored key and a public label. This gives us a computationally secure implementation of the *ShareLabel* command. The *ShareZero* command can be done in a similar way, by just having players choose random sharings of 0, using polynomials of the required degree. If players are actively corrupted, *ShareLabel* can still be done efficiently for a small number of players, using *pseudorandom secret sharing* as proposed in [CDI05]. The protocols from [CDI05] exactly implement what we need for both commands, even without communication. That solution has exponential complexity in n, however, and it is an open problem to implement *ShareLabel* in the malicious case as efficiently as the semi-honest version. A solution for information theoretic macs is to have a key consisting of two random elements $\alpha, \beta \in \mathbb{F}$. The tag for a message $x \in \mathbb{F}$ is now $y = \alpha x + \beta$. A tag is checked by simply verifying that this equation holds. It is easy to see that given x, y, any guess at a different message and tag x', y' is correct with probability $1/|\mathbb{F}|$. If $\mathbb{F}$ is such that this probability is not sufficiently small, we can simply have several tags in parallel

[17] We need to allow the adversary to choose shares for corrupted players, in order to be able to implement the functionality.

with independently chosen keys. Based on this, we show an extended version of the write protocol, which assumes that each server stores, with each block with address *blockaddr* it holds, a counter $c_{blockaddr}$, that is initially 0 and in general is the number of times that block was written. The protocol assumes that $t_p < n/4$.

Protocol $RobustWrite(addrs, blockaddr)$

1. Send "write *blockaddr*" to the servers. Each server returns its $c_{blockaddr}$ value, players decide the correct value by majority and increment the value.

2. Call $ShareLabel(d - \ell + 1 + 2m, (blockaddr, c_{blockaddr}))$ from F_{rand} to get sharings of random values $[r_1], \ldots, [r_{d-\ell+1}], [\alpha_1], \ldots, [\alpha_m], [\beta_1], \ldots, [\beta_m]$. Also call $ShareZero(m, 2t_p)$ to get $[0]_{h_1}, \ldots, [0]_{h_m}$ from F_{Rand}.

3. For $i = 1, \ldots, m$ use the result of Lemma 4 to compute

$$[f(i)] = \sum_{k=1}^{\ell} \lambda_k^i [x_k] + \sum_{k=\ell+1}^{d+1} \lambda_k^i [r_{k-\ell}]$$

   as well as

$$[y_i]_{h_{\alpha_i} h_{f(i)} + h_{\beta_i} + h_i} = [\alpha_i f(i) + \beta_i] + [0]_{h_i}$$

   where $\{[x_k]\}$ are a set of individually shared values pointed to by $addrs$, f is the polynomial from Lemma 4, $y_i = \alpha_i f(i) + \beta_i$ and $h_{\alpha_i}, h_{f(i)}, h_{\beta_i}$ are the polynomials used to share $\alpha_i, f(i), \beta_i$, respectively.

   Note that this only requires local computation, players just multiply their shares from $[\alpha_i], [f(i)]$ and add those from $[\beta_i]$ and $[0]_{h_i}$.

4. For $i = 1, \ldots, m$ each player sends his share of $[f(i)]$ and $[y_i]$ to server D_i.

5. For $i = 1, \ldots, m$ D_i uses the shares received to reconstruct and store $f(i)$ and y_i (under *blockaddr*). Note that this is possible using standard error correction methods since at most $t_p < n/4$ of the shares received will be incorrect and the degree of the polynomial used for sharing y_a is less than $2t_p$. Each server increments $c_{blockaddr}$.

This is secure by essentially the same proof as the passively secure version. The additional observation needed is that the addition of a random sharing of 0 ensures that server D_i will see only a random sharing of y_i of degree at most $2t_p$. Furthermore any authenticator y_i is uniform in the view of the adversary because the keys are unknown. It is therefore easy to simulate the authenticators and shares of them that corrupt servers receive. The following protocol for reading uses a global variable u which points to one of the players and is initially set to 1.

Protocol $RobustRead(addrs, blockaddr)$

1. Call $ShareLabel(2m, label)$ where $label$ is any unused value, to get sharings of values $[v_i], [w_i], i = 1, \ldots, m$. Each player sends "read $blockaddr$" to the servers, as well as his shares of v_i and w_i.

2. Each server D_i retrieves $f(i)$ and y_i using $blockaddr$, reconstructs v_i, w_i (using error correction if necessary) and informs the players on how to modify the sharings of v_i, w_i into sharings of $f(i), y_i$, i.e., he sends $\tau_i = f(i) - v_i$, $\sigma_i = y_i - w_i$ and the current value of $c_{blockaddr}$ to each player. It is actually only necessary to send τ_i, σ_i to P_u, but this way servers do not have to be aware of the value of u.

