ANONYMOUS OPT-OUT AND SECURE COMPUTATION IN DATA MINING

Samuel Steven Shepard

A Thesis

Submitted to the Graduate College of Bowling Green State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE, MASTER OF ARTS

December 2007

Committee:

Ray Kresman, Advisor

Larry Dunning

So-Hsiang Chou

Tong Sun
ABSTRACT

Ray Kresman, Advisor

Privacy preserving data mining seeks to allow users to share data while ensuring individual and corporate privacy concerns are addressed. Recently algorithms have been introduced to maintain privacy even when all but two parties collude. However, exogenous information and unwanted statistical disclosure can weaken collusion requirements and allow for approximation of sensitive information. Our work builds upon previous algorithms, putting cycle-partitioned secure sum into the mathematical framework of edge-disjoint Hamiltonian cycles and providing an additional anonymous “opt-out” procedure to help prevent unwanted statistical disclosure.
# Table of Contents

**CHAPTER 1: DATA MINING**

1.1 The Basics .......................................................... 1
1.2 Association Rule Mining Algorithms ............................ 2
1.3 Distributed Data Mining ......................................... 3
1.4 Privacy-preserving Data Mining ................................. 4
1.5 Secure Sum ......................................................... 5

**CHAPTER 2: HAMILTONIAN CYCLES**

2.1 Graph Theory: An Introduction ................................. 7
2.2 Generating Edge-disjoint Hamiltonian Cycles ................ 8
  2.2.1 Fully-Connected ............................................ 8
  2.2.2 Edge-disjoint Hamiltonian Cycles in Torus .............. 11

**CHAPTER 3: COLLUSION RESISTANCE WITH OPT-OUT**

3.1 Cycle-partitioned Secure Sum .................................. 14
3.2 Problems with CPSS: Statistical Disclosure ................. 21
3.3 Anonymous Opt-Out ............................................. 22
3.4 Anonymous ID Assignment ...................................... 26
3.5 CPSS with Opt-out ............................................... 30

**CHAPTER 4: ADDITIONAL CONSIDERATIONS**

4.1 Numerical Considerations ...................................... 37
4.2 Anonymous ID Assignment Termination ....................... 38
4.3 Communication Complexity ...................................... 41
  4.3.1 CPSS, Algorithm 2 ........................................... 41
  4.3.2 Anonymous ID Assignment, Algorithm 6 ................ 41
## List of Figures

1.1 Secure Sum, $N = 3$ ................................................................. 6  
2.1 Edge-disjoint Hamiltonian Decomposition of a Fully-Connected Graph, $M = 5$ ............ 10  
2.2 Hamiltonian cycles for a $5 \times 3$ Torus Network, generated by Method 4 .................. 12  
3.1 Cycle-partitioned Secure Sum, $M = 5$ .............................................. 17  
3.2 Naïve Opt-out with node $N_3$ withdrawing.............................................. 25  
3.3 Simple Anonymous ID Assignment, 3 nodes.............................................. 28  
3.4 CPSOO: Inputs and outputs of an arbitrary node, $M = 5$ nodes & $C = 2$ cycles........... 33  
4.1 AIDA—Maximum Iterations for 10,000 trials............................................ 39  
4.2 AIDA—Average Iterations for 10,000 trials............................................. 40
List of Algorithms

1. Edge-disjoint Hamiltonian Cycle Generation .................................................. 9
2. CPSS: Cycle-Partitioned Secure Sum ................................................................. 16
3. RandomlyPartition(A, C) ................................................................................. 16
4. CPSS with Naive Opt-Out .................................................................................. 24
5. Simple Anonymous ID Assignment ................................................................... 27
6. AIDA: Anonymous ID Assignment, with $K$-collusion Resistance .................... 29
7. RandomlySelectSlot(S, A, Len, C) ................................................................. 30
8. CPSOO: Cycle-partitioned Secure Sum with Opt-out ........................................ 32
9. RandomlyBitPartition(B, Q) ............................................................................. 33
List of Tables

2.1 Example Cycle Generation .............................................. 10
2.2 Method 4—Example Cycle Generation ............................. 13
CHAPTER 1

DATA MINING

1.1 The Basics

Data is collected every day by a varied and diverse array of individuals—from governments to grocery stores, from game enthusiasts to research scientists. Sometimes, though, the full range of useful information in a data set is not immediately obvious to the observer. In order to solve this problem, the field of data mining provides techniques for new knowledge discovery—finding patterns in the data set such as classifiers and association rules that may be useful to the data miner. We consider only the latter for our research.

