A Multiparty Computation for Randomly Ordering Players and Making Random Selections

Latanya Sweeney and Michael Shamos
Data Privacy Laboratory
School of Computer Science
Carnegie Mellon University
{latanya, shamos}@privacy.cs.cmu.edu

Abstract

Consider a set of players who wish to randomly arrange themselves without a trusted third-party. For example, if there are 3 players, a, b and c, then a trusted third party could order them as abc, acb, bac, bca, cab, or cba. In the absence of a trusted third party, the players want to select one of these permutations for themselves at random. In this writing, a protocol (named "RandomSelect") is presented using multiplayer computation. From a bag of all possible ways the players could be ordered, RandomSelect provides a means for players to make local choices that when combined, jointly select a permutation randomly. The RandomSelect protocol supports any number (n) of two or more players and computes properly even if n-1 players collude. Communication is O(n) using a broadcast channel. More generally, necessary and sufficient conditions for a class of functions called "RandomOrder" functions are defined. A RandomOrder function uses n inputs to make a random selection of a string from a bag of n! strings where all possible selections are uniformly distributed over the possible inputs and over the strings. Any RandomOrder function can be used in the RandomSelect protocol. Bio-terrorism surveillance is used as an example application.

Keywords: random selection, randomized protocol, randomized multiparty computation, multiparty computation, randomness

1. Motivation and Background

This work presents a multiparty protocol named RandomSelect in which a set of players make local choices that combine to jointly make a random selection from a uniform distribution of choices in the absence of a trusted third party and in the face of colluding players. The reader familiar with multiparty computation and not in need of a motivating application may elect to skip ahead to section 2 to begin discussions of protocol methods.

1.1. Secure Multiparty Computation

This section describes the basic tenets of multiparty computation pertinent to this work. For general reference, see [1, 2].

1. In "distributed computing" a number of networked players carry out a joint computation of a function on their inputs. The aim of "secure multiparty computation" (or simply, **multiparty computation**), in contrast, is to enable players to carry out distributed computing tasks on their private information while under attack by an external entity ("the adversary") and/or by a subset of malicious players ("the colluding players"). The purpose of the attack is to learn the private information of non-colluding, honest players or to cause the computation to be incorrect. As a result, there are two important requirements of a multiparty computation protocol: privacy and correctness.

2. The **privacy requirement** of a multiparty computation states that information deemed private should not be learned.

3. The **correctness requirement** states that the result of a multiparty computation should be correct. The adversary or colluding players must not be able to cause the result of the computation to deviate from the function that the players had set out to compute.

4. It is assumed throughout that when a message is sent, it arrives before some time bound. More specifically, a protocol proceeds in "rounds." In each round, a player may send a message to other players, and all messages are delivered before the next round begins.

5. The work presented in this writing assumes that each player can send a message to all other players using a **broadcast channel**. It cannot usually be verified that any player actually received the broadcast message however. An example of a broadcast channel is a satellite in which messages sent to the satellite during one round are transmitted in the next round so that any parties seeking to receive the messages can do so. Low-level infrastructure in computer networks can also provide a broadcast channel.

6. It is sometimes convenient to consider operation in which each pairs of players can communicate secretly, point-to-point, over perfectly private and authenticated channels (called **secure channels**). Clearly, secure channels can be used for broadcasting by having a player send a message directly to every other player. In this case, each player's receipt of the broadcast can be verified.

7. A colluding player is one that may misbehave "passively" or "actively." **Passive corruption** means that the adversary and colluding players obtain the complete information held by corrupted players, but all players execute the protocol correctly. **Active corruption** means that the adversary and colluding players may misbehave in an arbitrary manner.

8. Another assumption made throughout this work is that the adversary and colluding players are "static." Players can be corrupted and become colluders before the protocol starts, but during protocol execution, non-colluding players cannot be corrupted.

9. It is assumed by the use of secure channels that the adversary has bounded computing power. [3] introduced a general protocol that allows n players to securely compute an arbitrary function even if an adversary actively corrupts any $t < n/2$ of the players and makes them misbehave maliciously. In a model with secure channels, [4, 5] proved that unconditional security is possible if at most $t < n/3$ of the players are actively corrupted. This bound was improved by [6, 7] to $t < n/2$ by additionally using a broadcast channel.

10. In this work the notion of "private information" as it relates to a multiparty computation refers to a player not revealing his local choice prior to broadcasting that choice to all players. Collusion refers to a subset of players sharing their choices prior to broadcast or more generally, not making random local decisions. For n players, a random selection of a permutation of the players must compute properly even of n-1 players collude.

1.2. Example of Trivial Secret Sharing

1. A protocol to randomly select a permutation of players may be useful within a larger protocol that solves a particular task. Details of bio-terrorism surveillance as a sample application appear in Appendix A. A brief summary appears in this section. Trivial secret sharing, a well-known multiparty computation, is compared to a randomized version in which players make joint random choices. The purpose is to identify efficiency tradeoffs afforded by randomization.

2. The problem addressed in this sample application is as follows. A set of players and a reporting authority operate over secure channels. Each player holds a private number. The players wish to compute the sum of their numbers in a manner that preserves the secrecy of their individual values while providing the sum to the reporting authority, even if some of the players collude passively.

3. Trivial secret sharing offers a solution. For n players, each player's private number is represented as a set of n values (called "shares") that sum to the player's original private number. Each pair of players exchanges shares. Then, each player adds the shares he receives with his remaining share and sends his total to the reporting authority, who adds the totals received from each player and announces the final sum.

4. An alternative solution involves randomly arranging n players in m rings, where $1 \leq m \leq n$. Each ring has all n players, but the order in which a player appears around any ring is determined randomly. Each player has m shares that sum to her private value, and contributes one share to each ring as follows. The reporting authority provides a random number to the first players of each ring, who in turn add one share and send their sums to the second players, who add one share, and so on, around the rings. The last players send their sums to the reporting authority, which adds them and subtracts the random numbers provided originally to get the overall total.

5. For n players with each player having $1 \leq m \leq n$ shares, the number of communications improved with randomization from $O(n^2)$ to $O(m \bullet n)$, but the randomized version assumes random orderings of players can be done within this bound. Providing a method for players to jointly select random orderings of themselves within $O(n)$ communication is the subject of the remainder of this writing. Doing so completes the randomized solution in this sample application.

6. The broader goal of this work is to introduce the notion of a "randomized multiparty protocol" as a multiparty protocol in which a subset of 2 or more players jointly make a random choice based on a given probability distribution.

2. General Approach

Ways in which *n* players might jointly select a random permutation of themselves are informally explored in this section by examining a game (Section 2.1), an approach using exclusive-OR (Section 2.2), and an approach using modulus and addition (Section 2.3). This section provides an informal preamble to the formal methods of Section 3 by sketching out the basic requirements desired.

2.1. The Rock Paper Scissors Game

1. Inspiration for a method to randomly order players was found in the childhood game known as "Rock Paper Scissors," which is commonly used to decide who goes first [12]. The game is played with only two players. Prior to play, a set of token gestures known as rock, paper, and scissors are understood. A hand in a fist represents rock, a horizontal flat hand represents paper, and two horizontal fingers represent scissors. These tokens are circularly ordered such that rock beats scissors, scissors beats paper, and paper beats rock in determining which player goes first. Play consists of synchronous display of a token gesture by each player. Figure 1 shows the possible outcome for players Alice and Bob. Notice how a player cannot select a token that assures a resulting order. For example, Alice displaying rock can have her appear first if Bob displays scissors, or last if Bob displays paper. Both players' tokens in combination determine the final ordering. When both players select the same token, a "collision" occurs and no order can be determined. In that case, the players perform additional rounds as needed until no collision is encountered.

Alice	Bob	Outcome
rock	rock	*none*
rock	paper	Bob, Alice
rock	scissors	Alice, Bob
paper	rock	Alice, Bob
paper	paper	*none*
paper	scissors	Bob, Alice
scissors	rock	Bob, Alice
scissors	paper	Alice, Bob
scissors	scissors	*none*

Figure 1. Possible outcome sequences to determine who goes first and who goes second, Alice then Bob or Bob then Alice, based on the Rock Paper Scissors game.

2. The work reported herein expands the notion of Rock Paper Scissors for use with an arbitrary number of players (two or more) operating over a network to achieve the following. Each player is given a set of tokens from which to draw. Collisions are avoided. Just as occurred in the original game, the assignment of tokens to outcomes is provided to each player beforehand with uniform distribution of outcomes assigned to token choices. During a round of play, each player provides a token. The outcome is a permutation of the players selected by the joint choices of tokens.

2.2. Basic math notation used

1. Before informally exploring two ad hoc extensions to Rock Paper Scissors, some basic bath notation must be presented.

2. Given an integer *b*, the "factorial of *b*" is the product: $b \bullet (b\text{-}1) \bullet \ldots \bullet 1$, and is written *b*!.

Example. The factorial of 6 (written 6!) is $6 \bullet 5 \bullet 4 \bullet 3 \bullet 2 \bullet 1 = 720$.

3. Given integers *a*, *b* and *r*, if *a* divided by *b* has remainder *r*, then "*r* is congruent to *a* modulo *b*," which is written $r \equiv a \bmod b$. If *r* is 0, then "*a* is divisible by *b*," which is written $b|a$, and there exists an integer *d* such that $a = b \bullet d$.

Example. The integer 6 divided by 3 has a quotient of 2 and a remainder of 0. This is denoted by the following expressions $0 \equiv 6 \bmod 3$ and $3|6$.

2.3. Using Exclusive-OR

1. Consider an extension of Rock Paper Scissors to $n \geq 2$ players using bitwise exclusive-or (denoted $x \oplus y$) to jointly select a possible permutation. The exclusive-OR of two 1-bit values *x* and *y* is 1 if exactly one of *x* and *y* is 1 and is 0 otherwise. If *x* and *y* are *r*-bit numbers, then the 1-bit exclusive-or operation is performed on corresponding bits in the two values, respectively.

2. Here is a protocol named "RandomXOR" that uses bitwise exclusive-or. In a preliminary round, all possible permutations of the players are enumerated. A table containing these enumerations (or an equivalent algorithm) is distributed to each player prior to play commencing. Each player receives the same table. Rounds of play utilize the same table with no further distribution of a table needed. The protocol has one round. Each player P_i selects a number v_i (token), where v_i is between 1 and the number of permutations, and broadcasts the pair (P_i, v_i). Each player then performs bitwise exclusive-or to determine the joint selection (*s*), where $s = v_1 \oplus \ldots \oplus v_n$. The permutation associated with *s* in the table is the selected permutation.

Example. Given *n*=3 players, a, b and c, RandomXOR can be used to select one of the 6 possible permutations of {a, b, c} as follows. Before any rounds of play commence, the permutations are associated with the values 1 to 6 as shown in Figure 2(a) and distributed to each player. The players use this table in all rounds of play. To make a random selection, each player selects a value between 1 and 6. Suppose Player a selects v_a=3, Player b selects v_b=2 and Player c selects v_c=5, then $v_a \oplus v_b \oplus v_c = 3 \oplus 2 \oplus 5 = 4$. This computation is detailed in Figure 2(b). The permutation associated with 4 in Figure 2(a) is acb, so acb is the joint selection.

1. cba
2. bca
3. cab
4. acb
5. bac
6. abc
(a)

t_a = 3 = 011
t_b = 2 = 010
t_c = 5 = 101
s = 100 = 4 →acb

(b)

Figure 2. Enumeration of permutations for 3 players (a, b, and c) is shown in (a). An example selection of acb using bitwise exclusive-OR is shown in (b) where player a selects 3 (bit pattern 011), b selects 2 (bit pattern 010), and c selects 5 (bit pattern (101). The result is acb because $3 \oplus 2 \oplus 5 = 4$ (bit pattern (100), and 4 is assigned the permutation acb in (a).

3. Some selections by players in RandomXOR can result in values not listed in the enumerated permutations. These are considered collisions and require re-plays to occur until no collision is encountered.

Example. Given n=3 players, a, b and c, suppose RandomXOR is used to select one of the 6 possible permutation enumerated in Figure 2(a). Let Player a select v_a=3, Player b select v_b=1 and Player c select v_c=5, then $v_a \oplus v_b \oplus v_c = 3 \oplus 1 \oplus 5 = 7$, which has no associated permutation. The combination of choices, (3, 1, 5), provides an undeterminable result.

4. Players can collude in RandomXOR to influence the final result so that some outcomes are less likely than others.

Example. Given n=3 players, a, b and c, let RandomXOR be used to select one of the 6 possible permutation enumerated in Figure 2(a). Suppose Players b and c collude to limit the placement of Player a so that either Player a or b appears last. Player b agrees to always select v_b=2 and Player c agrees to always select v_c=1. If Player a selects 1 or 2, then permutations cba and bca result in which Player a appears last. If Player a selects 3 or 4, a collision results. Finally, if Player a selects 5 or 6, then permutations bac or abc result in which Player c appears last. See the highlighted entries in Figure 23(a).

5. While RandomXOR provides many desirable characteristics, problems with collisions and with colluding players exist. The nature of these shortcomings will be analyzed in Section 5 after formal methods are introduced in Sections 3 and 4.

2.4. Using the Modulus of the Sum

1. Consider using the modulus (mod) of the sum of player choices as a way to jointly select a possible permutation for $n \geq 2$ players. Here is a protocol named "ModSum" that operates very similar to RandomXOR. In a preliminary round, each of the permutations of the players are enumerated and distributed to each player. For n players, there are $n!$ permutations. The protocol has one round. Each player P_i selects a number v_i, where $0 \leq v_i < n!$, and broadcasts the pair (P_i, v_i). Each player then sums the broadcasted values and uses the mod of the sum divided by $n!$ to determine the joint selection (r); that is, $r = (\sum v_i) \bmod n!$. The permutation associated with r in the table is the selected permutation.

Example. Given n=3 players, a, b and c, let ModSum be used to select one of the $n! = 6$ permutations of {a, b, c}. Before any rounds of play, the permutations are associated with the values 0 to 5 as shown in Figure 2(a) and distributed to each player. To make a random selection, each player selects a value between 0 and 5. Suppose Player a selects v_a=3, Player b selects v_b=2 and Player c selects v_c=5, then $v_a + v_b + v_c = 3 + 2 + 5 = 10$, and 10 mod 6 is 4. The permutation associated with 4 in Figure 2(a) is bac, so bac is the joint selection.

2. All selections based on ModSum provide a legal selection with no collisions because the modulus operation guarantees that the final result will be in the range 0 to ($n!$-1). Recall selection $r = (\sum v_i) \bmod n!$, where v_i is the choice of each player, $0 \leq v_i < n!$, and $n!$ is the total number of permutations. Each play yields a selection $0 \leq r < n!$.

3. Unlike RandomXOR above, colluding players gain no advantage using ModSum. (This will be examined further in Section 5 after formal methods are introduced.)

Example. Let n=3 players, a, b and c, use ModSum to select one of the 6 possible permutation enumerated in Figure 2(a). Suppose Players b and c collude. Player b agrees to always select v_b=2 and Player c agrees to always select v_c=1. The final outcome is equally likely to be any of the permutations depending on the selection made by Player a. If v_a=0, acb results; if v_a=1, bac results; if v_a=2, abc results; if v_a=3, cba results; if v_a=4, bca results; and, if v_a=5, cab results.

4. The informal discussions in this section introduced the basic requirements for a protocol that allows n players to make local choices that combine to make a joint random selection with uniform distribution of outcomes to choices. There should be no collisions, and the protocol should perform properly in the face of n-1 colluding players. In the next section, formal definitions and sufficient and necessary conditions are provided for acceptable solutions. In Section 5, after the formal presentation of Section 3, attention returns to RandomXOR and SumMod for formal assessment.

3. Model and Definitions

This section and the next provide a formal presentation of the RandomSelect protocol. First, general terms and notation (Section 3.1) are introduced. In Section 3.2, conditions for choosing values appropriately are investigated. In Section 4, the class of RandomOrder functions is defined, the RandomSelect protocol is presented, and effects of collusion are examined.

3.1. General terms and notation

1. Let $P=\{P_1, ..., P_n\}$ be the n players seeking to be arranged randomly, where $n \geq 2$. There must be at least 2 players.

Example. P={a, b, c} represents the 3 players a, b, and c.

2. Let $O=\{o_1, ..., o_{n!}\}$ be the $n!$ distinct permutations of size n in which the players of P may be arranged. Each o_i permutation has no repetition of players. These o_i's are termed the **sequences** of P. The idea is that a permutation $P_x,\ldots,P_z$ expresses the random ordering in which the players of P will be arranged. P_x will appear first, and so on, with P_z appearing last.

Example. Given P={a, b, c}, then the sequences of P are O={abc, acb, bac, bca, cab, cba}. There are 3 players and 3!=3•2=6 permutations of size 3. Each permutation contains all players and each player appears only once in a permutation. The sequence abc has player a appearing first, player b appearing second, and player c appearing last, whereas the sequence cba has player c first, player b second and player a last.

3. Each player P_i has k_x **token** values, $\{v_{i1}, \ldots, v_{ix}\}$. The idea is that player P_i will select one of its k_x tokens as its contribution to determining the final random arrangement of players. For generality, assume each value v_{iq} , where $1 \leq q \leq x$, is unique across all players; that is, unless otherwise noted, for each P_i, P_j, there does not exist v_{iq} and v_{jr} such that $v_{iq}=v_{jr}$.

Example. In Figure 3, player a has tokens a_1 and a_2 from which to choose; player b has choices b_1, b_2, and b_3, and player c has choices c_1 and c_2. Notice that no token is the same for any two players in this example.

Player	Tokens from which to choose
a	$\{a_1, a_2\}$
b	$\{b_1, b_2, b_3\}$
c	$\{c_1, c_2\}$

Figure 3. Choices for players a, b, and c are the tokens a_1 and a_2 for player a; b_1, b_2, and b_3 for player b; and, c_1 and c_2 for player c.

4. The number of possible ways *n* players can jointly make selections by each choosing one of its tokens is Πk_i, for i=1,.., n. Each such way is termed a **configuration**, and is written as an n-tuple $(v_{1x}, \ldots, v_{ny})$ where v_{1x} is Player 1's chosen token and v_{ny} is Player n's chosen token.

Example. Given choices for a to be $\{a_1, a_2\}$, choices for b to be $\{b_1, b_2, b_3\}$, and choices for c to be $\{c_1, c_2\}$, there are 2•3•2=12 possible configurations. These are itemized in Figure 4.