3. Each player decides on the value of $c_{blockaddr}$ by majority decision, and P_u broadcasts the other values he received: $(\tau_i, \sigma_i)_{i=1,\ldots,m}$. Players adopt the values from P_u as (τ_i, σ_i).

4. Players call $ShareLabel(d - \ell + 1 + 2m, (blockaddr, c_{blockaddr}))$ to get sharings of random values $[r_1], \ldots, [r_{d-\ell+1}]$, $[\alpha_1], \ldots, [\alpha_m]$, $[\beta_1], \ldots, [\beta_m]$ (where only the α's and β's will be used here).

5. For $i = 1, \ldots, m$, the players compute by local computation

$$\sigma_i + [w_i] - (\tau_i + [v_i])[\alpha_i] - [\beta_i] = [y_i - \alpha_i f(i) - \beta_i]$$

   and send all shares to P_u who reconstructs and checks which values are 0. He broadcasts the set S_u of those i where the value was 0 (this saves the cost of opening to everyone).

6. If the size of S_u is not at least $m - t_s$, disqualify P_u, set $u = u+1$ and restart the *RobustRead* protocol. Else call $Share(1)$, to get a random shared value $[x]$. Open x by each player sending his share to all other players, who reconstruct using error correction. Let $v_{i_b} = y_{i_b} - \alpha_{i_b} f(i_b) - \beta_{i_b}$ be the values in S_u that P_u claims are all 0, for $b = 0, \ldots, |S_u| - 1$. Players now compute by local operations

$$[y] = \sum_b x^b [v_{i_b}]$$

   and open y. If $y \neq 0$, disqualify P_u, set $u = u + 1$ and restart the *RobustRead* protocol.

7. For $k = 1, \ldots, \ell$, the players compute locally $[x_k] = \sum_{i=1}^{m} \delta_i'^k [f(i)]$ where $\delta_i'^k$ is a modified set of interpolation coefficients such that $\delta_i'^k = 0$ if i is not in S_u, and the rest is set such that the correct value for x_k is computed. This is possible because we set the parameters such that there are enough honest servers to reconstruct only from their data. This still allows t_s to be in $\Theta(m)$.

The *RobustRead* protocols can be shown to securely implement the *Read* command using a similar technique as for the passive version. Some additional

facts are needed, however: Since a majority of the servers is honest, it is guaranteed that the value used for $c_{blockaddr}$ in a read is indeed the value set in the last write command. Therefore players will use the correct keys for the authenticators, and since the keys α_i, β_i are information theoretically hidden from the adversary, a corrupt server can return a correct y_i for an incorrect $f(i)$ with only negligible probability. A corrupt P_u can attempt to send incorrect values of τ_i, σ_i, even if D_i is honest, but this also requires breaking the MAC scheme. Furthermore, the verification that the shared values $[v_{i_b}]$ are all 0 works by an observation from [DN07]: we think of the v_{i_b}'s as coefficients in a polynomial which, if it is not 0, can have at most $|S_u| - 1$ roots. Therefore if not all u_i are 0, the check passes with probability $(|S_u|)/|\mathbb{F}|$ [18]. It follows that if S_u is as large as required, we have a sufficient number of correct shares from servers to to the interpolation in the last step.

We note that if players are only passively corrupted, the above protocols can be simplified in a straightforward way (servers need no error correction, and we always trust a player to verify the test values for the macs) and will then tolerate $t_p < n/2$. Hence we have:

Theorem 21. *Together with the standard techniques for implementing the input, open and arithmetic operations, RobustRead and RobustWrite form a statistically secure implementation of $\mathcal{F}$ in the F_{Rand}-hybrid model for an adversary corrupting at most $t_p < n/4$ players and $t_s \in \Theta(m)$ servers actively, or an adversary corrupting at most $t_p < n/2$ players passively and $t_s \in \Theta(m)$ servers actively. Each call to RobustRead or RobustWrite involves communication of $O(nm)$ field elements and $O(nm^2)$ local field operations.*

D Security of the Scalable Robust Protocols

We want to show that the *ScalableRobustRead* and *ScalableRobustWrite* protocols, together with standard techniques for implementing input, output and arithmetic operations, form an implementation of $\mathcal{F}$ that is computationally secure against t_p statically actively corrupted players and t_s adaptively actively corrupted servers, under the assumption that the adversary cannot solve discrete logarithm problems and cannot break Schnorr signatures.