For example, one could mine a set of retail transactions from a store to generate a set of association rules based upon the pattern of the sales. In particular, one association rule might reveal that the purchase of office chairs are often associated with the purchase of fancy staplers. Hence, the retailer could make strategic deals offering free, fancy staplers with their more expensive line of office chairs. In the context of our example, each retail transaction is a really a record in the database, and contains any number of items purchased from the store by a particular customer at a particular time. A unique set of items from the database, that may or may not be contained in a record, is known as an “itemset.”

The word often in “often associated” is important. Association rules are governed chiefly by two statistical parameters: support and confidence. Support is a measure of how frequently a given set of items appears in the database (DB). Confidence, on the other hand, is a conditional probability; that is, given a set of items, what is the likelihood another disjoint set of items will be in the same itemset.

We give an example. Let \( \mathcal{I} \) be the set of all items in the database \( \mathcal{D} \). As before, the database \( \mathcal{D} \) may be thought of as a collection records, that is, \( \mathcal{D} \) contains all records \( \mathcal{R} \) such that \( \mathcal{D} = \bigcup_{i=1}^{N} \mathcal{R}_i \) where each \( \mathcal{R}_i \subseteq \mathcal{I} \) and \( N \) is the DB size. Let us consider two itemsets \( A, B \subseteq \mathcal{I} \). The set \( A \cup B \) is a frequent itemset or
has support if counting over all records in \( D \) the probability \( P(A \cup B) = \frac{\text{count}(A \cup B)}{N} \geq \text{MinSupport} \) where the parameter \( \text{MinSupport} \in [0, 1] \) is supplied by the data miner (Han & Kamber, 2006). An association rule is written in the form \( A \Rightarrow B \) and can be read “B is likely whenever A is likely.”

Confidence of the rule \( A \Rightarrow B \) is measured by the probability \( P(B|A) = \frac{\text{count}(A \cup B)}{\text{count}(A)} \) counting over all records in \( D \) (Han & Kamber, 2006). Similarly, set \( A \cup B \) has confidence if \( P(B|A) \geq \text{MinConfidence} \) where the parameter \( \text{MinConfidence} \in [0, 1] \) is also supplied by the data miner. Therefore, it is up to the data miner and not the algorithm to decide which association rules are truly “interesting.” Clearly, rules with higher support and confidence may tend to be more interesting, but their usefulness must actually be determined by the data miner.

Association rules are useful in more areas than just marketing strategies. For example, one could mine sets of medications and their reported side-effects over a database of hospital patient records. A school system might like to see what course electives are often taken together. Furthermore, perhaps a bioinformatics specialist might like to mine automatically generated genome annotations against negative association rules to weed out annotation errors (Artamonova et al., 2007). The degree to which association rule mining (ARM) may be useful is limited only to the imagination of the data miner.

Given such diverse applications, it should not surprise the reader that the algorithms involved in data mining are also somewhat varied. Notwithstanding their uniqueness in approach, ARM algorithms each have the same job to do in order to generate association rules: count each frequent itemset. It is the organization of data and the method of the counting that affects the efficiency, space, and memory performance of the algorithm in question. Therefore, we next examine the basic characteristics of the algorithms used in ARM.

1.2 Association Rule Mining Algorithms

As we have said, the most fundamental operation for ARM algorithms is the counting of itemsets. This is commonly done by counting \( k \)-itemsets in some carefully planned sequence, where \( k \) is the number of distinct items in an itemset. As a convention, we shall denote the set of frequent \( k \)-itemsets as \( L_k \) and the set of candidate \( k \)-itemsets, that is, those with potential to be frequent, as \( C_k \). Also, let \( 1 \leq k \leq N \), where \( N \) is the number of distinct items in the database (cardinality of \( I \)).

One of the first ARM algorithms, Apriori, relies on the simple observation that if a \( k \)-itemset is non-frequent then all supersets of that set will also be non-frequent (Agrawal & Srikant, 1994). Hence, when generating a new candidate itemset in \( C_{k+1} \) to count, one need only choose itemsets from \( L_k \) that are already frequent. Upon counting the candidate itemsets one can retain the frequent ones for the next round of counting, that is, those whose support meet the minimum support threshold. Apriori occurs in distinct
phases: (1) count all frequent 1-itemsets to obtain $L_1$ (2) use $L_1$ to generate $C_2$ (3) count all 2-itemsets in $C_2$ and retain frequent ones in $L_2$ (4) repeat steps 2 and 3 until all frequent itemsets are counted, that is, obtain $L_{k+1}$ by counting $C_{k+1}$ being in turn generated from $L_k$.