1. $(a_1\ b_1\ c_1)$
2. $(a_1\ b_2\ c_1)$
3. $(a_1\ b_3\ c_1)$
4. $(a_1\ b_2\ c_2)$
5. $(a_1\ b_1\ c_2)$
6. $(a_1\ b_3\ c_2)$
7. $(a_2\ b_1\ c_1)$
8. $(a_2\ b_2\ c_1)$
9. $(a_2\ b_3\ c_1)$
10. $(a_2\ b_2\ c_2)$
11. $(a_2\ b_1\ c_2)$
12. $(a_2\ b_3\ c_2)$

Figure 4. Possible configurations for players a, b, and c where choices for a are $\{a_1, a_2\}$, choices for b are $\{b_1, b_2, b_3\}$, and choices for c are $\{c_1, c_2\}$

5. Configurations are **assigned** to sequences. The idea is that the players jointly determine a configuration based on their individual choices of values; then, the sequence, which is associated with that configuration, is the final randomized arrangement of players.

Example. Given the possible configurations in Figure 4 and the sequences {abc, acb, bac, bca, cab, cba}, Figure 5 provides an assignment of configurations to sequences. Other assignments are possible.

In the example provided in Figure 5, the configuration $a_1b_1c_1$ is assigned to the sequence abc. The idea is that if player a chooses a_1, player b chooses b_1 and player c chooses c_1, then the sequence abc results. But, if instead player a chooses a_2, player b chooses b_3 and player c chooses c_2, then the sequence cba results.

Sequences	Configurations	
abc	$a_1\ b_1\ c_1$	$a_2\ b_1\ c_1$
acb	$a_1\ c_1\ b_2$	$a_2\ c_1\ b_2$
bac	$b_3\ a_1\ c_1$	$b_3\ a_2\ c_1$
bca	$b_2\ c_2\ a_1$	$b_2\ c_2\ a_2$
cab	$c_2\ a_1\ b_1$	$c_2\ a_2\ b_1$
cba	$c_2\ b_3\ a_1$	$c_2\ b_3\ a_2$

Figure 5. An assignment of all configurations to all sequences for players a, b, and c where choices for a are $\{a_1, a_2\}$, choices for b are $\{b_1, b_2, b_3\}$, and choices for c are $\{c_1, c_2\}$.

6. A token has a **position** within a sequence that is dictated by the assignment of the configuration to a sequence. If a configuration is not assigned to a sequence, then the set of tokens chosen by the players has no sequence ordering. But once a configuration is assigned to a sequence, the order in which the players appear in the sequence imposes an ordering on the configuration. A token's position in its configuration is the position of the player in the sequence assigned to the configuration. Once assigned, the tokens of the configuration generated by the choices of the n players, $(v_{1i}, \ldots, v_{nj})$, can be written in the order of the sequence, $P_x \ldots P_y$, thereby producing $v_{xs} \ldots v_{yt}$ where v_{xs} has player x's token v_{xs} appearing first, and player y's token v_{yt} appearing last.

Example. If player a chooses a_2, player b chooses b_3 and player c chooses c_1, then the configuration (a_2, b_3, c_1) results. This configuration in isolation provides no ordering. Now assume the assignments shown in Figure 5. The configuration (a_2, b_3, c_1) is assigned to the sequence bac, which means the sequence imposes the ordering *b* then *a* then *c*. The configuration can be written as $b_3a_2c_1$ to identify the tokens chosen and the ordering of the sequence that results.

7. For n players, a sequence is a string of length n. When a configuration is assigned to a sequence, the assigned configuration is a string. The token t in position x of the assigned configuration s is written as $\mathrm{pos}(s, x) = t$, where $1 \le x \le n$.

Example. In Figure 5, the configuration (a_2, b_1, c_2) is assigned to the sequence cab providing the ordered configuration $c_2a_2b_1$. $\mathrm{pos}(c_2a2b_1, 1) = c_2$, $\mathrm{pos}(c_2a2b_1, 2) = a_2$, and $\mathrm{pos}(c_2a2b_1, 3) = b_1$.

8. Given assignments of configurations to sequences, the number of occurrences of a token t in a position x for a particular sequence s is denoted as $\mathrm{count}(s, t, x) = y$, or for the set of all sequences is written as $\mathrm{count}(t, x) = y$.

Example. In Figure 5, $\mathrm{count}(\mathrm{abc}, b_1, 2) = 2$, $\mathrm{count}(\mathrm{abc}, a_1, 1) = 1$, and $\mathrm{count}(\mathrm{abc}, a_1, 2) = 0$. Also, $\mathrm{count}(b_1, 2) = 2$, $\mathrm{count}(a_1, 1) = 2$, and $\mathrm{count}(a_1, 2) = 2$.

3.2. Conditions for Choosing Values Appropriately

Five conditions for appropriately choosing the number of configurations, the number of choices, and the assignment of configurations to sequences are provided in this section. These principles help assure "randomness" in determining the final sequence of players by making sure there is no bias in each of these characteristics.

1. For all possible configurations of n players to be evenly distributed over all sequences, the number of choices for each player (k_i) must be selected such that $0 \equiv (k_1 \bullet \ldots \bullet k_n) \bmod (n!)$. This is identified as Condition 1, "the Configurations Condition."

Condition 1 ("Configurations"). Assignments are performed such that the set of all possible configurations are evenly distributed over the set of all possible sequences. In order to satisfy this condition, it is sufficient and necessary that $0 \equiv (k_1 \bullet \ldots \bullet k_n) \bmod (n!)$.

NUMBER OF PARTIES = 3					NUMBER OF PARTIES = 4					
k1	k2	k3	k1*k2*k3	(k1*k2*k3) mod (n!)	k1	k2	k3	k4	k1*k2*k3*k4	(k1*k2*k3*k4) mod (n!)
1	1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	1	1	1	2	2	2
1	2	2	4	4	1	1	2	2	4	4
2	2	2	8	2	1	2	2	2	8	8
2	2	3	12	0	2	2	2	2	16	16
2	3	3	18	0	2	2	2	3	24	0
3	3	3	27	3	2	2	3	3	36	12
3	3	4	36	0	2	3	3	3	54	6
3	4	4	48	0	3	3	3	3	81	9
4	4	4	64	4	3	3	3	4	108	12
4	4	5	80	2	3	3	4	4	144	0
4	5	5	100	4	3	4	4	4	192	0
5	5	5	125	5	4	4	4	4	256	16
5	5	6	150	0	4	4	4	5	320	8
5	6	6	180	0	4	4	5	5	400	16
6	**6**	**6**	**216**	**0**	4	5	5	5	500	20
					5	5	5	5	625	1
					5	5	5	6	750	6
					5	5	6	6	900	12
					5	6	6	6	1080	0
					6	**6**	**6**	**6**	**1296**	**0**

Figure 6. Different number of choices (k_i) for 3 players (left side) and 4 players (right side). The highlighted rows identify those numbers of choices that satisfy Condition 1.

Example. Consider the sequences and configurations in Figure 5. Notice that the 12 configurations are distributed evenly over the 6 sequences for the 3 players, thereby satisfying Condition 1. The remainder of the number of configurations ($k_1 \bullet \ldots \bullet k_n$) divided by the number of sequences (n!) is 0. Specifically, let r be the remainder: $r \equiv (2 \bullet 3 \bullet 2) \bmod (3!)$. This simplifies to $r \equiv 12 \bmod 6$, so $r = 0$.

Example. In Figure 5, there are 2 ways the players may jointly select any specific sequence. This results because the number of configurations is divisible by the number of sequences, and the quotient is 2. That is, 6|12 and 12/6 = 2.

Example. Consider the sequences and configurations in Figure 5. Notice that if the number of choices for player b was 2 and not 3, then the total number of possible configurations would be 2•2•2=8. But there are only 6 sequences. The configurations cannot be evenly distributed over the sequences in this case, so Condition 1 cannot be satisfied. Some sequences will appear more often despite each player making configuration choices randomly.

Example. Figure 6 shows the relationship of Condition 1, the Configurations Condition, to different numbers of choices for n=3 players (left side) and n=4 players (right side). Each row in Figure 6 identifies the number of choices each player may have (k_i), the product of the number of choices, which is the total number of possible configurations, and the remainder (modulus) of the total number of configurations divided by the total number of sequences (which is the factorial of n). In the fourth row on the left side, k_1=2, k_2=2, and k_3=2. There are 2•2•2 = 8 possible configurations. There are 3!= 3•2 = 6 sequences. Because $2 \equiv 8 \bmod 6$ is number of choices for the players and it is not congruent to 0, this number of choices for players fails to satisfy Condition 1.

Example. Consider the fifth row on the left side of Figure 6. This is the topmost highlighted row. k_1=2, k_2=3 and k_3=3. There are 2•2•3 = 12 possible configurations. There are 3!= 3•2 = 6 sequences and $0 \equiv 12 \bmod 6$. This number of choices for the players therefore satisfies Condition 1; the set of all possible configurations can be evenly distributed over the set of all possible sequences. Each of the highlighted rows in Figure 6 satisfies Condition 1.

2. The number of tokens (k) for each of the n players should be the same and should be more than one. This is identified as Condition 2, "the Choices Condition." If a player has only one token, then the player has no choice. If one player has more tokens (or choices) than the other players, then that player may gain an ability to reliably restrict the final sequence to be drawn from one of a limited number of sequences rather than being equally likely from all possible sequences.

Condition 2 ("Choices")[1]. Each player has the same number of tokens (k), where $k \geq 2$.

Example. Consider the sequences and configurations in Figure 5. Notice the configurations containing a_1. Player a's selection of a_1 cannot improve the likelihood of any particular sequence being determined. This is also true of Player a's choice of a_2. Any selection of a_1 or a_2 by Player a can lead to any final sequence. But unlike Player a, Player b has 3 choices. Player b's selection of b_1 restricts the final sequences to being either abc or cab. In these sequences, b never appears in the 1st position. Likewise, Player b's choice of b_2 restricts the final sequences to acb and bca in which b never appears in the middle position. Finally, Player b's selection of b_3 restricts the final sequences to bac or cba, in which b never appears in the last position. In order for each player to have the same lack of determination over the final sequence, each player should have the same number of choices. The configurations in Figure 5 do not satisfy Condition 2.

Example. Figure 7 lists all 216 possible configurations for 3 players having 6 choices each. There are 6 possible sequences to which each of these 216 configurations could be assigned. If the configurations are evenly distributed over the sequences, there would be 36 configurations per sequence. The selection of k=6 as the number of choices for each of the n=3 players satisfies Condition 1, as shown in this computation of r. Let $r \equiv (6 \bullet 6 \bullet 6) \bmod (3!)$, which simplifies to $r \equiv 216 \bmod 6$, so r=0. The selection of k=6 also satisfies Condition 2 because each of the players has the same number of k=6 choices.

3. The number of tokens (k) for each of the n players should be selected so that the total number of tokens (combining all tokens from all players) is evenly distributed over the number of configurations. In order for this to occur, $0 \equiv \Pi k \bmod \Sigma k$. This is identified as Condition 3, "the Tokens Condition."

Condition 3 ("Tokens"). The set of tokens from all players (having size Σk) should evenly distribute over the set of all possible configurations (having size Πk). In order for this condition to be satisfied, it is sufficient and necessary that $0 \equiv \Pi k \bmod \Sigma k$. Since each of the n players has k choices (Condition 2), this is the same as: $0 \equiv k^n \bmod n \bullet k$.

Example. Figure 7 lists all 216 possible configurations for 3 players having 6 choices each. This selection of k=6 choices for n=3 players satisfies Condition 3, as shown in this computation of r. Let $r \equiv 6^3 \bmod (3 \bullet 6)$, which simplifies to $r \equiv 216 \bmod 18$, so $r = 0$.

Example. Figure 5 lists all 12 possible configurations for 3 players, where two of the players have 2 choices and one player has 3 choices. This selection fails to satisfy Condition 3, as shown in this computation of r. Let $r \equiv (2 \bullet 2 \bullet 3) \bmod (2+2+3)$, which simplifies to $r \equiv 12 \bmod 7$, so r=5, which is not 0.

[1] Condition 2 is updated to include the constraint $0 \equiv k \bmod n!$ in paragraph 10 of this subsection.

1: (a_1 b_1 c_1)	37: (a_1 b_1 c_2)	73: (a_1 b_1 c_3)	109: (a_1 b_1 c_4)	145: (a_1 b_1 c_5)	181: (a_1 b_1 c_6)
2: (a_2 b_1 c_1)	38: (a_2 b_1 c_2)	74: (a_2 b_1 c_3)	110: (a_2 b_1 c_4)	146: (a_2 b_1 c_5)	182: (a_2 b_1 c_6)
3: (a_3 b_1 c_1)	39: (a_3 b_1 c_2)	75: (a_3 b_1 c_3)	111: (a_3 b_1 c_4)	147: (a_3 b_1 c_5)	183: (a_3 b_1 c_6)
4: (a_4 b_1 c_1)	40: (a_4 b_1 c_2)	76: (a_4 b_1 c_3)	112: (a_4 b_1 c_4)	148: (a_4 b_1 c_5)	184: (a_4 b_1 c_6)
5: (a_5 b_1 c_1)	41: (a_5 b_1 c_2)	77: (a_5 b_1 c_3)	113: (a_5 b_1 c_4)	149: (a_5 b_1 c_5)	185: (a_5 b_1 c_6)
6: (a_6 b_1 c_1)	42: (a_6 b_1 c_2)	78: (a_6 b_1 c_3)	114: (a_6 b_1 c_4)	150: (a_6 b_1 c_5)	186: (a_6 b_1 c_6)
7: (a_1 b_2 c_1)	43: (a_1 b_2 c_2)	79: (a_1 b_2 c_3)	115: (a_1 b_2 c_4)	151: (a_1 b_2 c_5)	187: (a_1 b_2 c_6)
8: (a_2 b_2 c_1)	44: (a_2 b_2 c_2)	80: (a_2 b_2 c_3)	116: (a_2 b_2 c_4)	152: (a_2 b_2 c_5)	188: (a_2 b_2 c_6)
9: (a_3 b_2 c_1)	45: (a_3 b_2 c_2)	81: (a_3 b_2 c_3)	117: (a_3 b_2 c_4)	153: (a_3 b_2 c_5)	189: (a_3 b_2 c_6)
10: (a_4 b_2 c_1)	46: (a_4 b_2 c_2)	82: (a_4 b_2 c_3)	118: (a_4 b_2 c_4)	154: (a_4 b_2 c_5)	190: (a_4 b_2 c_6)
11: (a_5 b_2 c_1)	47: (a_5 b_2 c_2)	83: (a_5 b_2 c_3)	119: (a_5 b_2 c_4)	155: (a_5 b_2 c_5)	191: (a_5 b_2 c_6)
12: (a_6 b_2 c_1)	48: (a_6 b_2 c_2)	84: (a_6 b_2 c_3)	120: (a_6 b_2 c_4)	156: (a_6 b_2 c_5)	192: (a_6 b_2 c_6)
13: (a_1 b_3 c_1)	49: (a_1 b_3 c_2)	85: (a_1 b_3 c_3)	121: (a_1 b_3 c_4)	157: (a_1 b_3 c_5)	193: (a_1 b_3 c_6)
14: (a_2 b_3 c_1)	50: (a_2 b_3 c_2)	86: (a_2 b_3 c_3)	122: (a_2 b_3 c_4)	158: (a_2 b_3 c_5)	194: (a_2 b_3 c_6)
15: (a_3 b_3 c_1)	51: (a_3 b_3 c_2)	87: (a_3 b_3 c_3)	123: (a_3 b_3 c_4)	159: (a_3 b_3 c_5)	195: (a_3 b_3 c_6)
16: (a_4 b_3 c_1)	52: (a_4 b_3 c_2)	88: (a_4 b_3 c_3)	124: (a_4 b_3 c_4)	160: (a_4 b_3 c_5)	196: (a_4 b_3 c_6)
17: (a_5 b_3 c_1)	53: (a_5 b_3 c_2)	89: (a_5 b_3 c_3)	125: (a_5 b_3 c_4)	161: (a_5 b_3 c_5)	197: (a_5 b_3 c_6)
18: (a_6 b_3 c_1)	54: (a_6 b_3 c_2)	90: (a_6 b_3 c_3)	126: (a_6 b_3 c_4)	162: (a_6 b_3 c_5)	198: (a_6 b_3 c_6)
19: (a_1 b_4 c_1)	55: (a_1 b_4 c_2)	91: (a_1 b_4 c_3)	127: (a_1 b_4 c_4)	163: (a_1 b_4 c_5)	199: (a_1 b_4 c_6)
20: (a_2 b_4 c_1)	56: (a_2 b_4 c_2)	92: (a_2 b_4 c_3)	128: (a_2 b_4 c_4)	164: (a_2 b_4 c_5)	200: (a_2 b_4 c_6)
21: (a_3 b_4 c_1)	57: (a_3 b_4 c_2)	93: (a_3 b_4 c_3)	129: (a_3 b_4 c_4)	165: (a_3 b_4 c_5)	201: (a_3 b_4 c_6)
22: (a_4 b_4 c_1)	58: (a_4 b_4 c_2)	94: (a_4 b_4 c_3)	130: (a_4 b_4 c_4)	166: (a_4 b_4 c_5)	202: (a_4 b_4 c_6)
23: (a_5 b_4 c_1)	59: (a_5 b_4 c_2)	95: (a_5 b_4 c_3)	131: (a_5 b_4 c_4)	167: (a_5 b_4 c_5)	203: (a_5 b_4 c_6)
24: (a_6 b_4 c_1)	60: (a_6 b_4 c_2)	96: (a_6 b_4 c_3)	132: (a_6 b_4 c_4)	168: (a_6 b_4 c_5)	204: (a_6 b_4 c_6)
25: (a_1 b_5 c_1)	61: (a_1 b_5 c_2)	97: (a_1 b_5 c_3)	133: (a_1 b_5 c_4)	169: (a_1 b_5 c_5)	205: (a_1 b_5 c_6)
26: (a_2 b_5 c_1)	62: (a_2 b_5 c_2)	98: (a_2 b_5 c_3)	134: (a_2 b_5 c_4)	170: (a_2 b_5 c_5)	206: (a_2 b_5 c_6)
27: (a_3 b_5 c_1)	63: (a_3 b_5 c_2)	99: (a_3 b_5 c_3)	135: (a_3 b_5 c_4)	171: (a_3 b_5 c_5)	207: (a_3 b_5 c_6)
28: (a_4 b_5 c_1)	64: (a_4 b_5 c_2)	100: (a_4 b_5 c_3)	136: (a_4 b_5 c_4)	172: (a_4 b_5 c_5)	208: (a_4 b_5 c_6)
29: (a_5 b_5 c_1)	65: (a_5 b_5 c_2)	101: (a_5 b_5 c_3)	137: (a_5 b_5 c_4)	173: (a_5 b_5 c_5)	209: (a_5 b_5 c_6)
30: (a_6 b_5 c_1)	66: (a_6 b_5 c_2)	102: (a_6 b_5 c_3)	138: (a_6 b_5 c_4)	174: (a_6 b_5 c_5)	210: (a_6 b_5 c_6)
31: (a_1 b_6 c_1)	67: (a_1 b_6 c_2)	103: (a_1 b_6 c_3)	139: (a_1 b_6 c_4)	175: (a_1 b_6 c_5)	211: (a_1 b_6 c_6)
32: (a_2 b_6 c_1)	68: (a_2 b_6 c_2)	104: (a_2 b_6 c_3)	140: (a_2 b_6 c_4)	176: (a_2 b_6 c_5)	212: (a_2 b_6 c_6)
33: (a_3 b_6 c_1)	69: (a_3 b_6 c_2)	105: (a_3 b_6 c_3)	141: (a_3 b_6 c_4)	177: (a_3 b_6 c_5)	213: (a_3 b_6 c_6)
34: (a_4 b_6 c_1)	70: (a_4 b_6 c_2)	106: (a_4 b_6 c_3)	142: (a_4 b_6 c_4)	178: (a_4 b_6 c_5)	214: (a_4 b_6 c_6)
35: (a_5 b_6 c_1)	71: (a_5 b_6 c_2)	107: (a_5 b_6 c_3)	143: (a_5 b_6 c_4)	179: (a_5 b_6 c_5)	215: (a_5 b_6 c_6)
36: (a_6 b_6 c_1)	72: (a_6 b_6 c_2)	108: (a_6 b_6 c_3)	144: (a_6 b_6 c_4)	180: (a_6 b_6 c_5)	216: (a_6 b_6 c_6)

Figure 7. All 216 possible configurations for k=6 choices and n=3 players. These configurations have not yet been assigned to any sequences, so the ordering is not determined.