We can think of $\mathcal{F}$ as an extension of a simpler functionality for secure function evaluation, one which doesn't have read and write operations. We assume that we already have a protocol for implementing the simpler functionality, which might be [BOGW88], [DN07] or [BTH08]. We have then extended this basic protocol with the read and write protocols. To show that this implements $\mathcal{F}$, we show how to extend the simulator for the basic protocol, turning it into a simulator for $\mathcal{F}$. We assume the base simulator works by picking arbitrary inputs for the honest players and simulating a protocol execution by simply

[18] If $\mathbb{F}$ is not large enough for this to be negligible, we can choose x in an extension field of $\mathbb{F}$ and do the check there.

running the protocol where the simulator plays for the honest players. This is indeed the case for the examples mentioned.

As initialization, the simulator chooses a random key pair for the Schnorr signature scheme (α, β) with $\beta = \alpha^a$ and (g, h) for the Pedersen commitment scheme. Then, it simulates the main protocol, and behaves as detailed below for each invocation of the read and write protocol, respectively.

The write protocol

To simulate a single execution of the write protocol, the simulator proceeds as follows:

Leak "begin write at *blockaddr*" to each corrupted server, then select uniformly random values for s_i, v_i, u_i for each server i and a random value for x, and a set of random values $r_1, \ldots, r_{d-\ell+1}$. Record the adversary's choice of shares of $[v_i]$, $[u_i]$ and $[r_k]$ for the corrupt players (as allowed by F_{Rand}), and compute a set of share consistent with the adversary's choices.

Note the adversary indirectly can choose the values of $[s_i]_j$ for each corrupt player j. Note also that the "shares" s_i don't form a consistent block sharing, but that the adversary will only ever see unqualified sets of them.

If player u is honest, leak $s_i - u_i$ towards each corrupt player and a set of shares of x consistent with its chosen value and what the adversary knows. On the other hand, if player u is corrupted, open $[s_i - u_i]$ towards him and await his reply. If the check value is non-zero, leak shares of $[s_i - u_i]$ towards each corrupted player and set $u = u + 1$. In either case, the simulator computes c_i, chooses $r_1, \ldots, r_m$ uniformly at random, computes γ_i and $[\delta_i]_j$ for each server i and honest player j, and leaks "complete write at *blockaddr* using $([s_i], [v_i], [\delta_i], \gamma_i)$" towards each corrupt server.

The leaked data is x, $s_i - u_i$ for every i and $([s_i], [v_i], [\delta_i], \gamma_i)$ for every corrupt server i. Note that $s_i - u_i$, v_i and x are uniformly randomly distributed and that since the adversary only sees an unqualified set of values of s_i and v_i, their distribution is consistent with any block of values $x_1, \ldots, x_\ell$ and each such block is equally likely given what the adversary knows. Observe also that the signatures on c_i are distributed as in the real world as they are generated as in the real world, and that the adversary is unable to cheat the honest players (i.e. the simulator) into computing a wrong set of values—the worst the adversary can do is impose some extra communication cost.

Note that the simulator can always compute or store what each server knows; if the adversary corrupts any server, simply leak its stored data to the adversary.

The read protocol

To simulate an execution of the read protocol, the simulator proceeds like this:

Leak "read at *blockaddr* towards u" to each corrupt server. Compute a set of shares of $[s_i]$ and $[v_i]$ consistent with the values most recently stored by an execution of the write protocol and consistent with what the adversary chooses for the corrupt servers. If player u is corrupted, leak γ_i, δ_i and c_i for each honest server i.

Invent a set of uniformly random values $b_{x,i}$ and $b_{y,i}$ for $i = 1, \ldots, m$, let the adversary choose the shares of corrupted players, and compute a set of consistent shares $[b_{x,i}]$ and $[b_{y,i}]$. If P_u is not corrupted, leak $(\gamma_i, \delta_i, c_i, x_i, y_i)$ for enough values of i (i.e. at least $m - t_s$) to each corrupt player. Otherwise, leak $[s_i - b_{x,i}]$ and $[v_i - b_{y,i}]$ from every player towards P_u and await a response. If the response is invalid (either malformed or malsized), increment u and restart the simulation. Else, the simulator is done.

Once again, in reaction to the adversary corrupting a server, simply leak all its stored state; as t_s is small, this tells the adversary nothing). Observe that since the $b_{x,i}$ and $b_{y,i}$ are uniformly random, $s_i - b_{x,i}$ and $v_i - b_{y,i}$ have the same (uniform) distribution as in the real world. All other data was chosen during the write protocol; we have already argued this to be distributed just as in the real world. If we assume the adversary is unable to break commitments and unable to forge signatures, he cannot make the honest players accept any value except s_i as any server's block share, and since he is unable (by assumption) to corrupt more than t_s servers, an honest P_u will always have at least $m - t_s$ correct tuples to demonstrate the validity of. In short, the adversary cannot make the honest players behave or compute incorrectly.