One big problem with Apriori is when $L_k$ is large one must generate a large number of candidate itemsets in $C_{k+1}$. Park et al. (1997) trim down the candidate set size by using a direct hashing and pruning algorithm (DHP). DHP filters out itemsets during candidate generation using hashed information gathered from previous algorithm passes and thereby reduces the size of candidates sets. In particular, Park et al. (1997) observe that reducing the size of $L_2$ early on prevents bottlenecks in the data mining process. Additionally, DHP saves counting time by using information saved in a hash to reduce the effective size of the database.

However, even with the reduction afforded by hashing techniques it can be an expensive procedure to generate and mine candidate itemsets for frequent ones. Han et al. (2004) discuss a method for generating frequent itemsets without candidate set generation—using a frequent-pattern tree (FP-tree) data structure that only requires two scans of the database to construct. The algorithm works by adding suffix patterns recursively to a frequent prefix path itemset. One potential drawback to the FP-tree approach is that the FP-tree could get quite large such that one would have to write it to disk rather than keep it in main memory. One possible solution suggested by the authors is to partition the database into projected databases to keep the FP-tree small. This is at the cost of more complexity and the need for parallel or sequential database processing.

In order to use the power of recursion without relying on complicated data structures like the FP-tree, a simple and efficient algorithm was developed in (Borgelt, 2005) to solve this problem. This algorithm works by pre-processing and recursive elimination—using singly linked lists instead of an FP-tree and without the need to re-represent the database records. It is therefore effective in time versus Apriori and memory usage versus the FP-tree algorithm.

We have reviewed some examples of ARM algorithms and seen that there is a variety of methods used to count frequent itemsets within databases. However, what if multiple data miners would like to confer together to mine their data corporately? Distributed data mining is the answer to just such a problem.

### 1.3 Distributed Data Mining

While we are able to discover interesting new association rules by mining a single site, we may discover new rules or learn the true value of our local rules if we can mine our data with a set of peers. Distributed data mining (DDM) offers these sorts of advantages. Specifically, we look at a couple of algorithms in the
child field of distributed association rule mining (DARM).

Fast Distributed Data Mining (FDM) algorithm by Cheung et al. (1996) uses an Apriori style approach to mine data iteratively in a bottom-up fashion from multiple sites. One of the techniques used in FDM is to mine itemsets only if they are locally frequent in at least one site in the network. Indeed, an itemset cannot be globally large if it is not locally frequent in at least one site.

An algorithm loosely based off of FDM is the optimized distributed association rule mining algorithm (ODAM) developed in (Ashrafi et al., 2004). ODAM is concerned with reducing the communication cost and therefore the candidate generation of DARM algorithms. Two techniques employed by ODAM for optimization are achieved by combining redundant records in memory and by removing all globally infrequent 1-itemsets from each record, thus in turn creating a greater possibility for redundant records that may be merged. (We use the word “global” to refer to itemsets that are infrequent over all databases combined, not according to support level of each constituent.)

It should be apparent to the reader that when working with DDM algorithms efficiency concerns are paramount because the algorithm must scale with the number of different sites or nodes. Communication and coordination of information are only two very important problems to DDM. A third issue is one of privacy. While it is useful to mine data with one’s peers, it may be disadvantageous to divulge private information to those peers if they are also your competitors. Privacy-preserving data mining seeks to address just such an issue.

1.4 Privacy-preserving Data Mining

DDM offers the data miner a larger data set with the possibility of stronger and, perhaps, novel association rule findings. However, DDM comes at the cost of having to work with a diverse set of one’s peers whose honesty, friendliness, and corporate affiliation may not favor the open collaboration of one’s private data. Imagine competing schools trying to discover a set of core classes associated with high grades.

While knowing that calculus, English 10, and Spanish 101 are associated with high scholastic achievement on an aggregate level, it would be unfavorable, in particular, for Underdog Academy to admit its rampant failure at teaching any of these subjects with excellence. Hence, Underdog Academy would like to know what classes to put its small budget into without giving its competition a leg up on its weaknesses. Privacy-preserving data mining (PPDM) offers just such an opportunity.

Encryption has been a staple method in PPDM to ensure privacy, though other methods are used (Reiter & Rubin, 1998). One such method is described in (Kantarcioglu, 2004) using commutative encryption, that is, the order of encryption does not affect the value of the ciphertext. Other algorithms have used public
key cryptography, such as ElGamal, to ensure privacy (Brickell & Shmatikov, 2006). A problem with cryptography is that it is a computationally expensive addition to any algorithm or protocol and sometimes requires setting up keys to do the encryption and decryption as in public key cryptography. The number of key pairs scales with respect to the number of sites in the data mining operation.

One interesting piece of research suggests relying more on the anonymity of values to achieve a measure of security, that is, if a data miner cannot figure out which value belongs to whom, a level of privacy is therefore afforded when combined with cryptographic techniques (Ishai et al., 2006). Unfortunately, as the author points out, anonymity alone merely obscures the data values while not hiding them (giving true privacy) and so this methodology must still be combined with encryption of some sort, although anonymity may help increase efficiency concerns.