4. If Condition 3 is satisfied, the number of occurrences of each token in assigned configurations is the number of configurations times the number of players, divided by the total number of tokens. This can be expressed as $(n \bullet k^n) \div (\Sigma k)$.

Example. Figure 7 lists all 216 possible configurations for n=3 players, where each player has k=6 choices. There are 18 distinct tokens, a_1, a_2, …, c_5, c_6. The number of occurrences of each token in the set of configurations is $(3 \bullet 6^3)/(6+6+6) = 648/18 = 36$. Each of the 18 tokens, a_1, …, c_6, appears in 36 configurations.

5. In order for Conditions 1, 2 and 3 to be satisfied for n players, a selection for k (the number of choices for each player) must be made such that $0 \equiv k^n \bmod n!$ (Condition 1), $k \geq 2$ (Condition 2), and $0 \equiv k^n \bmod n \bullet k$ (Condition 3). The theorem below claims there always exist such a k for any $n \geq 2$. The associated lemma identifies one such k as $n!$.

Theorem A. For any number of players (n), where $n \geq 2$, there exists a selection for k satisfying $0 \equiv k^n \bmod n!$ (Condition 1), $k \geq 2$ (Condition 2), and $0 \equiv k^n \bmod n \bullet k$ (Condition 3).

Proof. Let $k = n!$. Because $n \geq 2$, and $k=n!$, $k \geq 2$. This satisfies Condition 2. For Condition 1 to be satisfied, $0 \equiv k^n \bmod n!$. Let $r_1 \equiv k^n \bmod n!$. Substituting $n!$ for k yields $r_1 \equiv (n!)^n \bmod n!$. Because $(n!)^n$ is divisible by $n!$, as shown below, Condition 1 is satisfied.

$$\frac{(n!)^n}{n!} = (n!)^{n-1}$$ is an integer value. There is no remainder, $r_1=0$.

For Condition 3 to be satisfied, $0 \equiv k^n \bmod n \bullet k$. Let $r_3 \equiv k^n \bmod n \bullet k$. Substituting $n!$ for k yields $r_3 \equiv (n!)^n \bmod n \bullet (n!)$. Because $(n!)^n$ is divisible by $n \bullet n!$, as shown below, Condition 3 is satisfied.

$$\frac{(n!)^n}{n \bullet n!} = \frac{(n!)^{n-1}}{n} = (n-1)! \bullet (n!)^{n-2}$$ is an integer value, so $r_3=0$.

□

Lemma B. For any number of players (n), where $n \geq 2$, a selection of $k=n!$ satisfies $0 \equiv k^n \bmod n!$ (Condition 1), $k \geq 2$ (Condition 2), and $0 \equiv k^n \bmod n \bullet k$ (Condition 3).

Example. Figure 8 shows the minimum selection for k that satisfies Conditions 1, 2 and 3, for n = 2 to 6 players. For n=2 players, where each player has k=2 choices, there are k^n=4 possible configurations, namely $\{a_1b_1, a_2b_1, a_2b_1, a_2b_2\}$. There are $n \bullet k$=4 tokens in total and $n!$=2 possible sequences, namely ab and ba. The number of configurations is divisible by the number of sequences ($0 \equiv 4 \bmod 2$) and the number of configurations is divisible by the number of token ($0 \equiv 4 \bmod 4$).

Example. In Figure 8, the selection of k for n=2 and 3 are $n!$, but for n=4, 5 and 6, a value for k smaller than $n!$ was found to satisfy Conditions 1, 2 and 3. In these cases, selecting k to be $n!$ would also satisfy the Conditions 1, 2 and 3. Results where $k=n!$ are shown in Figure 20.

6. Once the members of the set of all possible configurations are assigned to members of the set of all possible sequences, the tokens should be evenly distributed across all token positions in sequences. There should not exist a token appearing in more positions than any other token. This is identified as Condition 4, "the Token Position Condition."

Condition 4 ("Token Position"). For all tokens t_j to be evenly distributed over the positions p_x of the set of sequences, it is sufficient and necessary that: $\text{count}(t_j, p_x) = k^n/(k \bullet n)$, for each token j=1,…,k (assuming each player has k distinct tokens) and for each position x=1,…,n.

	Condition 2				Condition 1	Condition 3
n	***k***	k^n	$n \bullet k$	***n!***	k^n ***mod n!***	$k^n \bmod n \bullet k$
2	2	4	4	2	0	0
3	6	216	18	6	0	0
4	6	1296	24	24	0	0
5	30	24300000	150	120	0	0
6	30	729000000	180	720	0	0

Figure 8. Minimum selection for k that satisfies Conditions 1, 2 and 3, for n = 2 to 6 players. For n=2 and 3, $k=n!$, but for n=4, 5, and 6, $k < n!$.

7. In comparison, Condition 3 assures there are a proper number of tokens to evenly distribute over the set of all possible configurations. It does not address the positions of the tokens within those configurations. This is addressed by Condition 4. Suppose Condition 4 is grossly ignored even though Conditions 1, 2 and 3 are satisfied, and a token appears in only one position. Then, the player that chooses that token can reliably predict its placement in the final sequence.

Example. Figure 9 shows an assignment, of all 216 configurations k=6 choices for n=3 players. These configurations satisfy Conditions 1, 2 and 3, as described earlier. In Figure 9, the configurations are assigned to sequences cba, bca, cab, acb, bac and abc. Other assignments are possible. The assignments shown do not satisfy Condition 4 because b_3 and b_4 appear in the last position only, b_1 and b_6 appear in the middle position only, and b_2 and b_5 appear in the first positions only. Player b's token are not evenly distributed across the positions. As a result, player b can dictate his position in the final sequence.

Example. Figure 10 reports the frequency of each player's token in each of the token position listed in the sequences in Figure 9. All 36 copies of each token for Players a and c appear evenly distributed across the sequence positions – satisfying Condition 3. But all 36 copies of each of Player b's tokens only appear in single position – failing to satisfy the count requirement of Condition 4, which is $k^n/(k \bullet n) = 12$, not 36.

Example. Figure 11 shows a different assignment of the configurations from Figure 9 such that Conditions 1, 2, 3 and 4 are all satisfied. The frequencies of each player's token in each token position are reported in Figure 12. Each token, $a_1,\ldots,a_6$, $b_1,\ldots, b_6,\ldots, c_1, \ldots, c_6$, appear 12 times in each position, which agrees with the count requirement of Condition4, which is 12.

Example. Figure 13 shows another assignment of the same configurations from Figure 9 and Figure 11. The assignments in Figure 13, like those in Figure 11, satisfy all four conditions.

8. In order to defeat up to n-1 colluding players, assume all but one player agrees to specific values, $v_1, \ldots, v_{n-1}$. The k possible choices for Player n should allow any of the sequences to be equally selected by Player n. If so, the collusion provided no advantage. All configurations $(v_1, \ldots, v_{n-1}, v_{nx})$ for x=1,…,k, should be evenly distributed over the sequences. This requirement is identified as Condition 5, "the Anti-Collusion Condition."

Condition 5 ("Anti-Collusion"). Given n players with each player having k choices, let v_1 be Player 1's choice, …, v_{n-1} be Player (n-1)'s choice. There are k configurations containing these specific values, which are characterized by the k choices, $v_{n1}, \ldots, v_{nk}$. It is sufficient and necessary that each configuration of the form $(v_1, \ldots, v_{n-1}, v_{nx})$, for x=1,…,k, be assigned such that all sequences have an equal number of one or more such configurations assigned.

Example. Given 3 players, a, b, and c, with each player having 6 choices, and the assignment of configurations to sequences shown in Figure 11, collusion by Players b and c dictates the final selection no matter the choice made by Player a. If Player b selects b_6 and Player c selects c_1, the result is abc regardless of the choice of Player a. This bias occurs even though the assignments in Figure 11 satisfy Conditions 1, 2, 3 and 4. Figure 11 does not satisfy Condition 5.

		cba [0]	bca [1]	cab [2]	acb [3]	bac [4]	abc [5]
	[1]	1: $c_1 b_1 a_1$	7: $b_2 c_1 a_1$	13: $c_1 a_1 b_3$	19: $a_1 c_1 b_4$	25: $b_5 a_1 c_1$	31: $a_1 b_6 c_1$
	[2]	2: $c_1 b_1 a_2$	8: $b_2 c_1 a_2$	14: $c_1 a_2 b_3$	20: $a_2 c_1 b_4$	26: $b_5 a_2 c_1$	32: $a_2 b_6 c_1$
I	[3]	3: $c_1 b_1 a_3$	9: $b_2 c_1 a_3$	15: $c_1 a_3 b_3$	21: $a_3 c_1 b_4$	27: $b_5 a_3 c_1$	33: $a_3 b_6 c_1$
	[4]	4: $c_1 b_1 a_4$	10: $b_2 c_1 a_4$	16: $c_1 a_4 b_3$	22: $a_4 c_1 b_4$	28: $b_5 a_4 c_1$	34: $a_4 b_6 c_1$
	[5]	5: $c_1 b_1 a_5$	11: $b_2 c_1 a_5$	17: $c_1 a_5 b_3$	23: $a_5 c_1 b_4$	29: $b_5 a_5 c_1$	35: $a_5 b_6 c_1$
	[6]	6: $c_1 b_1 a_6$	12: $b_2 c_1 a_6$	18: $c_1 a_6 b_3$	24: $a_6 c_1 b_4$	30: $b_5 a_6 c_1$	36: $a_6 b_6 c_1$
	[7]	37: $c_2 b_1 a_1$	43: $b_2 c_2 a_1$	49: $c_2 a_1 b_3$	55: $a_1 c_2 b_4$	61: $b_5 a_1 c_2$	67: $a_1 b_6 c_2$
	[8]	38: $c_2 b_1 a_2$	44: $b_2 c_2 a_2$	50: $c_2 a_2 b_3$	56: $a_2 c_2 b_4$	62: $b_5 a_2 c_2$	68: $a_2 b_6 c_2$
II	[9]	39: $c_2 b_1 a_3$	45: $b_2 c_2 a_3$	51: $c_2 a_3 b_3$	57: $a_3 c_2 b_4$	63: $b_5 a_3 c_2$	69: $a_3 b_6 c_2$
	[10]	40: $c_2 b_1 a_4$	46: $b_2 c_2 a_4$	52: $c_2 a_4 b_3$	58: $a_4 c_2 b_4$	64: $b_5 a_4 c_2$	70: $a_4 b_6 c_2$
	[11]	41: $c_2 b_1 a_5$	47: $b_2 c_2 a_5$	53: $c_2 a_5 b_3$	59: $a_5 c_2 b_4$	65: $b_5 a_5 c_2$	71: $a_5 b_6 c_2$
	[12]	42: $c_2 b_1 a_6$	48: $b_2 c_2 a_6$	54: $c_2 a_6 b_3$	60: $a_6 c_2 b_4$	66: $b_5 a_6 c_2$	72: $a_6 b_6 c_2$
	[13]	73: $c_3 b_1 a_1$	79: $b_2 c_3 a_1$	85: $c_3 a_1 b_3$	91: $a_1 c_3 b_4$	97: $b_5 a_1 c_3$	103: $a_1 b_6 c_3$
	[14]	74: $c_3 b_1 a_2$	80: $b_2 c_3 a_2$	86: $c_3 a_2 b_3$	92: $a_2 c_3 b_4$	98: $b_5 a_2 c_3$	104: $a_2 b_6 c_3$
III	[15]	75: $c_3 b_1 a_3$	81: $b_2 c_3 a_3$	87: $c_3 a_3 b_3$	93: $a_3 c_3 b_4$	99: $b_5 a_3 c_3$	105: $a_3 b_6 c_3$
	[16]	76: $c_3 b_1 a_4$	82: $b_2 c_3 a_4$	88: $c_3 a_4 b_3$	94: $a_4 c_3 b_4$	100: $b_5 a_4 c_3$	106: $a_4 b_6 c_3$
	[17]	77: $c_3 b_1 a_5$	83: $b_2 c_3 a_5$	89: $c_3 a_5 b_3$	95: $a_5 c_3 b_4$	101: $b_5 a_5 c_3$	107: $a_5 b_6 c_3$
	[18]	78: $c_3 b_1 a_6$	84: $b_2 c_3 a_6$	90: $c_3 a_6 b_3$	96: $a_6 c_3 b_4$	102: $b_5 a_6 c_3$	108: $a_6 b_6 c_3$
	[19]	109: $c_4 b_1 a_1$	115: $b_2 c_4 a_1$	121: $c_4 a_1 b_3$	127: $a_1 c_4 b_4$	133: $b_5 a_1 c_4$	139: $a_1 b_6 c_4$
	[20]	110: $c_4 b_1 a_2$	116: $b_2 c_4 a_2$	122: $c_4 a_2 b_3$	128: $a_2 c_4 b_4$	134: $b_5 a_2 c_4$	140: $a_2 b_6 c_4$
IV	[21]	111: $c_4 b_1 a_3$	117: $b_2 c_4 a_3$	123: $c_4 a_3 b_3$	129: $a_3 c_4 b_4$	135: $b_5 a_3 c_4$	141: $a_3 b_6 c_4$
	[22]	112: $c_4 b_1 a_4$	118: $b_2 c_4 a_4$	124: $c_4 a_4 b_3$	130: $a_4 c_4 b_4$	136: $b_5 a_4 c_4$	142: $a_4 b_6 c_4$
	[23]	113: $c_4 b_1 a_5$	119: $b_2 c_4 a_5$	125: $c_4 a_5 b_3$	131: $a_5 c_4 b_4$	137: $b_5 a_5 c_4$	143: $a_5 b_6 c_4$
	[24]	114: $c_4 b_1 a_6$	120: $b_2 c_4 a_6$	126: $c_4 a_6 b_3$	132: $a_6 c_4 b_4$	138: $b_5 a_6 c_4$	144: $a_6 b_6 c_4$
	[25]	145: $c_5 b_1 a_1$	151: $b_2 c_5 a_1$	157: $c_5 a_1 b_3$	163: $a_1 c_5 b_4$	169: $b_5 a_1 c_5$	175: $a_1 b_6 c_5$
	[26]	146: $c_5 b_1 a_2$	152: $b_2 c_5 a_2$	158: $c_5 a_2 b_3$	164: $a_2 c_5 b_4$	170: $b_5 a_2 c_5$	176: $a_2 b_6 c_5$
V	[27]	147: $c_5 b_1 a_3$	153: $b_2 c_5 a_3$	159: $c_5 a_3 b_3$	165: $a_3 c_5 b_4$	171: $b_5 a_3 c_5$	177: $a_3 b_6 c_5$
	[28]	148: $c_5 b_1 a_4$	154: $b_2 c_5 a_4$	160: $c_5 a_4 b_3$	166: $a_4 c_5 b_4$	172: $b_5 a_4 c_5$	178: $a_4 b_6 c_5$
	[29]	149: $c_5 b_1 a_5$	155: $b_2 c_5 a_5$	161: $c_5 a_5 b_3$	167: $a_5 c_5 b_4$	173: $b_5 a_5 c_5$	179: $a_5 b_6 c_5$
	[30]	150: $c_5 b_1 a_6$	156: $b_2 c_5 a_6$	162: $c_5 a_6 b_3$	168: $a_6 c_5 b_4$	174: $b_5 a_6 c_5$	180: $a_6 b_6 c_5$
	[31]	181: $c_6 b_1 a_1$	187: $b_2 c_6 a_1$	193: $c_6 a_1 b_3$	199: $a_1 c_6 b_4$	205: $b_5 a_1 c_6$	211: $a_1 b_6 c_6$
	[32]	182: $c_6 b_1 a_2$	188: $b_2 c_6 a_2$	194: $c_6 a_2 b_3$	200: $a_2 c_6 b_4$	206: $b_5 a_2 c_6$	212: $a_2 b_6 c_6$
VI	[33]	183: $c_6 b_1 a_3$	189: $b_2 c_6 a_3$	195: $c_6 a_3 b_3$	201: $a_3 c_6 b_4$	207: $b_5 a_3 c_6$	213: $a_3 b_6 c_6$
	[34]	184: $c_6 b_1 a_4$	190: $b_2 c_6 a_4$	196: $c_6 a_4 b_3$	202: $a_4 c_6 b_4$	208: $b_5 a_4 c_6$	214: $a_4 b_6 c_6$
	[35]	185: $c_6 b_1 a_5$	191: $b_2 c_6 a_5$	197: $c_6 a_5 b_3$	203: $a_5 c_6 b_4$	209: $b_5 a_5 c_6$	215: $a_5 b_6 c_6$
	[36]	186: $c_6 b_1 a_6$	192: $b_2 c_6 a_6$	198: $c_6 a_6 b_3$	204: $a_6 c_6 b_4$	210: $b_5 a_6 c_6$	216: $a_6 b_6 c_6$

Figure 9. All 216 possible configurations for k=6 choices and n=3 players from Figure 7 are assigned to the $n!$=6 sequences (columns numbered [0] to [5]) imposing an ordering on each configuration. The assignments form a 2-dimensional array (or table) having 5 columns and 36 rows.