E A Simulation Proof of Security of the $share^+$ Protocol

We want to prove that the $share^+(num)$ protocol is secure against a polynomial time rushing adversary who corrupts less than a third of the players, statically. Recall that it generates num random shared values $[r_1], \ldots, [r_{num}]$ and also sends α^{r_i} to every player for $i = 1, \ldots, num$.

The protocol actually generates random values in chunks of length $m(n-t_p)$. For simplicity we therefore show how to simulate the generation of a single chunk.

Assume by renumbering that the adversary corrupts players $1, \ldots, t_p$. He sees an unqualified set of shares in $r_1, \ldots, r_k$ and $\alpha^{r_1}, \ldots, \alpha^{r_k}$, where we let $k = m(n - t_p)$ for brevity. The simulator has to use the adversary's outputs to compute the values leaked to him in the protocol; these are $\alpha^{r_b^a}$ for all a and b, an unqualified subsets of shares in the r_b^a's and the actual values of r_b^a for corrupt P_a, and finally some public values (x and y).

The simulator can choose the public values r_b^a for corrupt players and unqualified subsets of shares uniformly at random, as in the real world; the challenge is to generate matching values $\alpha^{r_b^a}$ for the honest players P_a. Now note that if we can compute the r_b^a's as linear functions of $r_1, \ldots, r_k$ we are in business: then we can apply the linear function (using the homomorphic properties of exponentiation), to $\alpha^{r_1}, \ldots, \alpha^{r_k}$, to obtain $\alpha^{L_b^a(r_1, \ldots, r_k)}$ for all a and b, where $L_b^a(\cdot)$ is the function computing r_b^a.

So assume $r_1, \ldots, r_k$ are given, as we now need to refer to the protocol description, we will again call them s_b^a for $a = 1, \ldots, n - t_p$ and $b = 1, \ldots, m$,

where s_b^a is the a'th entry of $M \cdot V_b$, as per the protocol description. That is,

$$s_b^a = \sum_{i=1}^{n} M_{a,i} r_b^i \quad \text{or equivalently} \quad \begin{pmatrix} s_b^1 \\ \vdots \\ s_b^{n-t_p} \end{pmatrix} = M' \cdot \begin{pmatrix} r_b^1 \\ \vdots \\ r_b^n \end{pmatrix},$$

where M' consists of the $n - t_p$ topmost rows of M, the public hyperinvertible matrix. Let K be the matrix consisting of the columns $t_p + 1, \ldots, n$ of M'. As M is hyperinvertible, K must be invertible. Let $S_b^a = s_b^a - \sum_{i=1}^{t_p} M_{a,i} r_b^i$; this is the output "residue" after having figured in the parts from the corrupt players. Then

$$\begin{pmatrix} S_b^1 \\ \vdots \\ S_b^{n-t_p} \end{pmatrix} = K \cdot \begin{pmatrix} r_b^{t_p+1} \\ \vdots \\ r_b^n \end{pmatrix} \quad \text{and so} \quad \begin{pmatrix} r_b^{t_p+1} \\ \vdots \\ r_b^n \end{pmatrix} = K^{-1} \cdot \begin{pmatrix} S_b^1 \\ \vdots \\ S_b^{n-t_p} \end{pmatrix}.$$

Note that S_b^a can be computed as a linear function of the s_b^a's and data the simulator knows. By composing these linear functions with the one defined by K^{-1}, we get a linear function we can apply to the $\alpha^{s_b^a}$'s which, as promised above, yields $\alpha^{r_b^a}$ for all players P_a

This shows that we can simulate the adversary's view of the protocol perfectly. Hence the only way to distinguish the real from the ideal process is if the input/output behavior of honest players differs in the two cases. In the ideal case, honest players always get shares that are consistent with the public $\alpha^{s_b^a}$ values. But this also happens except with negligible probability in the real protocol. This follows from the fact that the checks done in steps 2e of *share+* and step 3 of *AmortizedExp* fail with high probability unless the honest players have shares of the correct values:

Observe that in step 2e, we compute y_a by evaluating a polynomial p_r with coefficients r_b^a in a random point x_a. Now, no matter how the χ_b^a were computed, they can be written as $\alpha^{z_a^b}$ for some z_a^b, and we can define a polynomial p_z with coefficients $z_b^a, b = 0, \ldots, m$. Now, the χ_b^a are all correct exactly if $p_r = p_z$, and checking the equation in step 2e is equivalent to verifying that $p_r(x_a) = p_z(x_a)$. Since the polynomials have degree m, if $p_r \neq p_z$, they can agree in at most m points, and hence the check goes through in this case with probability at most $m/|\mathbb{F}|$ which is negligible. Correctness of the check in step 3 of *AmortizedExp* can be argued in a similar way.