Player a

Token	Pos 1	Pos 2	Pos 3
a_1	12	12	12
a_2	12	12	12
a_3	12	12	12
a_4	12	12	12
a_5	12	12	12
a_6	12	12	12

Player b

Token	Pos 1	Pos 2	Pos 3
b_1	0	36	0
b_2	36	0	0
b_3	0	0	36
b_4	0	0	36
b_5	36	0	0
b_6	0	36	0

Player c

Token	Pos 1	Pos 2	Pos 3
c_1	12	12	12
c_2	12	12	12
c_3	12	12	12
c_4	12	12	12
c_5	12	12	12
c_6	12	12	12

Figure 10. Counts of the number of times the tokens of each player appears in each sequence position in the configuration assignments shown in Figure 9.

		cba [0]	bca [1]	cab [2]	acb [3]	bac [4]	abc [5]
I	[1]	1: c_1 b_1 a_1	7: b_2 c_1 a_1	13: c_1 a_1 b_3	19: a_1 c_1 b_4	25: b_5 a_1 c_1	31: a_1 b_6 c_1
	[2]	2: c_1 b_1 a_2	8: b_2 c_1 a_2	14: c_1 a_2 b_3	20: a_2 c_1 b_4	26: b_5 a_2 c_1	32: a_2 b_6 c_1
	[3]	3: c_1 b_1 a_3	9: b_2 c_1 a_3	15: c_1 a_3 b_3	21: a_3 c_1 b_4	27: b_5 a_3 c_1	33: a_3 b_6 c_1
	[4]	4: c_1 b_1 a_4	10: b_2 c_1 a_4	16: c_1 a_4 b_3	22: a_4 c_1 b_4	28: b_5 a_4 c_1	34: a_4 b_6 c_1
	[5]	5: c_1 b_1 a_5	11: b_2 c_1 a_5	17: c_1 a_5 b_3	23: a_5 c_1 b_4	29: b_5 a_5 c_1	35: a_5 b_6 c_1
	[6]	6: c_1 b_1 a_6	12: b_2 c_1 a_6	18: c_1 a_6 b_3	24: a_6 c_1 b_4	30: b_5 a_6 c_1	36: a_6 b_6 c_1
II	[7]	37: c_2 b_6 a_1	43: b_1 c_2 a_1	49: c_2 a_1 b_2	55: a_1 c_2 b_3	61: b_4 a_1 c_2	67: a_1 b_5 c_2
	[8]	38: c_2 b_6 a_2	44: b_1 c_2 a_2	50: c_2 a_2 b_2	56: a_2 c_2 b_3	62: b_4 a_2 c_2	68: a_2 b_5 c_2
	[9]	39: c_2 b_6 a_3	45: b_1 c_2 a_3	51: c_2 a_3 b_2	57: a_3 c_2 b_3	63: b_4 a_3 c_2	69: a_3 b_5 c_2
	[10]	40: c_2 b_6 a_4	46: b_1 c_2 a_4	52: c_2 a_4 b_2	58: a_4 c_2 b_3	64: b_4 a_4 c_2	70: a_4 b_5 c_2
	[11]	41: c_2 b_6 a_5	47: b_1 c_2 a_5	53: c_2 a_5 b_2	59: a_5 c_2 b_3	65: b_4 a_5 c_2	71: a_5 b_5 c_2
	[12]	42: c_2 b_6 a_6	48: b_1 c_2 a_6	54: c_2 a_6 b_2	60: a_6 c_2 b_3	66: b_4 a_6 c_2	72: a_6 b_5 c_2
III	[13]	73: c_3 b_5 a_1	79: b_6 c_3 a_1	85: c_3 a_1 b_1	91: a_1 c_3 b_2	97: b_3 a_1 c_3	103: a_1 b_4 c_3
	[14]	74: c_3 b_5 a_2	80: b_6 c_3 a_2	86: c_3 a_2 b_1	92: a_2 c_3 b_2	98: b_3 a_2 c_3	104: a_2 b_4 c_3
	[15]	75: c_3 b_5 a_3	81: b_6 c_3 a_3	87: c_3 a_3 b_1	93: a_3 c_3 b_2	99: b_3 a_3 c_3	105: a_3 b_4 c_3
	[16]	76: c_3 b_5 a_4	82: b_6 c_3 a_4	88: c_3 a_4 b_1	94: a_4 c_3 b_2	100: b_3 a_4 c_3	106: a_4 b_4 c_3
	[17]	77: c_3 b_5 a_5	83: b_6 c_3 a_5	89: c_3 a_5 b_1	95: a_5 c_3 b_2	101: b_3 a_5 c_3	107: a_5 b_4 c_3
	[18]	78: c_3 b_5 a_6	84: b_6 c_3 a_6	90: c_3 a_6 b_1	96: a_6 c_3 b_2	102: b_3 a_6 c_3	108: a_6 b_4 c_3
IV	[19]	109: c_4 b_4 a_1	115: b_5 c_4 a_1	121: c_4 a_1 b_6	127: a_1 c_4 b_1	133: b_2 a_1 c_4	139: a_1 b_3 c_4
	[20]	110: c_4 b_4 a_2	116: b_5 c_4 a_2	122: c_4 a_2 b_6	128: a_2 c_4 b_1	134: b_2 a_2 c_4	140: a_2 b_3 c_4
	[21]	111: c_4 b_4 a_3	117: b_5 c_4 a_3	123: c_4 a_3 b_6	129: a_3 c_4 b_1	135: b_2 a_3 c_4	141: a_3 b_3 c_4
	[22]	112: c_4 b_4 a_4	118: b_5 c_4 a_4	124: c_4 a_4 b_6	130: a_4 c_4 b_1	136: b_2 a_4 c_4	142: a_4 b_3 c_4
	[23]	113: c_4 b_4 a_5	119: b_5 c_4 a_5	125: c_4 a_5 b_6	131: a_5 c_4 b_1	137: b_2 a_5 c_4	143: a_5 b_3 c_4
	[24]	114: c_4 b_4 a_6	120: b_5 c_4 a_6	126: c_4 a_6 b_6	132: a_6 c_4 b_1	138: b_2 a_6 c_4	144: a_6 b_3 c_4
V	[25]	145: c_5 b_3 a_1	151: b_4 c_5 a_1	157: c_5 a_1 b_5	163: a_1 c_5 b_6	169: b_1 a_1 c_5	175: a_1 b_2 c_5
	[26]	146: c_5 b_3 a_2	152: b_4 c_5 a_2	158: c_5 a_2 b_5	164: a_2 c_5 b_6	170: b_1 a_2 c_5	176: a_2 b_2 c_5
	[27]	147: c_5 b_3 a_3	153: b_4 c_5 a_3	159: c_5 a_3 b_5	165: a_3 c_5 b_6	171: b_1 a_3 c_5	177: a_3 b_2 c_5
	[28]	148: c_5 b_3 a_4	154: b_4 c_5 a_4	160: c_5 a_4 b_5	166: a_4 c_5 b_6	172: b_1 a_4 c_5	178: a_4 b_2 c_5
	[29]	149: c_5 b_3 a_5	155: b_4 c_5 a_5	161: c_5 a_5 b_5	167: a_5 c_5 b_6	173: b_1 a_5 c_5	179: a_5 b_2 c_5
	[30]	150: c_5 b_3 a_6	156: b_4 c_5 a_6	162: c_5 a_6 b_5	168: a_6 c_5 b_6	174: b_1 a_6 c_5	180: a_6 b_2 c_5
VI	[31]	181: c_6 b_2 a_1	187: b_3 c_6 a_1	193: c_6 a_1 b_4	199: a_1 c_6 b_5	205: b_6 a_1 c_6	211: a_1 b_1 c_6
	[32]	182: c_6 b_2 a_2	188: b_3 c_6 a_2	194: c_6 a_2 b_4	200: a_2 c_6 b_5	206: b_6 a_2 c_6	212: a_2 b_1 c_6
	[33]	183: c_6 b_2 a_3	189: b_3 c_6 a_3	195: c_6 a_3 b_4	201: a_3 c_6 b_5	207: b_6 a_3 c_6	213: a_3 b_1 c_6
	[34]	184: c_6 b_2 a_4	190: b_3 c_6 a_4	196: c_6 a_4 b_4	202: a_4 c_6 b_5	208: b_6 a_4 c_6	214: a_4 b_1 c_6
	[35]	185: c_6 b_2 a_5	191: b_3 c_6 a_5	197: c_6 a_5 b_4	203: a_5 c_6 b_5	209: b_6 a_5 c_6	215: a_5 b_1 c_6
	[36]	186: c_6 b_2 a_6	192: b_3 c_6 a_6	198: c_6 a_6 b_4	204: a_6 c_6 b_5	210: b_6 a_6 c_6	216: a_6 b_1 c_6

Figure 11. Assignments of the 216 possible configurations from k=6 token choices for n=3 players, a, b and c, to the n!=6 sequences (columns numbered [0] to [5]) such that Condition 4 (and Conditions 1,2 and 3) is satisfied.

Player a

Token	Pos 1	Pos 2	Pos 3
a_1	12	12	12
a_2	12	12	12
a_3	12	12	12
a_4	12	12	12
a_5	12	12	12
a_6	12	12	12

Player b

Token	Pos 1	Pos 2	Pos 3
b_1	12	12	12
b_2	12	12	12
b_3	12	12	12
b_4	12	12	12
b_5	12	12	12
b_6	12	12	12

Player c

Token	Pos 1	Pos 2	Pos 3
c_1	12	12	12
c_2	12	12	12
c_3	12	12	12
c_4	12	12	12
c_5	12	12	12
c_6	12	12	12

Figure 12. Counts of the number of times the tokens of each player appears in each sequence position in the configuration assignments shown in Figure 11. Each token, $a_1,\ldots,a_6,b_1,\ldots,b_6,c_1,\ldots,c_6$, appears 12 times in each token position.

		cba [0]	bca [1]	cab [2]	acb [3]	bac [4]	abc [5]
	[1]	1: $c_1 b_1 a_1$	7: $b_2 c_1 a_1$	13: $c_1 a_1 b_3$	19: $a_1 c_1 b_4$	25: $b_5 a_1 c_1$	31: $a_1 b_6 c_1$
	[2]	2: $c_1 b_6 a_2$	8: $b_1 c_1 a_2$	14: $c_1 a_2 b_2$	20: $a_2 c_1 b_3$	26: $b_4 a_2 c_1$	32: $a_2 b_5 c_1$
I	[3]	3: $c_1 b_5 a_3$	9: $b_6 c_1 a_3$	15: $c_1 a_3 b_1$	21: $a_3 c_1 b_2$	27: $b_3 a_3 c_1$	33: $a_3 b_4 c_1$
	[4]	4: $c_1 b_4 a_4$	10: $b_5 c_1 a_4$	16: $c_1 a_4 b_6$	22: $a_4 c_1 b_1$	28: $b_2 a_4 c_1$	34: $a_4 b_3 c_1$
	[5]	5: $c_1 b_3 a_5$	11: $b_4 c_1 a_5$	17: $c_1 a_5 b_5$	23: $a_5 c_1 b_6$	29: $b_1 a_5 c_1$	35: $a_5 b_2 c_1$
	[6]	6: $c_1 b_2 a_6$	12: $b_3 c_1 a_6$	18: $c_1 a_6 b_4$	24: $a_6 c_1 b_5$	30: $b_6 a_6 c_1$	36: $a_6 b_1 c_1$
	[7]	37: $c_2 b_6 a_1$	43: $b_1 c_2 a_1$	49: $c_2 a_1 b_2$	55: $a_1 c_2 b_3$	61: $b_4 a_1 c_2$	67: $a_1 b_5 c_2$
	[8]	38: $c_2 b_5 a_2$	44: $b_6 c_2 a_2$	50: $c_2 a_2 b_1$	56: $a_2 c_2 b_2$	62: $b_3 a_2 c_2$	68: $a_2 b_4 c_2$
II	[9]	39: $c_2 b_4 a_3$	45: $b_5 c_2 a_3$	51: $c_2 a_3 b_6$	57: $a_3 c_2 b_1$	63: $b_2 a_3 c_2$	69: $a_3 b_3 c_2$
	[10]	40: $c_2 b_3 a_4$	46: $b_4 c_2 a_4$	52: $c_2 a_4 b_5$	58: $a_4 c_2 b_6$	64: $b_1 a_4 c_2$	70: $a_4 b_2 c_2$
	[11]	41: $c_2 b_2 a_5$	47: $b_3 c_2 a_5$	53: $c_2 a_5 b_4$	59: $a_5 c_2 b_5$	65: $b_6 a_5 c_2$	71: $a_5 b_1 c_2$
	[12]	42: $c_2 b_1 a_6$	48: $b_2 c_2 a_6$	54: $c_2 a_6 b_3$	60: $a_6 c_2 b_4$	66: $b_5 a_6 c_2$	72: $a_6 b_6 c_2$
	[13]	73: $c_3 b_5 a_1$	79: $b_6 c_3 a_1$	85: $c_3 a_1 b_1$	91: $a_1 c_3 b_2$	97: $b_3 a_1 c_3$	103: $a_1 b_4 c_3$
	[14]	74: $c_3 b_4 a_2$	80: $b_5 c_3 a_2$	86: $c_3 a_2 b_6$	92: $a_2 c_3 b_1$	98: $b_2 a_2 c_3$	104: $a_2 b_3 c_3$
III	[15]	75: $c_3 b_3 a_3$	81: $b_4 c_3 a_3$	87: $c_3 a_3 b_5$	93: $a_3 c_3 b_6$	99: $b_1 a_3 c_3$	105: $a_3 b_2 c_3$
	[16]	76: $c_3 b_2 a_4$	82: $b_3 c_3 a_4$	88: $c_3 a_4 b_4$	94: $a_4 c_3 b_5$	100: $b_6 a_4 c_3$	106: $a_4 b_1 c_3$
	[17]	77: $c_3 b_1 a_5$	83: $b_2 c_3 a_5$	89: $c_3 a_5 b_3$	95: $a_5 c_3 b_4$	101: $b_5 a_5 c_3$	107: $a_5 b_6 c_3$
	[18]	78: $c_3 b_6 a_6$	84: $b_1 c_3 a_6$	90: $c_3 a_6 b_2$	96: $a_6 c_3 b_3$	102: $b_4 a_6 c_3$	108: $a_6 b_5 c_3$
	[19]	109: $c_4 b_4 a_1$	115: $b_5 c_4 a_1$	121: $c_4 a_1 b_6$	127: $a_1 c_4 b_1$	133: $b_2 a_1 c_4$	139: $a_1 b_3 c_4$
	[20]	110: $c_4 b_3 a_2$	116: $b_4 c_4 a_2$	122: $c_4 a_2 b_5$	128: $a_2 c_4 b_6$	134: $b_1 a_2 c_4$	140: $a_2 b_2 c_4$
IV	[21]	111: $c_4 b_2 a_3$	117: $b_3 c_4 a_3$	123: $c_4 a_3 b_4$	129: $a_3 c_4 b_5$	135: $b_6 a_3 c_4$	141: $a_3 b_1 c_4$
	[22]	112: $c_4 b_1 a_4$	118: $b_2 c_4 a_4$	124: $c_4 a_4 b_3$	130: $a_4 c_4 b_4$	136: $b_5 a_4 c_4$	142: $a_4 b_6 c_4$
	[23]	113: $c_4 b_6 a_5$	119: $b_1 c_4 a_5$	125: $c_4 a_5 b_2$	131: $a_5 c_4 b_3$	137: $b_4 a_5 c_4$	143: $a_5 b_5 c_4$
	[24]	114: $c_4 b_5 a_6$	120: $b_6 c_4 a_6$	126: $c_4 a_6 b_1$	132: $a_6 c_4 b_2$	138: $b_3 a_6 c_4$	144: $a_6 b_4 c_4$
	[25]	145: $c_5 b_3 a_1$	151: $b_4 c_5 a_1$	157: $c_5 a_1 b_5$	163: $a_1 c_5 b_6$	169: $b_1 a_1 c_5$	175: $a_1 b_2 c_5$
	[26]	146: $c_5 b_2 a_2$	152: $b_3 c_5 a_2$	158: $c_5 a_2 b_4$	164: $a_2 c_5 b_5$	170: $b_6 a_2 c_5$	176: $a_2 b_1 c_5$
V	[27]	147: $c_5 b_1 a_3$	153: $b_2 c_5 a_3$	159: $c_5 a_3 b_3$	165: $a_3 c_5 b_4$	171: $b_5 a_3 c_5$	177: $a_3 b_6 c_5$
	[28]	148: $c_5 b_6 a_4$	154: $b_1 c_5 a_4$	160: $c_5 a_4 b_2$	166: $a_4 c_5 b_3$	172: $b_4 a_4 c_5$	178: $a_4 b_5 c_5$
	[29]	149: $c_5 b_5 a_5$	155: $b_6 c_5 a_5$	161: $c_5 a_5 b_1$	167: $a_5 c_5 b_2$	173: $b_3 a_5 c_5$	179: $a_5 b_4 c_5$
	[30]	150: $c_5 b_4 a_6$	156: $b_5 c_5 a_6$	162: $c_5 a_6 b_6$	168: $a_6 c_5 b_1$	174: $b_2 a_6 c_5$	180: $a_6 b_3 c_5$
	[31]	181: $c_6 b_2 a_1$	187: $b_3 c_6 a_1$	193: $c_6 a_1 b_4$	199: $a_1 c_6 b_5$	205: $b_6 a_1 c_6$	211: $a_1 b_1 c_6$
	[32]	182: $c_6 b_1 a_2$	188: $b_2 c_6 a_2$	194: $c_6 a_2 b_3$	200: $a_2 c_6 b_4$	206: $b_5 a_2 c_6$	212: $a_2 b_6 c_6$
VI	[33]	183: $c_6 b_6 a_3$	189: $b_1 c_6 a_3$	195: $c_6 a_3 b_2$	201: $a_3 c_6 b_3$	207: $b_4 a_3 c_6$	213: $a_3 b_5 c_6$
	[34]	184: $c_6 b_5 a_4$	190: $b_6 c_6 a_4$	196: $c_6 a_4 b_1$	202: $a_4 c_6 b_2$	208: $b_3 a_4 c_6$	214: $a_4 b_4 c_6$
	[35]	185: $c_6 b_4 a_5$	191: $b_5 c_6 a_5$	197: $c_6 a_5 b_6$	203: $a_5 c_6 b_1$	209: $b_2 a_5 c_6$	215: $a_5 b_3 c_6$
	[36]	186: $c_6 b_3 a_6$	192: $b_4 c_6 a_6$	198: $c_6 a_6 b_5$	204: $a_6 c_6 b_6$	210: $b_1 a_6 c_6$	216: $a_6 b_2 c_6$

Figure 13. Assignments of the 216 configurations for *n*=3 players (a, b, c), with each player having 6 choices, to the *n*!=6 sequences such that Condition 1, 2, 3, 4 and 5 are all satisfied. This table describes a RandomOrder function.

9. Satisfying Condition 4 assures that no sequence has a token appearing more often than any other token. But it does not address a situation in which 2 or more tokens appear in more configurations than any other 2 tokens.

Example. Figure 13 shows another assignment of the same configurations in Figure 11, but in Figure 13 the assignments satisfy Condition 5, as well as Conditions 1 through 4. Collusion by Player b and c does not dictate the outcome. If Player b selects b_6 and Player c selects c_1, the results depends on Player a. If Player a selects: a_1, abc results; a_2, cba results; a_3, bca results; a_4, cab results; a_5, acb results; and, a_6, bac results.

10. A corollary to Condition 5 id $0 \equiv k \bmod n!$ because there must be sufficient numbers of choices to distribute the configurations described in Condition 5. Condition 2 is updated so that all players have an equal number of choices congruent to $n!$.

1.	**Configurations**	$0 \equiv (k_1 \bullet \ldots \bullet k_n) \bmod n!$
2.	**Choices**	$k = k_1 = \ldots = k_n$
3.	**Tokens**	$0 \equiv k^n \bmod (n \bullet k)$
4.	**Token Position**	$\text{count}(t_j, p_x) = k^n / (k \bullet n)$ for each token j=1,…,$k \bullet n$; and for each position x=1,…,n
5.	**Anti-Collusion**	Let v_1, …, v_{n-1} be selected choices for Players 1,…, n-1. Distribute configurations (v_1,…,v_{n-1},v_{nx}), for x=1,…,n where v_{nx} are choices for Player n, evenly over the sequences.

Figure 14. Summary of the 5 conditions that help assure random selection by *n* players.

11. Figure 14 lists a summary of the 5 conditions presented in this subsection. Adherence to these conditions helps assure overall random selection.

12. The first three conditions make sure the choices available to players combine into configurations that are evenly distributed over the set of possible selections. Condition 5 assures the selections cannot be biased by collusion. Together, Conditions 1, 2, 3, and 5 (not including 4) are generally applicable and are not specific to the selection of sequences. For example, a function satisfying Conditions 1, 2, 3 and 5 can be used to enable *n* players to select a random integer between 0 and (*n*!-1).

13. Condition 4, however, is specific to the semantics of selecting sequences. The relationship between the position of players occurring within a selected sequence and the choices made by players to select the sequence must not be related.

14. A function satisfying the five conditions summarized in Figure 14 is termed a "RandomOrder function," the details of which are the presented in the next section.

4. Class of RandomOrder Functions

This section begins (Section 4.1) by constructing a function that adheres to the 5 conditions presented in the previous section. In Section 4.2, the class of RandomOrder functions is introduced. This section ends with a proof that the probability of a RandomOrder function selecting any particular permutation is not zero and is the same for all permutations. Additional discussion about collusion is also provided.

4.1. Construction of a Function That Satisfies the 5 Conditions

1. In this subsection, a function is constructed in three parts that adheres to the 5 conditions presented in the previous section. First, an algorithm is introduced that generates all possible configurations and assigns them to sequences such that Conditions 1, 2, and 3 are satisfied. The algorithm is then modified to also satisfy Condition 4. Finally, the algorithm is modified again to satisfy all 5 conditions.

2. Figure 15 lists an algorithm named BasicAssign() that given *n* players, with each player having *n*! choices, produces a 2-dimensional array (or table) in which all possible combinations of token choices are assigned to sequences with an even distribution. The 2-dimensional array is named assigns, and it has *n*! columns, one for each sequence, and (k^n/*n*!) rows. A cell, assigns[*row*,*col*] contains a configuration.

3. The primary construction of BasicAssign() consists of *n* nested counting loops (lines 2 through 6). Each loop cycles through a player's token choices. By nesting these loops, each possible combination of choices is enumerated, one at a time, and processed in lines 7 through 10.

Method: ***BasicAssign()***

Input: (1) n players, $n \geq 2$; (2) each player has $k=n!$ choices; (3) player choices are enumerated as tokens v_{px} for player p's choice of token x, for $1 \leq x \leq n!$; and, (4) an array of the permutations of players named *sequences* having one permutation per array position.

Output: a table (2-dimensional array) named assigns having $n!$ columns, one for each sequence, and $(k^n/n!)$ rows. A cell, assigns[*row*, *col*], contains a configuration assigned to the sequence associated with *col*.

```
1. let i = 1, j = 0
2. for p_n = 1 to n! do:
        3. ...
              4. for p_3 = 1 to n! do:
                    5. for p_2 = 1 to n! do:
                          6. for p_1 = 1 to n! do:
                                7. config_i = (v_P1, v_P2, …, v_Pn)
                                8. assigns[j/n!, j mod n!] = config_i      // uses only integer part of j/n
                                9. i = i+1
                                10. j = j+1
```

Figure 15. BasicAssign() Algorithm. This algorithm assigns configurations to sequences for *n* players, with each player having *n*! choices using *n* nested counting loops to generate each possible combination.

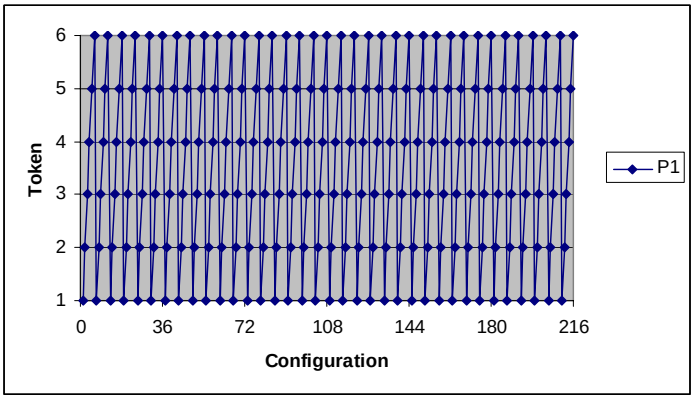
4. Line 7 temporarily stores the combination generated within an iteration of the loops as the *i*-th configuration. The *i*-th configuration is then stored in assigns at row int((*i*-1)/*n*!), where int() refers to the integer part of the division, and at column (*i*-1) mod *n*!. See line 8 in which *j* is *i*-1. The first *n*! configurations are assigned to column [0], the first sequence. The second batch of configurations, for $i=(n!+1),\ldots, 2 \bullet n!$ are stored in column [1], the second sequence. These batch assignments continue through the sequences. The configurations generated for $i=(n!-1) \bullet n!+1,\ldots, n!^2$ are stored in column [*n*!-1], the last sequence. Afterwards, batch assignments begin again with column [0] and continue cycling through the sequences in this manner until $i=(n!)^n$.

Example. An example of assignments made by BasicAssign() is shown in Figure 9 for $n=3$ players, where *p1* (line 6 in Figure 15) is player a, *p2* (line 5 in Figure 15) is player b, and *p3* (line 4 in Figure 15) is player c. Only those three loops are used. Figure 9 lists the contents of the resulting assigns array. There are 6 columns indexed [0] to [5], one for each sequence. There are 36 rows. Each cell shows the assigned configuration along with the order (*i*) in which the configuration was assigned (see line 8 in Figure 15). assigns[1,0] has $c_1b_1a_1$ and was entered in the array first; assigns[1,1] has $b_2c_1a_1$ and was entered 7th; and, assigns[36,5] has $a_6b_6c_6$ and was entered last.

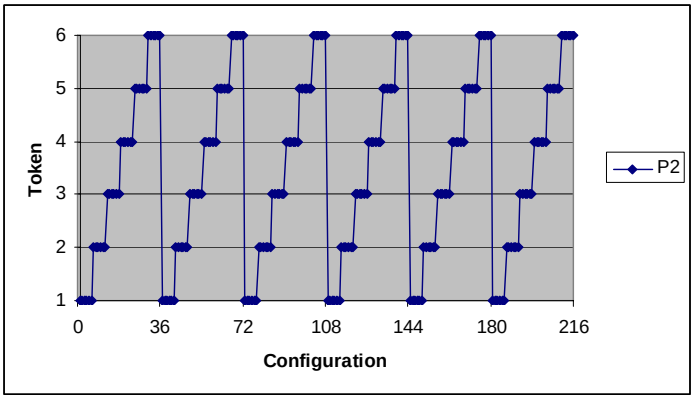
Lemma C. Executing the BasicAssign() algorithm with *n* players, where each player has *n*! token choices, produces a table of configurations assigned to sequences that satisfies Conditions 1, 2 and 3.

Proof sketch.
Because the number of choices was selected to be $n!$, there are sufficient number of tokens and configurations to be evenly distributed; see Lemma B. All possible combinations of choices are generated by the nested loops, totaling $(n!)^n$ configurations. Batches of sequentially generated unique configurations are assigned to sequences, where the size of each batch is $n!$. Because $0 \equiv (n!)^n \bmod n!$, there is an even distribution of batches to sequences.
□

(a)



(b)



(c)

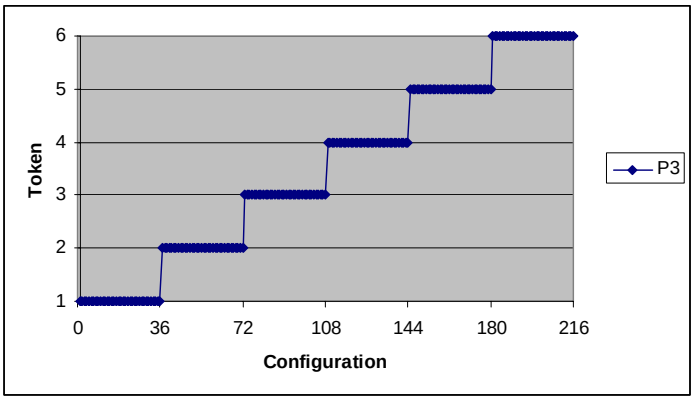


Figure 16. The frequencies at which BasicAssigns() in Figure 15 cycles through tokens for Player P1 (a), Player P2 (b) and Player P3 (c) while enumerating all possible combinations (configurations). These frequencies are 1 token for each configuration for Player P1, 1 token for every 6 configurations for Player P2, and 1 token for every 36 configurations for Player P3.

5. The nested loops in BasicAssign() provide regularity in the frequency at which a player's tokens are cycled. The resulting pattern appears in the configuration assignments, thereby thwarting the "randomness" of the assignments of tokens to token positions. Because BasicAssign() assigns batches of configurations to the same sequence, where each batch contains $n!$ configurations, players having tokens cycled at a frequency congruent to 1 token for every $n!$ configurations (e.g., Player 2) will not have tokens evenly distributed over token positions within sequences.

Example. Figure 16 shows the frequencies at which BasicAssign() cycles through player tokens in generating the table in Figure 9. Player P2 (b) has a frequency of 1 token for every 6 (which is n!) configurations. As a result, each batch of configurations has the same token for Player P2 (b) and these are configurations are all assigned to the same sequence. Figure 10 reports the frequencies of tokens in sequences and shows Player P2 (b) failing to have its tokens evenly distributed over token positions. As discussed in Section 3.2, the table in Figure 9 therefore fails to satisfy Condition 4.

6. The ShiftGroups() algorithm, listed in Figure 17, corrects the improper distribution of tokens resulting from BasicAssign(). This is done by shifting batches of configurations so they are reassigned to different sequences. In BasicAssign(), there will exist a player for which a specific token appears exclusively with a specific sequence, for all tokens and sequences. These tokens need to be redistributed across the sequences. ShiftGroups() accomplishes this by shifting batches of configurations (of size $n!$) to other sequences, in a horizontal rotary pattern. so that a batch originally assigned to the sequence having column [x] will be reassigned to all other sequences for x=0,…,(n!-1).

Example. An example of assignments made by BasicAssign(), listed in Figure 15, is shown in Figure 9 for n=3 players, a, b, and c. Figure 9 lists the contents of the resulting assigns array. There are 6 columns indexed [0] to [5], one for each sequence, and 36 rows. As noted in Section 3.2, the tokens for player b are not evenly distributed in token positions. Summary counts are available in Figure 10. Figure 11 then shows the contents of the assigns array after ShiftGroups(), listed in Figure 17, executes. Batches of configurations are reassigned to sequences. As summarized in Figure 12, the tokens are now even distributed across token positions, thereby satisfying Condition 4.

Method: ***ShiftGroups()***

Input: 2-dimensional array assigns resulting from BasicAssign() algorithm in Figure 15 for n players, having $n!$ columns, one for each sequence, and $(n!)^{(n-1)}/n!$ rows.

Output: modified assigns table such that batches of configurations are reassigned ("shifted") to different sequences in order to evenly distribute token positions (Condition 4).

```
1. let shift = 1
2. let row = n!
3. for x = 1 to shift do:                                   // increasing number of shifts
      4. let temp = assigns[row+1, n!-1] … assigns[row+n!, n!-1]   // save batch of configurations
      5. for col = n!-2 down to 0 do:
            6. move assigns[row+1, col] … assigns[row+n!, col]      // shift right one column
               to assigns[row+1, col+1] … assigns[row+n!, col+1]
      7. move temp to assigns[row+1, 0] … assigns[row+n!, 0]        // store in first column
8. shift = shift+1
9. row = row+n!
10. if row < (n!)^(n-1) then go to step 3
```

Figure 17. ShiftGroups() Algorithm redistributes tokens in the assigns table from BasicAssign() in Figure 15, by reassigning batches of configurations to other sequences in a rotary horizontal shift.

Example. Figure 9 shows the contents of assigns before ShiftGroups(), listed in Figure 17, executes and Figure 11 shows the contents of assigns afterwards. The rightmost column in the before listing (Figure 9) has been highlighted to show how batches of configurations are shifted in ShiftGroups(). For illustration, rows are grouped and numbered I through VI where each group has *n*! rows. Group I has no shifting performed. Group II has a single right shift performed. The effect of shifting a group involves moving the batch from column [0] to column [1], the batch from column [1] to column [2], and so on. The batch in the last column [5] shifts to the first column [0]. As the groups advance, more iterations of shifting is done. Group III has two right shifts performed; Group IV has three, and so on. Group VI, the final group, has five shifts.

Lemma D. Executing the BasicAssign() algorithm with *n* players, where each player has *n*! token choices, and then executing ShiftGroups() produces a table of configurations assigned to sequences that satisfies Conditions 1, 2, 3 and 4.

Proof sketch.
BasicAssign() satisfies Conditions 1, 2 and 3; see Lemma C. The shifting operations performed by ShiftGroups() does not alter the number of configurations, choices or tokens, or the fact that all configurations are distributed evenly across the sequences. For each player having its tokens specific to a sequence, the effect of redistributing *n*! batches of these configurations to every other sequence assures the tokens are no longer specific to a particular sequence and that the tokens are evenly distributed across the sequences. Further, the sequences themselves, by virtue of being permutations, distribute the assigned tokens across each token position.
□

7. The nested loops in BasicAssign() provide another regularity in the frequency at which a player's tokens are cycled that is problematical. Because BasicAssign() assigns batches of configurations to the same sequence, where each batch contains *n*! configurations, a player having tokens cycled at a frequency of 1 token for each configuration (e.g., Player 1) is vulnerable to collusion by *n*-1 players, because all of its tokens appear in the configurations of the batch and all the configurations in the batch are assigned to the same sequence. This makes the player's choice unnecessary to determining the outcome.

Method: ***ShiftRows()***

Input: 2-dimensional array assigns resulting from ShiftGroups() algorithm in Figure 17 for *n* players, having *n*! columns, one for each sequence, and $(n!)^n/n!$ rows.

Output: modified assigns table such that configurations are reassigned ("shifted") to different sequences on the same row in order to combat collusion (Condition 5).

```
1. let shift = 0
2. for row = 1 to (n!)^(n-1) do:
    3. for x = 1 to shift mod n! do:
        4. let temp = assigns[row, n!-1]
        5. for col = n!-2 down to 0 do:
            6. move assigns[row, col] to assigns[row, col+1]
        7. move temp to assigns[row, 0]
    8. shift = shift+1
```

Figure 18. ShiftRows() Algorithm redistributes configurations that were originally assigned as a batch of size *n*!. to one sequence so that the configurations are reassigned across all sequences.

8. The ShiftRows() algorithm, listed in Figure 18, provides anti-collusion protection by redistributing configurations within batches (of size $n!$) across sequences. Configurations within the batch are redistributed so that one configuration in the batch is assigned to each sequence. Without this redistribution, each batch is of the form $(v_{1x}, v_2, \ldots, v_n)$ where $v_2,\ldots,v_n$ are the same exact tokens in each configuration in the batch, and v_{1x} for $x=1,\ldots,n!$ are the tokens of Player 1. If Players 2 through n all collude and select tokens $v_2,\ldots,v_n$, then no matter the token selected by Player 1, the same sequence results. By redistributing the configurations within the batch, such collusion is thwarted.

Example. Figure 11 shows the contents of the assigns array after ShiftGroups(), listed in Figure 17, executes for $n=3$ players, with each player having 6 choices. Notice the topmost batch of $n!$ configurations assigned to sequence [5] abc. Each of these configurations are of the form (a_j, b_6, c_1) for $j=1,\ldots,6$. If Player b selects b_6 and Player a selects c_1, the outcome is abc no matter the selection of Player a. Figure 13 shows the contents of the assigns array after ShiftRows(), listed in Figure 18, executes. The configurations that had previously been within the batch are now distributed across the sequences. Now, no two players can dictate the outcome. As long as any one player makes a random local choice, the outcome will be randomly selected.

Lemma E. Executing the BasicAssign() algorithm with n players, where each player has $n!$ token choices, and then executing ShiftGroups() followed by ShiftRows() produces a table of configurations assigned to sequences that satisfies Conditions 1, 2, 3, 4, and 5.

Proof sketch.
BasicAssign() and ShiftGroups() satisfies Conditions 1, 2, 3 and 4; see Lemma D. The shifting of configuration batches across sequences performed in ShiftRows() does not alter the number of configurations, tokens, or choices, or the fact that all configurations are evenly distributed across all sequences. Token position assignments are preserved because specific tokens are further distributed evenly from within a batch to a single sequence to one configuration for each sequence. Therefore, Condition 4 is satisfied.

Finally, there no longer exists any set of configurations of size $n!$ assigned to the same sequence in which all but one token remains the same. Further, for every subset of tokens, all combinations of the remaining tokens are distributed across the sequences. This is guaranteed by the frequencies of the cycles in the original assignments and the subsequent shifting.
□

9. Figure 19 provides the complete function that sequentially executes BasicAssign(), ShiftGroups(), and then ShiftRows(). The function is called Table-*f*. While the construction of Table-*f* in this subsection provided insight into how to satisfy the conditions, Table-*f* itself is not practical for real-world use. Listing all possible input-outcome pairs is prohibitive, even for relatively small values of n. Figure 20 shows how the size of the table (which has k^n cells) grows as n increases when $k=n!$. More practical functions are needed, so after the introduction of the class of RandomOrder functions and some related theoretical discussion in the next subsections, attention returns to examining RandomXOR (bitwise exclusive-OR from Section 2.3) and ModSum (modulus of the sum from Section 2.4) as possible real-world solutions.

Method: ***Table-f***

Input: (1) n players, $n \geq 2$; (2) each player has $k=n!$ choices; (3) player choices are enumerated as tokens v_{px} for player p's choice of token x, for $1 \leq x \leq n!$; and, (4) an array of the permutations of players named *sequences* having one permutation per array position.

Output: a table (2-dimensional array) named assigns having $n!$ columns, one for each sequence, and $(k^n/n!)$ rows. A cell, assigns[*row*, *col*], contains a configuration assigned to the sequence associated with *col*. The assignments of configurations to sequences in assigns satisfies Conditions 1, 2, 3, 4 and 5.

1. *BasicAssigns()* // see Figure 15
2. *ShiftGroups()* // see Figure 17
3. *ShiftRows()* // see Figure 18

Figure 19. Table-*f*, a RandomOrder function.

n	k	k^n	$n*k$	$n!$	$k^n \bmod n!$	$k^n \bmod n*k$
2	2	4	4	2	0	0
3	6	216	18	6	0	0
4	24	331776	96	24	0	0
5	120	24883200000	600	120	0	0
6	720	139314069504000000	4320	720	0	0
7	5040	826064112539035000000000000	35280	5040	0	0

Figure 20. Given n players, each player having $n!$ choices, the number of configurations (k^n) grows quickly ever for small values of n.

4.2. RandomOrder Functions

1. This subsection introduces the class of RandomOrder functions. Let TOKS be the set of player choices, CONFIGS be the set of all possible configurations realized from the combinations of those choices, and SEQS be the set of the permutations of the players. Let f be a function from TOKS to SEQS (f:TOKS→SEQS) satisfying Conditions 1, 2, 3, 4 and 5. Figure 21 denotes these sets and the mapping of f.

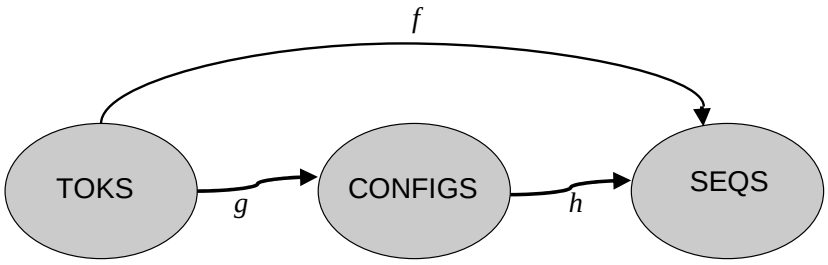


Figure 21. A RandomOrder function (*f*) maps player choices (TOKS) to sequences (SEQS). *f* can also be the composition of *h*, which maps configurations to sequences (SEQS), and *g*, which maps player choices (TOKS) to configurations (CONFIGS).

2. For n players, a **RandomOrder function**, f, is an n-ary function of the form $f(v_1,\ldots,v_n) = r$, where f satisfies Conditions 1, 2, 3, 4 and 5, $v_1 \in$ TOKS is a token chosen by Player 1, …, and $v_n \in$ TOKS is a token chosen by Player n, and $r \in$ SEQS is a permutation of the n players. The inverse of f, written f^{-1}, is defined such that f^{-1}:SEQS→TOKS and $v_1,\ldots,v_n \in f^{-1}(f(v_1,\ldots,v_n))$ for $v_1 \in$ TOKS, …, $v_n \in$ TOKS.

Example. Figure 13 lists in tabular form a RandomOrder function, *f*, for 3 players, a, b, and c, where each player has 6 choices, $a_1,\ldots,a_6,b_1,\ldots,b_6,c_1,\ldots,c_6$. It shows that $f(a_1, b_3, c_5)=cba$; $f(a_2, b_1, c_2)=cab$; and, $f(a_1, b_1, c_1)=cba$. Figure 19 defines Table-*f*, which is a RandomOrder function that generates the table in Figure 13.

3. Players select tokens from subsets of TOKS. Given *n* players, each player P_i selects tokens from the set $TOKS_i$, $1 \le i \le n$ and $TOKS = \bigcup_{j=1}^{n} TOKS_j$.

4. Let *g* be an *n*-ary function on $TOKS_1 \times \ldots \times TOKS_n$ (g:TOKS $\rightarrow$CONFIGS) such that *g* maps all legal combinations of the elements of TOKS to CONFIGS. The inverse of *g*, written g^{-1}, is defined such that g^{-1}:CONFIGS$\rightarrow$TOKS and $v_1,\ldots,v_n \in g^{-1}(g(v_1,\ldots,v_n))$ for $v_1 \in$TOKS, …, $v_n \in$TOKS.

5. Let *h* be a function, *h*:CONFIGS$\rightarrow$SEQS, satisfying Conditions 4 and 5. The function *h* is responsible for assigning configurations to sequences. The inverse of *h*, written h^{-1}, is defined such that h^{-1}:SEQS$\rightarrow$CONFIGS and $c \in h^{-1}(h(c))$ for $c \in$CONFIGS.

6. A RandomOrder function, *f*, is the composition of *h* and *g* defined by $f(v_1,\ldots,v_n) = h(g(v_1,\ldots,v_n))$. The mappings of these functions are shown in Figure 21. It is sometimes convenient to think of the properties of *f* by considering those of *g* and *h* separately.

4.3. Theoretical Test of Randomness

1. The primary goal of a RandomOrder function is to assist *n* players in selecting permutations of themselves without bias. It is important to therefore predict how well a RandomOrder function will behave. Theorem F claims that a RandomOrder function can select a permutation with probability $1/n!$.

Theorem F. Let *f* be a RandomOrder function for *n* players, each player having *n*! local choices that combine to jointly select one of *n*! permutations of themselves. Let v_1 be the choice made by Player 1, …, v_n be the choice made by Player *n*. Assuming at least one player, Player *i*, selects v_i randomly, $1 \le i \le n$, the probability of a permutation, $r_i = f(v_1,\ldots,v_n)$, being selected is $1/n!$ for $i=1,\ldots,n!$.

Proof.
This is a proof by contradiction. There are several cases to consider. The definitions found in Section 4.2 and the conditions described in Section 3.2 are used throughout this proof.

Assume the probability of $r_i = f(v_1,\ldots,v_n)$ being selected is not $1/n!$.

Case 1. The number of sequences is not *n*!.
By definition, SEQS is the set of all permutations of the players. For *n* players, |SEQS| = *n*!.

Case 2. Either: (a) not all combinations of tokens appear in CONFIGS; or, (b) not all combinations of tokens map from CONFIGS to SEQS.
Let CONFIGS′ be the union of all $h^{-1}(r_j)$ for all $r_j \in$ SEQS. The idea is to determine the size of CONFIGS′ to see if any combinations are missing. Because *f* is a RandomOrder function, *f* satisfies Condition 4: the number of occurrences of each token t_j appearing in each position p_x is count(t_j, p_x) = $k^n/(k \bullet n)$, for $j=1,\ldots,k$ and $x=1,\ldots n$. It is given that each player has $k=n!$ tokens. So, |CONFIGS′| = $\sum\sum$ count(t_j, p_x) = $k^n/(k \bullet n) = (k^n/(k \bullet n)) \bullet (k \bullet n) = k^n$. But k^n is also the total number of possible combinations of token choices, so all combinations are present and mapped.

Case 3. Not all elements of CONFIGS are evenly distributed over the elements of SEQS.
If so, $|h^{-1}(r_i)| \neq |h^{-1}(r_j)|$ for some $r_i, r_j \in$ SEQS. Let $x_1 = |h^{-1}(r_i)|$ and $x_2 = |h^{-1}(r_j)|$. The idea is to determine whether $x_1=x_2$. Let $h(c_m)=r_i$, where $c_m \in$ CONFIGS and $c_m=(v_1,\ldots,v_{n-1},v_{ny})$ for $v_1 \in TOKS_1$, $\ldots,v_{n-1} \in TOKS_{n-1}$, $v_{ny} \in TOKS_n$. Because *f* satisfies Condition 5, there exists *n*! versions of c_m, one for each $y=1,\ldots,n!$, and one of each of these is mapped to a sequence. One version is mapped to r_i and another version to r_j. This condition holds for each player's contribution to each configuration. So, $x_1 = x_2 = (k^n)/n!$.

Case 4. There are not enough tokens for player choices to combine to evenly distribute over the elements of CONFIGS.
It is given that each player has *n*! choices. And *f*, as a RandomOrder function, satisfies Condition 3. So, Lemma B assures there are sufficient player choices.

Case 5. Not all combinations of tokens map from TOKS to CONFIGS.
It is given that each player has *n*! choices. The number of combinations of size *n* (" *n*-combinations") of these choices is $(n!)^n$. The idea is to compare this to the size of CONFIGS. In the discussion of Case 2 above, it was shown that |CONFIGS| = $(n!)^n$, thereby accounting for all possible *n*-combination of tokens.

Case 6. Not all combinations of TOKS are evenly distributed over the elements of CONFIGS.
From the definition of *g*, and the discussion in Case 5, all *n*-combinations of tokens appear in CONFIGS and |CONFIGS| = k^n. Therefore, for every $v_1,\ldots,v_n$, where $v_1 \in TOKS_1,\ldots,v_n \in TOKS_n$, $g(v_1,\ldots,v_n) \in$ CONFIGS. And conversely, because |TOKS| < |CONFIGS|, there cannot exist $(v_1,\ldots,v_n) \in$ CONFIGS such that $|g^{-1}(g(v_1,\ldots,v_n))| > 1$ else *g* would not be a function.
□

2. Another way to test a RandomOrder function is through empirical experimentation. Results of specific statistical tests are reported in Appendix B.

4.4. The RandomSelect Protocol

1. RandomOrder functions can be used within the RandomSelect protocol listed in Figure 22 to allow *n* players to make local choices that combine to make a joint random selection with uniform distribution of outcomes to choices. Collisions are avoided, and the protocol performs properly if at least one player makes local decisions randomly.

Preliminary round: Each player receives a table of assignments of sequences, a list of tokens from which to make selections, and a RandomOrder function.
Round of play: Each player selects a token, broadcasts the chosen token, and records all tokens broadcast by all players.
Aftermath: Each player applies the RandomOrder function to the broadcast tokens to determine the joint selection.

Figure 22. RandomSelect Protocol

2. In a preliminary round, each player receives an enumeration of the permutations, a set of tokens from which players make selections, and a RandomOrder function. Once the preliminary round concludes, there can be an indefinite number of plays with no further modification of preliminary materials. The protocol has one round of play. Each player selects one of its tokens and broadcasts it. The final outcome is determined by applying the RandomOrder function to the tokens broadcast by each player.

4.5. Collusion by Players

1. If all players trust one another, then there is no reason to use RandomSelect. One player can simply use a random number generator on her local machine and broadcast the results. So, RandomSelect is used in situations of distrust. In this section, the effect of different forms and amounts of distrust on RandomSelect are examined.

2. As was discussed in Section 3.2 regarding Condition 5, Anti-Collusion, and proven in Theorem F, RandomSelect can compute properly if at least one player makes a local choice randomly. For *n* players therefore, RandomSelect can compute properly even if *n*-1 playes collude. But if all *n* players collude, there is no randomness at all.

3. Attention must be paid to "inadvertent collusion." This occurs when players are not necessarily joining together a priori to agree on choices, but when players are not making local choices randomly. In inadvertent collusion, players exhibit bias in making their local choices. For example, suppose for convenience, each player independently adopts a strategy of selecting the first token on the first round of play, the second token on the second round of play, and so on. The outcomes can then be predicted! While the RandomSelect protocol protects against the traditional multiparty view of collusion (a malicious attack), care must be taken that at least one non-colluding player does not engage in inadvertent collusion.

5. Results

Sections 2.3 and 2.4 provided informal discussions of two operations to consider as RandomOrder functions. These were RandomXOR, which used bitwise exclusive-or, and ModSum, which used the modulus of the sum. These functions are now re-examined in this section to determine whether they qualify as RandomOrder functions.

5.1. RandomXOR Revisited

1. Consider using RandomXOR (first introduced in Section 2.3) as a RandomOrder function in the RandomSelect protocol. Given n players, each player P_i, for i=1,…,n, selects a number v_i (as a token), where $1 \le v_i \le n!$, and broadcasts it. The outcome is determined by each player performing bitwise exclusive-or on the broadcast values of all players.

2. When $n > 2$, as was demonstrated in Section 2.3, not all combinations of tokens map to values in the range 1,…,n!. Those that do not map are collisions. In RandomXOR, the total number of combinations is determined by the number of significant bits required to represent (n!-1), assuming the range of token values is 0,…,(n!-1) rather than 1,…,n!. The constraint $2^m = n!$ is satisfied only when n=2 (or m=1). There is no other integer value satisfying this constraint. As a result, when n >2, RandomXOR fails Condition 4 (Token Position) and Condition 5 (Anti-Collusion).

Example. Figure 23(a) lists all non-collision combinations possible using RandomXOR for 3 players (a, b and c), with each player selecting a value between 1 and 6. Of the 216 possible configurations, only 168 (or 78%) are not collisions. Figure 23(b) shows that tokens are not evenly distributed over token positions.

5.2. ModSum Revisited

1. Theorem G claims that ModSum (first introduced in Section 2.4) is a RandomOrder function. As such, ModSum can be used in RandomSelect as follows. Given n players, each player P_i selects a number v_i (as a token), where $0 \le v_i \le (n!-1)$, and broadcasts it. The outcome is determined by each player computing $(\sum v_i) \bmod n!$ for i=1,…,n.

Theorem G. For n players, $n \ge 2$, with each player P_i selecting a token v_i, where $0 \le v_i \le (n!-1)$ and i=1,…,n, ModSum(v_1,…v_n) is a RandomOrder function.

Proof.
Condition 2 is trivially satisfied: $k=k_1=\ldots=k_n=n!$.

Condition 1. Let $r \equiv (k_1 \bullet \ldots \bullet k_n) \bmod n!$ and show r is 0. Because $k=k_1=\ldots=k_n=n!$, $r \equiv n!^n \bmod n$. So, r is 0.

Condition 3. Let $r \equiv k^n \bmod (n \bullet k)$ and show r is 0. Because k is $n!$, $r \equiv n!^n \bmod (n \bullet n!)$. So, r is 0 for $n \ge 2$.

Conditions 4 and 5. Consider configurations of the form (v_1,…,v_{n-1},v_{nx}) for x=0,…,(n!-1). Let D be the constant $v_1+\ldots+v_{n-1}$. These configurations are mapped to sequences based on $(D+x) \bmod n!$ for x=0,…,(n!-1). Therefore, these configurations map one-to-one to sequences.
□

Example. Figure 24(a) lists all combinations using ModSum for 3 players (a, b and c), with each player selecting a value between 1 and 6. There are no collisions. Figure 23(b) shows that tokens are evenly distributed across token positions.

2. The outcomes of the RandomOrder function Table-*f* can be equivalently expressed as a variant of ModSum. Given n players, $n \geq 2$, with each player P_i selecting a token v_i, where $0 \leq v_i \leq (n!-1)$ and $i=1,\ldots,n$, and the permutations of the players enumerated in an array sequences[*pos*] for $pos=0,\ldots,(n!-1)$, then the outcomes from Table-*f* can be expressed as sequences[*pos*] such that $0 \equiv ((\sum v_i) - (n+pos)) \bmod n!$.

Example. Figure 13 contains the assignments of Table-f for 3 players, a, b, and c, with each player having 6 choices. Let sequences[0],…,sequences[5] be the enumeration of the permutations of the players, as they appear as column headings in Figure 13. The assignments of configurations to sequences shown in Figure 13 can be derived by the following algorithm:

1. **let** *sum* = $\sum_{i=1}^{n} v_i$
2. **if** ((*sum*-3) mod 6) is 0 **then return** sequences [0].
3. **else if** ((*sum*-4) mod 6) is 0 **then return** sequences [1].
4. **else if** ((*sum*-5) mod 6) is 0 **then return** sequences [2].
5. **else if** ((*sum*-6) mod 6) is 0 **then return** sequences [3].
6. **else if** ((*sum*-7) mod 6) is 0 **then return** sequences [4].
7. **else if** ((*sum*-8) mod 6) is 0 **then return** sequences [5].

6. Future Work

1. Randomness has proven itself useful in a wide variety of applications, algorithms and protocols. This writing ends with a historical survey of randomness in computing because the work presented herein introduced randomness into secure multiparty computation in a new way. Perhaps this will be another venue in which randomness will prove itself useful.

2. The construction of machines that can generate a sequence of random values dates back as early as 1939 [13]. Shortly after computers were introduced, methods were sought for efficiently providing random numbers to computer programs [14]. By 1969, random numbers allowed computers to simulate natural phenomena, provide random samples, solve complicated numerical problems, and develop optimal decision-making strategies.

3. Among the first pseudo-random number generators for computers was the linear congruential algorithm [14], which has the form of ModSum. As random number generators moved beyond being the modulus of a sum, future work on RandomOrder functions promises to reveal other kinds of formulations.

4. In the late 1980's studies on the paradigm of "randomized algorithms" ignited. These are algorithms in which random choices are made during the execution of the algorithm. Significant benefits emerged. These can be grossly described as providing faster and/or simpler algorithms than their deterministic counterparts [15].

5. More recently, "randomized protocols" have been used for many tasks, including contract signing [16] and asynchronous consensus [17]. In randomized protocols one or more players involved in the protocol make random choices.

6. The work presented in this paper extends the notion of randomized protocols in such a way that a subset of 2 or more players make local decisions that combine to jointly make a random choice of a permutation of themselves in the absence of a third party and in the face of colluding players. Future work may be more generalized to allow a set of players to jointly make random choices based on some probability distribution. This work terms these protocols **randomized multiparty computation**. An aim is for randomized multiparty computations to provide to multiparty computation the kinds of complexity efficiencies, simplicities, and insights that randomized algorithms afforded deterministic algorithms.

(a)

cba [1]	bca [2]	cab [3]	acb [4]	bac [5]	abc [6]
1: c1 b1 a1	2: b1 c1 a2	3: c1 a3 b1	4: a4 c1 b1	5: b1 a5 c1	6: a6 b1 c1
8: c1 b2 a2	7: b2 c1 a1	13: c1 a1 b3	18: a6 c1 b3	12: b2 a6 c1	11: a5 b2 c1
15: c1 b3 a3	30: b5 c1 a6	24: c1 a6 b4	19: a1 c1 b4	25: b5 a1 c1	16: a4 b3 c1
22: c1 b4 a4	35: b6 c1 a5	34: c1 a4 b6	33: a3 c1 b6	32: b6 a2 c1	21: a3 b4 c1
29: c1 b5 a5	37: b1 c2 a1	45: c2 a3 b2	46: a4 c2 b2	42: b1 a6 c2	26: a2 b5 c1
36: c1 b6 a6	44: b2 c2 a2	50: c2 a2 b3	53: a5 c2 b3	47: b2 a5 c2	31: a1 b6 c1
38: c2 b1 a2	51: b3 c2 a3	59: c2 a5 b4	56: a2 c2 b4	52: b3 a4 c2	41: a5 b1 c2
43: c2 b2 a1	58: b4 c2 a4	64: c2 a4 b5	63: a3 c2 b5	57: b4 a3 c2	48: a6 b2 c2
66: c2 b5 a6	65: b5 c2 a5	73: c3 a1 b1	78: a6 c3 b1	62: b5 a2 c2	61: a1 b5 c2
71: c2 b6 a5	72: b6 c2 a6	80: c3 a2 b2	83: a5 c3 b2	67: b6 a1 c2	68: a2 b6 c2
75: c3 b1 a3	81: b2 c3 a3	87: c3 a3 b3	88: a4 c3 b3	82: b2 a4 c3	76: a4 b1 c3
85: c3 b3 a1	86: b3 c3 a2	94: c3 a4 b4	93: a3 c3 b4	89: b3 a5 c3	90: a6 b3 c3
96: c3 b4 a6	95: b4 c3 a5	101: c3 a5 b5	98: a2 c3 b5	92: b4 a2 c3	91: a1 b4 c3
106: c3 b6 a4	100: b5 c3 a4	108: c3 a6 b6	103: a1 c3 b6	99: b5 a3 c3	105: a3 b6 c3
112: c4 b1 a4	118: b2 c4 a4	114: c4 a6 b1	109: a1 c4 b1	117: b2 a3 c4	111: a3 b1 c4
126: c4 b3 a6	125: b3 c4 a5	119: c4 a5 b2	116: a2 c4 b2	122: b3 a2 c4	121: a1 b3 c4
127: c4 b4 a1	128: b4 c4 a2	124: c4 a4 b3	123: a3 c4 b3	131: b4 a5 c4	132: a6 b4 c4
141: c4 b6 a3	135: b5 c4 a3	129: c4 a3 b4	130: a4 c4 b4	136: b5 a4 c4	142: a4 b6 c4
149: c5 b1 a5	150: b1 c5 a6	134: c4 a2 b5	137: a5 c4 b5	145: b1 a1 c5	146: a2 b1 c5
156: c5 b2 a6	155: b2 c5 a5	139: c4 a1 b6	144: a6 c4 b6	152: b2 a2 c5	151: a1 b2 c5
169: c5 b5 a1	160: b3 c5 a4	154: c5 a4 b2	153: a3 c5 b2	159: b3 a3 c5	174: a6 b5 c5
176: c5 b6 a2	165: b4 c5 a3	161: c5 a5 b3	158: a2 c5 b3	166: b4 a4 c5	179: a5 b6 c5
186: c6 b1 a6	170: b5 c5 a2	164: c5 a2 b4	167: a5 c5 b4	173: b5 a5 c5	181: a1 b1 c6
191: c6 b2 a5	175: b6 c5 a1	171: c5 a3 b5	172: a4 c5 b5	180: b6 a6 c5	188: a2 b2 c6
196: c6 b3 a4	185: b1 c6 a5	184: c6 a4 b1	183: a3 c6 b1	182: b1 a2 c6	195: a3 b3 c6
201: c6 b4 a3	192: b2 c6 a6	198: c6 a6 b3	193: a1 c6 b3	187: b2 a1 c6	202: a4 b4 c6
206: c6 b5 a2	205: b5 c6 a1	199: c6 a1 b4	204: a6 c6 b4	210: b5 a6 c6	209: a5 b5 c6
211: c6 b6 a1	212: b6 c6 a2	213: c6 a3 b6	214: a4 c6 b6	215: b6 a5 c6	216: a6 b6 c6

(b)

Player a

Token	Pos 1	Pos 2	Pos 3
a_1	10	8	10
a_2	8	10	10
a_3	10	10	8
a_4	10	10	8
a_5	8	10	10
a_6	10	8	10

Player b

Token	Pos 1	Pos 2	Pos 3
b_1	8	12	8
b_2	12	8	8
b_3	8	8	12
b_4	8	8	12
b_5	12	8	8
b_6	8	12	8

Player c

Token	Pos 1	Pos 2	Pos 3
c_1	10	8	10
c_2	8	10	10
c_3	10	10	8
c_4	10	10	8
c_5	8	10	10
c_6	10	8	10

Figure 23. (a) Assignments by RandomXOR for 3 players, a, b, and c, with each player having 6 choices. Only 168 of the 216 possible combinations are used. The others are collisions. (b) counts of the number of times the tokens of each player appears in each sequence position.

(a)

cba [0]	bca [1]	cab [2]	acb [3]	bac [4]	abc [5]
4: c1 b1 a4	5: b1 c1 a5	6: c1 a6 b1	1: a1 c1 b1	2: b1 a2 c1	3: a3 b1 c1
9: c1 b2 a3	10: b2 c1 a4	11: c1 a5 b2	12: a6 c1 b2	7: b2 a1 c1	8: a2 b2 c1
14: c1 b3 a2	15: b3 c1 a3	16: c1 a4 b3	17: a5 c1 b3	18: b3 a6 c1	13: a1 b3 c1
19: c1 b4 a1	20: b4 c1 a2	21: c1 a3 b4	22: a4 c1 b4	23: b4 a5 c1	24: a6 b4 c1
30: c1 b5 a6	25: b5 c1 a1	26: c1 a2 b5	27: a3 c1 b5	28: b5 a4 c1	29: a5 b5 c1
35: c1 b6 a5	36: b6 c1 a6	31: c1 a1 b6	32: a2 c1 b6	33: b6 a3 c1	34: a4 b6 c1
39: c2 b1 a3	40: b1 c2 a4	41: c2 a5 b1	42: a6 c2 b1	37: b1 a1 c2	38: a2 b1 c2
44: c2 b2 a2	45: b2 c2 a3	46: c2 a4 b2	47: a5 c2 b2	48: b2 a6 c2	43: a1 b2 c2
49: c2 b3 a1	50: b3 c2 a2	51: c2 a3 b3	52: a4 c2 b3	53: b3 a5 c2	54: a6 b3 c2
60: c2 b4 a6	55: b4 c2 a1	56: c2 a2 b4	57: a3 c2 b4	58: b4 a4 c2	59: a5 b4 c2
65: c2 b5 a5	66: b5 c2 a6	61: c2 a1 b5	62: a2 c2 b5	63: b5 a3 c2	64: a4 b5 c2
70: c2 b6 a4	71: b6 c2 a5	72: c2 a6 b6	67: a1 c2 b6	68: b6 a2 c2	69: a3 b6 c2
74: c3 b1 a2	75: b1 c3 a3	76: c3 a4 b1	77: a5 c3 b1	78: b1 a6 c3	73: a1 b1 c3
79: c3 b2 a1	80: b2 c3 a2	81: c3 a3 b2	82: a4 c3 b2	83: b2 a5 c3	84: a6 b2 c3
90: c3 b3 a6	85: b3 c3 a1	86: c3 a2 b3	87: a3 c3 b3	88: b3 a4 c3	89: a5 b3 c3
95: c3 b4 a5	96: b4 c3 a6	91: c3 a1 b4	92: a2 c3 b4	93: b4 a3 c3	94: a4 b4 c3
100: c3 b5 a4	101: b5 c3 a5	102: c3 a6 b5	97: a1 c3 b5	98: b5 a2 c3	99: a3 b5 c3
105: c3 b6 a3	106: b6 c3 a4	107: c3 a5 b6	108: a6 c3 b6	103: b6 a1 c3	104: a2 b6 c3
109: c4 b1 a1	110: b1 c4 a2	111: c4 a3 b1	112: a4 c4 b1	113: b1 a5 c4	114: a6 b1 c4
120: c4 b2 a6	115: b2 c4 a1	116: c4 a2 b2	117: a3 c4 b2	118: b2 a4 c4	119: a5 b2 c4
125: c4 b3 a5	126: b3 c4 a6	121: c4 a1 b3	122: a2 c4 b3	123: b3 a3 c4	124: a4 b3 c4
130: c4 b4 a4	131: b4 c4 a5	132: c4 a6 b4	127: a1 c4 b4	128: b4 a2 c4	129: a3 b4 c4
135: c4 b5 a3	136: b5 c4 a4	137: c4 a5 b5	138: a6 c4 b5	133: b5 a1 c4	134: a2 b5 c4
140: c4 b6 a2	141: b6 c4 a3	142: c4 a4 b6	143: a5 c4 b6	144: b6 a6 c4	139: a1 b6 c4
150: c5 b1 a6	145: b1 c5 a1	146: c5 a2 b1	147: a3 c5 b1	148: b1 a4 c5	149: a5 b1 c5
155: c5 b2 a5	156: b2 c5 a6	151: c5 a1 b2	152: a2 c5 b2	153: b2 a3 c5	154: a4 b2 c5
160: c5 b3 a4	161: b3 c5 a5	162: c5 a6 b3	157: a1 c5 b3	158: b3 a2 c5	159: a3 b3 c5
165: c5 b4 a3	166: b4 c5 a4	167: c5 a5 b4	168: a6 c5 b4	163: b4 a1 c5	164: a2 b4 c5
170: c5 b5 a2	171: b5 c5 a3	172: c5 a4 b5	173: a5 c5 b5	174: b5 a6 c5	169: a1 b5 c5
175: c5 b6 a1	176: b6 c5 a2	177: c5 a3 b6	178: a4 c5 b6	179: b6 a5 c5	180: a6 b6 c5
185: c6 b1 a5	186: b1 c6 a6	181: c6 a1 b1	182: a2 c6 b1	183: b1 a3 c6	184: a4 b1 c6
190: c6 b2 a4	191: b2 c6 a5	192: c6 a6 b2	187: a1 c6 b2	188: b2 a2 c6	189: a3 b2 c6
195: c6 b3 a3	196: b3 c6 a4	197: c6 a5 b3	198: a6 c6 b3	193: b3 a1 c6	194: a2 b3 c6
200: c6 b4 a2	201: b4 c6 a3	202: c6 a4 b4	203: a5 c6 b4	204: b4 a6 c6	199: a1 b4 c6
205: c6 b5 a1	206: b5 c6 a2	207: c6 a3 b5	208: a4 c6 b5	209: b5 a5 c6	210: a6 b5 c6
216: c6 b6 a6	211: b6 c6 a1	212: c6 a2 b6	213: a3 c6 b6	214: b6 a4 c6	215: a5 b6 c6

(b)

Player a

Token	Pos 1	Pos 2	Pos 3
a_1	12	12	12
a_2	12	12	12
a_3	12	12	12
a_4	12	12	12
a_5	12	12	12
a_6	12	12	12

Player b

Token	Pos 1	Pos 2	Pos 3
b_1	12	12	12
b_2	12	12	12
b_3	12	12	12
b_4	12	12	12
b_5	12	12	12
b_6	12	12	12

Player c

Token	Pos 1	Pos 2	Pos 3
c_1	12	12	12
c_2	12	12	12
c_3	12	12	12
c_4	12	12	12
c_5	12	12	12
c_6	12	12	12

Figure 24. (a) Assignments by ModSum for 3 players, a, b, and c, with each player having 6 choices. (b) counts of the number of times the tokens of each player appears in each sequence position.

Acknowledgements

Much gratitude goes to the members of the Data Privacy Lab for a rich work environment. This work was supported in part by the Data Privacy Laboratory, of the Institute of Software Research International, in the School of Computer Science, at Carnegie Mellon University.

References

1. Goldreich , O. *The Foundations of Cryptography* - Volume 2, Cambridge University Press, (manuscript) to appear May 2004.
2. Cramer, R. and Damgard, I. *Lecture Notes on Multiparty Computation*. (unpublished manuscript), 2002. Available at <http:// www.daimi.au.dk/~ivan/mpc.ps >.
3. Goldreich, O. Micali, S., and Wigderson, A. How to Play any Mental Game | A Completeness Theorem for Protocols with Honest Majority, *Proc. ACM STOC '87*, pp 218-229.
4. Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation, *Proc. ACM STOC '88*, pp 1-10.
5. Chaum, D. Crepeau, C., and Damgard, I. Multiparty Unconditionally Secure Protocols (Extended Abstract), *Proc. ACM STOC '88*, pp. 11-19.
6. Rabin, T and Ben-Or, M. Verifiable Secret Sharing and Multiparty Protocols with Honest majority, *Proc. ACM STOC '89*, pp. 73-85.
7. Beaver, D. Secure Multiparty Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority, *J. Cryptology*, vol. 4 (1991) pp. 75-122.
8. Sweeney, L. and Edoho-Eket, S. *Detecting Bio-Terrorist Attacks and Naturally Occurring Outbreaks Over a Distributed Network While Protecting Privacy and Confidentiality: the PrivaSum Protocol*, Carnegie Mellon University, School of Computer Science, Tech Report, CMU ISRI 04-111. Pittsburgh: April 2004.
9. Centers for Disease Control and Prevention, 2004. Available at http://www.bt.cdc.gov/agent/agentlist.asp
10. Guillemin, J. *Anthrax: The Investigation of a Deadly Outbreak*. Dec. 1999. 367 Univ. of California.
11. Mostashari, F. and Hartman, J. Syndromic surveillance: a local perspective. *Journal of Urban Health*, 80 (2) 2003.
12. Walker, G. and Walker, D. *The Official Rock Paper Scissors Strategy Guide*. Fireside Books, to appear October 2004.
13. Kendall, D. and Babington-Smith, B. Randomness and random sampling numbers, *Journal Royal Statistical Society* 101 pp. 146- 166, 1938. See also *Journal Royal Statistical Society Supplement* 6, pp 51-36, 1939.
14. Knuth, D. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley: Reading, 1969.
15. Motwani, R. and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, New York (NY), 1995.
16. Even, S., Goldreich, O. and Lempel, A. A randomized protocol for signing contracts. *Communications of the ACM*, 28 (6), pp 637 – 647, 1985.
17. Aspnes, J. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16 (2-3), pp 165 – 175, 2003.
18. Abramowitz, M. and Stegun, I. *Handbook of Mathematical Functions*. U.S. Government Printing Office, Washington, DC, 1964.

Appendix A. Example of Trivial Secret Sharing

A motivating example is provided in this section. Trivial secret sharing, a simple multiparty computation, is compared to a randomized version in which players make joint random choices. The purpose is to see what kind of efficiency tradeoffs might be afforded by such randomization. Further motivation stems from using trivial secret sharing (or its randomized alternative) for real-world bio-terrorism surveillance.

1. Consider a set of players and a reporting authority operating over secure channels. Each player holds a private number. The players wish to compute the sum of their inputs in a manner that would preserve the secrecy of their individual values while providing the sum to the reporting authority. The players need to perform the computation even if some of the players collude passively.

2. Trivial secret sharing offers a solution. Rounds are described in Figure W for n players. Each player's private number is represented as a set of n values (called "shares") that sum to the player's original private number. Each pair of players exchanges shares. Then, each player adds the shares it received with its remaining share and sends its total to the reporting authority, who adds the totals it receives and announces the final sum. Communication is $O(n^2)$, characterized by having each player sending shares to every other player. Proof of correctness and of privacy protection can be found in [8].

3. The scenario described above has many real-world applications, including in bio-terrorism surveillance. Several biological agents, such as anthrax, when inhaled have initial symptoms resembling the flu [9, 10]. It is therefore expected that in the early days of an airborne release, victims will believe they have the flu and will behave accordingly. Yet, the earlier authorities realize a biological agent has been released, the more lives likely to be saved. Bio-terrorism surveillance systems therefore seek to determine whether an unusual number of people are acting ill by counting daily hospital visits, over-the-counter medication purchases, numbers of students absent from schools, etc. [11]. The example of monitoring absenteeism at schools is further modeled below, though the described approach is generally analogous to all these monitoring efforts.

Round 1.	Each player generates a set of n random integers that sum to the player's private value. These values are called "shares." Each player sends a distinct share to every other player and keeps one share.
Round 2.	As a consequence of Round 1, each player received a share from every other player. Each player now adds all the shares received to its remaining share and sends the total to the reporting authority.
Round 3.	As a consequence of Round 2, the reporting authority received a total from each player. The reporting authority adds these totals and broadcasts the final sum.

Figure W. Rounds of trivial secret sharing in which n players, each having a private value, provide the sum of the n privately held values to a reporting authority such that the private values are not revealed.

4. The local public health department, as a reporting authority, wants to know whether there are unusual numbers of students absent from local area schools. Schools in the United States are subject to the Family Educational Rights and Privacy Act (FERPA), and as such, may refuse to provide the names of students who are absent because of privacy responsibilities expressed in FERPA. Schools may also consider daily absenteeism counts confidential because of possible political and funding consequences related to school absenteeism. Concerned schools will only participate if both the privacy of the students (no student can be identified) and the confidentiality of school absenteeism (the number of students absent at a particular school) are not revealed.

5. Figure X shows how this real-world problem can be solved with trivial secret sharing. Each of the 6 schools produces a set of 6 random integers that sum to the school's private number. In Figure X(a), School$_1$ has the private value 125. It generates the shares {304, 45, -150, 10, -17, -67}, which sum to 125, and then distributes a distinct share to each of the other schools, keeping one share (304) for itself. As a result of this distribution, each school receives a share from every other school. In Figure X(b). School$_1$ receives shares 23, -134, 75, -45 and 631 from other schools. In the final steps, each school adds the shares it received to its remaining share and then sends the subtotal to Public Health. In Figure X(d), School$_1$ sends the subtotal 854, which is 23-134+75-45+631+304. Public Health then adds the values it receives and broadcasts the final sum (417). Notice that 417 is the sum of the original numbers of absentees from each school (417 = 125+12+34+132+39+75), even though no private values were revealed.

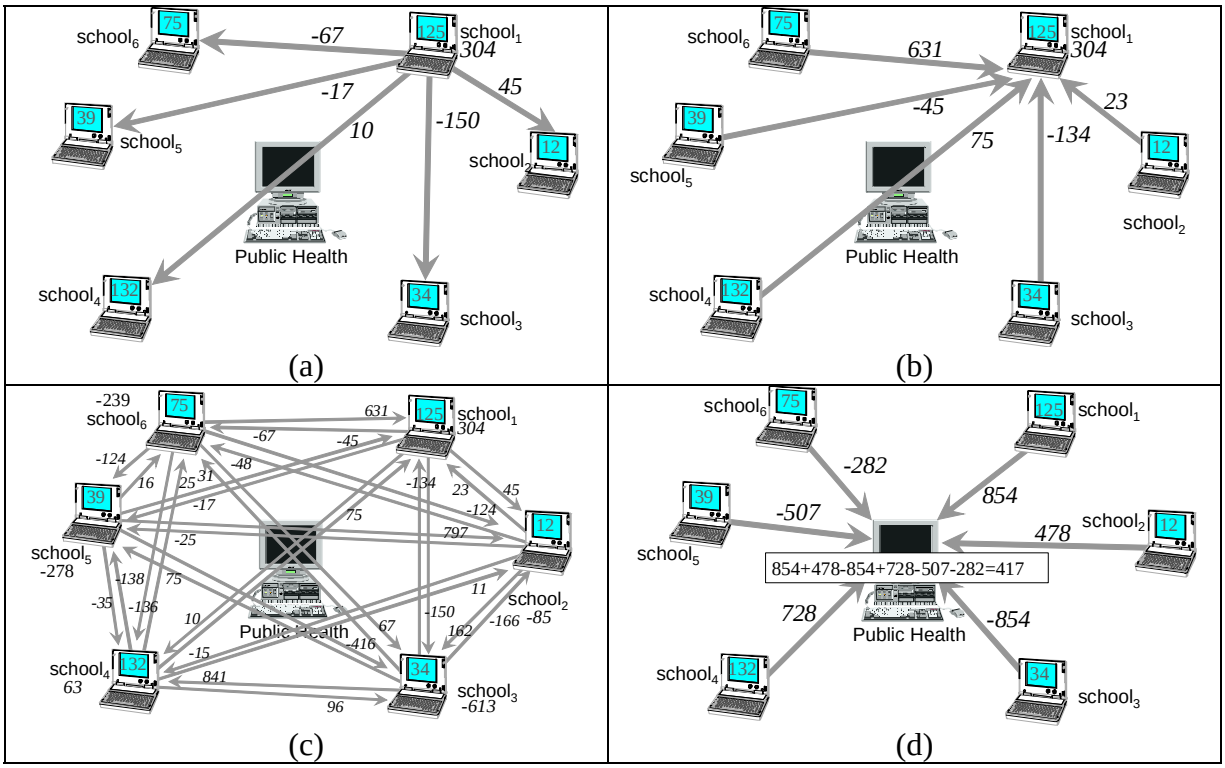


Figure X. Execution of trivial secret sharing by *n*=6 schools. In the first round (a), each school sends a share every other school. So, each school receives a share from every other school (b). Communication is O(n^2) as shown in (c). In the final round (d), each school sends its total to public health where they are added and the final sum broadcast.

6. In terms of communication costs, Figure X(c) shows that each of the n=6 schools sent a share to the other 5 schools, yielding 30 communications, which is $n\bullet(n\text{-}1) \rightarrow O(n^2)$. Additionally as shown in Figure X(d), there were n communications, one from each of the 6 schools to the reporting authority. The broadcast of the final sum required another n communications. In summary, overall communication is $n\bullet(n\text{-}1)+n+n \rightarrow O(n^2)$.

7. Now consider the following alternative approach to trivial secret sharing. Players are arranged sequentially in a "ring" and respond asynchronously around the ring to compute the sum. Figure Y(a) provides an illustration. The ordering of the players around the ring is determined beforehand. The reporting authority starts by providing a random number r_0 to the first player, who in turns adds her private value and sends the sum to the second player, and so on, around the ring. The last player sends his sum to the reporting authority, who subtracts the original random value r_0 to get the actual total.[2] In each round, a single player sends a sum, so for n players and a reporting authority, there are n+1 rounds. The overall number of communications is $O(n)$, which is an improvement over the $O(n^2)$ communication found in the traditional approach.

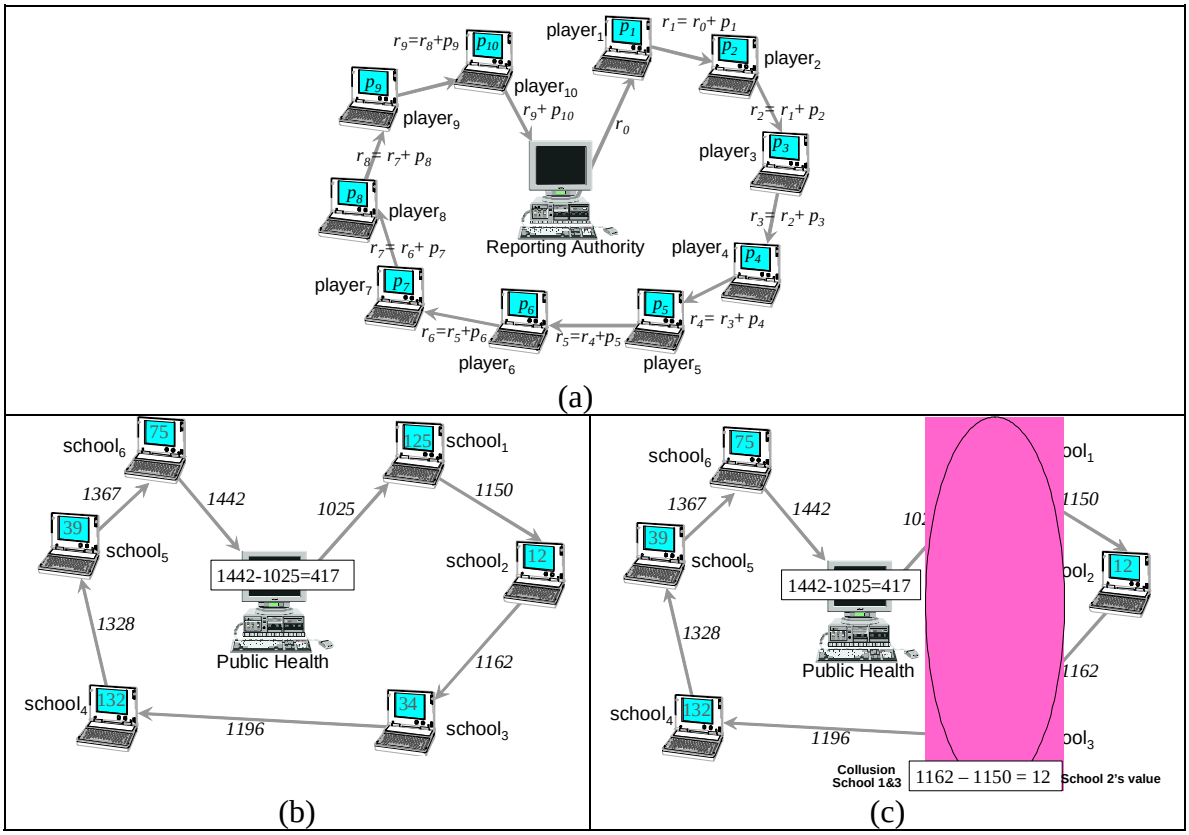


Figure Y. Alternative version of trivial secret sharing consists of asynchronous ring computations as shown in (a). An example with n=6 schools is shown in (b). A ring has O(n) communication. Collusion is possible in a ring computation if surrounded by colluding players, as shown in (c).

[2] The idea of a ring computation was first introduced by Manuel Blum and the theory group at Carnegie Mellon University in casual communications about bio-terrorism surveillance in 2002.

8. However, the version described above can be subject to a collusion attack. Figure Y(b) shows the protocol with 6 schools. $School_2$'s private number is 12. If $School_1$ and $School_3$ collude by subtracting the number $School_1$ provided to $School_2$ (1150) from the number $School_2$ provided to $School_3$ (1162), they can learn $School_2$'s private number (12=1162-1150). This is shown in Figure Y(c).

9. To combat collusion, randomization can be added. The ordering of n players around the ring could be determined randomly. Then, the probability that a non-colluding player will be sandwiched between two colluding players can be determined. The likelihood of successful collusion can be further reduced by having m distinct rings compute in parallel. Under this scheme, each player provides m random values (shares) that sum to her private value, where $1 \leq m \leq n$. The order in which the players are arranged around each of the m rings is determined randomly. During computation, each player provides a share to each ring. The reporting authority adds the totals from all m rings to get the final sum. Overall communication is $O(m \bullet n)$. The value m can be selected based on the amount of probabilistic privacy protection desired. The larger the value of m, the less likely collusion will be successful.

10. Figure Z provides an example of n=6 schools performing the randomized protocol with m=3 rings. Each school has 3 shares, one for each ring. There are $n!$=720 possible ways to possibly order the schools, 3 of which are shown in Figure Z. All 3 rings begin and end with Public Health. The 3 routes are [$School_1$, $School_2$, $School_3$, $School_4$, $School_5$, $School_6$], [$School_4$, $School_2$, $School_1$, $School_6$, $School_5$, $School_3$], and [$School_2$, $School_4$, $School_5$, $School_6$, $School_3$, $School_1$]. In the first round, Public Health sends a random value to $School_1$, $School_4$, and $School_2$ to seed the computations. In the final round, $School_6$, $School_3$, and $School_1$ sends their sums to Public Health, which sums the values received and subtracts the original seeds.

11. In this example, for n players with each player having $1 \leq m \leq n$ shares, the number of communications improved with randomization from $O(n^2)$ to $O(m \bullet n)$, but the actual time to complete the protocol changed from 3 rounds (non-randomized) to n+1 rounds (randomized). Privacy protection changed from deterministic (non-randomized) to probabilistic (randomized).[3]

12. The randomized version assumes random orderings of players can be done with communication no worse than $m \bullet n$. A method for players to jointly select random orderings of themselves is the subject of the remainder of this writing.

[3] PrivaSum is a non-randomized protocol that extends trivial secret sharing for n players, each player having $2 \leq m \leq n$ shares. The m shares are distributed in order to provide probabilistic privacy protection [8]. For m shares, the randomized version above offers better privacy protection than does PrivaSum.

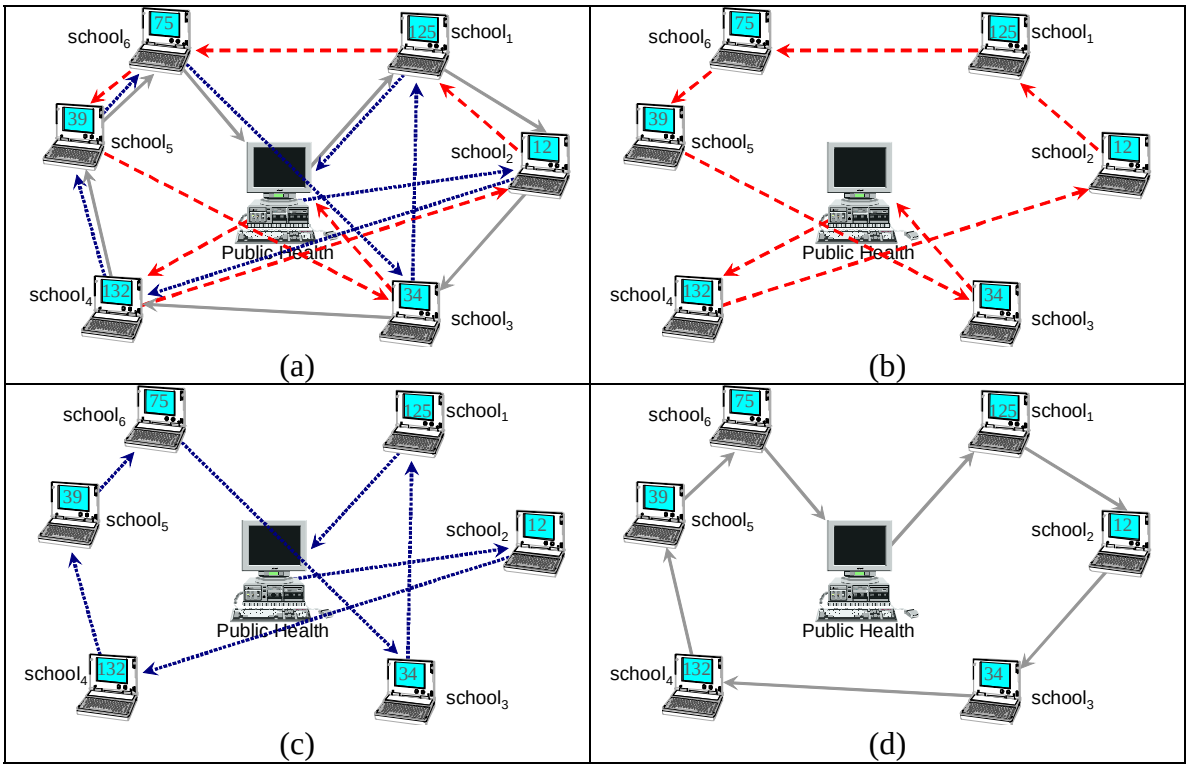


Figure Z Three simultaneous ring computations in which the order of the players in each ring is determined randomly. All 3 rings are shown in (a) and each distinct ring is shown in (b), (c) and (d).

Appendix B. Experimental Tests

This section reports on empirical tests of the RandomOrder function ModSum. The purpose is to determine how random selections are made in practice. Statistics provides some quantitative measures for randomness, but frankly, there is no limit to the number of tests that can be performed. In this section, experimental results are reported for a frequency test and a chi-square test.

B.1. Materials

Materials include a Java implementation of ModSum to simulate play by 3 players. The pseudo-random number generator available through the Java Programming languages is used to make a random local decision by a player.

B.2. Equidistribution or Frequency test

1. The first experiment consisted of 1000 simulated plays in which each player made a random local choice. The requirement is that each sequence should appear as often as any other sequence. Figure A shows the distribution of the results over the possible permutations. The results show a relatively even distribution of selections of permutations (determining whether this is good enough is the subject of the analysis in the next subsection).

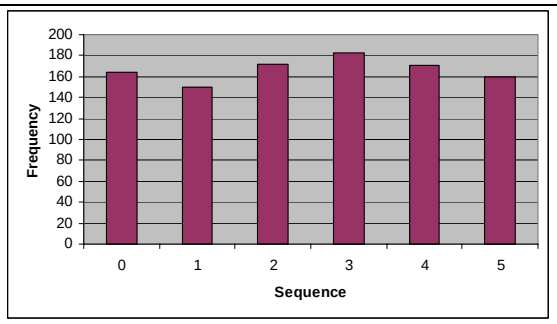


Figure A. Results from 1000 simulated plays of ModSum with 3 players, with each player having 6 choices. All 3 players made local choices randomly.

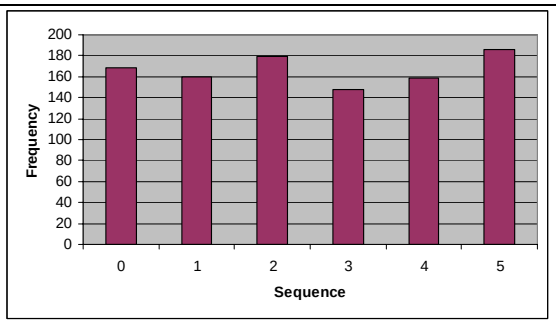


Figure B. Results from 1000 simulated plays of ModSum with 3 players, with each player having 6 choices. The simulation has two players collude by always providing choices 5 and 3. The third player makes random local choices.

2. The second experiment consisted of 1000 simulated plays in which two players collude and make the same choice (), but the third player makes random local choices. The requirement is that each sequence should appear as often as any other sequence. Figure B shows an even distribution of the results over the possible permutations, demonstrating performance when at least one player makes local random choices.

B.3. Chi-square test

1. For n players, Theorems F and G state that the probability of any particular sequence being selected by ModSum is $1/n!$. Using actual observations of outcomes reported in Figure C, the square of the differences between the observed frequencies and the expected frequencies can be computed. This is the basis of the "chi-square" statistic.

2. The Chi-square statistic informs as to how probable or improbable certain sets of outcomes are. A natural way to compute such a statistic is to consider the sum of the squares of the differences between the observed frequencies and the expected frequencies. For example, for 1000 plays of ModSum for 3 players, with each player having 6 choices, the expected frequencies are 1000/6=167. Each sequence is expected to be selected 167 times.

3. The Chi-Square statistic, V, appears below, for *n* players, and a test having *nruns* as the total number of plays for the test, and Y_i is the frequency at which sequence *i* was observed, where *i*=0,…,(*n*!-1).

$$V = \sum_{i=0}^{n!-1} \frac{(Y_i - (nruns)/n!)^2}{(nruns)/n!}$$

4. The test results in Figure C can be summarized as follows. For each of the 9 tests, the following was done. 1000 independent observations of ModSum play were simulated. A count of the number of observations for which play resulted in each of the 6 sequences, labeled [0] to [5], is reported. The Chi-Square statistic, V, is computed, as described above. Then, V is compared with the numbers in a published Table of Values of the Chi-Square Distribution [18], with 5 degrees of freedom. In these tests there are 6 possible outcomes, providing 6-1=5 degrees of freedom. If V is between 95 and 99 percent, or between 1 or 5 percent, the protocol would be considered "suspect." If V lies between 90 and 95 percent, or between 5 and 10 percent, the protocol might be "almost suspect." If at least 2 or 3 of the tests provided results that are suspect, then the protocol would not be regarded as sufficiently random. As shown in Figure C, the outcomes are satisfactory random with respect to the tests.

Sequences	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**	**Test 7**	**Test 8**	**Test 9**
[0]	175	149	163	173	157	178	181	176	178
[1]	170	172	173	173	168	172	150	150	170
[2]	155	186	158	180	177	178	180	162	176
[3]	163	165	164	162	169	153	178	169	139
[4]	147	160	177	154	149	167	150	159	163
[5]	190	168	165	158	180	152	161	184	174
V	6.554	2.719	1.392	2.853	3.627	3.358	5.371	3.988	5.590
p	25 to 50%	50 to 75%	75 to 95%	50 to 75%	50 to 75%	50 to 75%	25 to 50%	50 to 75%	25 to 50%

Figure C. Results from 9 tests. Each test 1000 simulated plays of ModSum with 3 players, with each player having 6 choices, [0] to [5], and each player making local choices randomly. The chi-square statistic ,V, is reported for each test, along with the chi-square distribution, *p*.