Many more PPDM algorithms exist. One good review of them is given in (Verykios et al., 2004). Additionally, PPDM algorithms closely related to our work are discussed in Chapter 3. An understanding of Chapter 2 will be useful to the reader before venturing too much further into PPDM. On the other hand, a number of useful algorithm primitives for PPDM are given in (Clifton, Kantarcioglu, Vaidya, Lin, & Zhu, 2002), including secure set union, secure size of intersection, and secure sum—the last of which we make extensive use of in our research.

1.5 Secure Sum

One very straightforward technique used in PPDM algorithms is the secure sum (SS) as described in (Clifton, Kantarcioglu, Vaidya, Lin, & Zhu, 2002). (We drop the “mod n” in our explanation of SS for simplicity, though it may be assumed by the reader if required.) The goal of SS is simple: let the value of each site’s individual input be masked while the global sum of all inputs is universally known. In other words, corporate privacy of data is preserved. (Corporate privacy is the privacy of the data in aggregate whereas individual privacy is that of individual transactions in the data set (Clifton, Kantarcioglu, & Vaidya, 2002).)

We define the “global sum” to be the sum of all count values for any particular itemset. Hence if we have $M$ sites or nodes and a count value $V_i$ for each site $S_i$, then we define the global sum as: $GlobalSum = \sum_{i=1}^{M} V_i$. We say also “global” because each site’s count value is actually a “local” sum, that is, it is the sum of the number of occurrences of an itemset in the local database.

Suppose one wished to compute the global (aggregate) support count for some arbitrary item in the database—as is done in algorithms finding association rules—such that each individual site’s support count is revealed to no other parties. Assume also a circuit network topology.
First, the initiating site, $S_1$, in the network circuit of data miners will calculate a random number $R$ and add it to its local support count, $V_1$, for the item in question. Next, $S_1$ will send $R + V_1$ to the next site, $S_2$, who will add its local support count, $V_2$, to the sum it received before sending on the final summation $R + V_1 + V_2$ to the next site in the circuit. The next site will repeat this process until the circuit closes.

When circuit does close, the initiating node, $S_1$, will receive the sum of all local supports plus the $R$ value. Since $S_1$ knows $R$, the global support can then be easily calculated by subtracting out the $R$ value.

Effectively, the $R$ value masks any site from knowing the exact value of the summation of support counts of all the previous sites in the network circuit. In particular, the $R$ value protects $S_1$ from divulging its support count to $S_2$ at the beginning of the circuit. An example of SecureSum is shown in Figure 1.1 for three participating sites.

SS is important because it allows one to transmit data securely without the high computation cost of encryption methods. However, in its current form, Secure Sum is highly vulnerable to collusion. For example, consider sites $S_1, S_2, S_3$ as above. To find $V_2$, observe that $V_2 = (sV_2 + V_1 + R) - (V_1 + R)$. Indeed, for any arbitrary $S_i$, $V_i$ can be found if $S_{i-1}$ and $S_{i+1}$ collude.

We see then that SS is a simple and important tool in PPDM. Having elucidated the basic concepts of ARM, DDM, and PPDM through the literature, we next turn to some basic graph theory and algorithms involved in generating edge-disjoint Hamiltonian cycles. This theory is fundamental to our research and contribution to PPDM: cycle-partitioned secure sum with opt-out.
CHAPTER 2

HAMILTONIAN CYCLES

A knowledge of some basic graph theory is recommended for the reader to understand the results given in Chapter 3. That notwithstanding, this section may still prove useful in clarifying the nomenclature and notation we adopt. The latter part of this chapter is devoted to the review of algorithms for generating edge-disjoint Hamiltonian cycles in a variety of network topologies.

2.1 Graph Theory: An Introduction

A graph $G$ contains a set of vertices $V$ and a set of edges $E$ such that $G = (V, E)$. An edge contains exactly two vertices, and can be denoted as an ordered pair $x = (a, b)$ where $a, b \in V$ and $x \in E$. The number of vertices in a graph is denoted $|V|$ and the number of edges is $|E|$. Notice that for a fully connected graph (where there is an edge between any two vertices) $|E| = \binom{|V|}{2}$. In networking algorithms a node may also be called a “site” or “host” and an edge a “connection” between them. A circuit of sites connected in a large loop is then analogous the concept of the “cycle”.

A cycle $C$ is defined as a sequence of edges where any two adjacent edges share at least one node and where the first and last edge share at least one node. For example, suppose we have the cycle $C_0 = (x_1, x_2, \ldots, x_N)$ where $x_i \in E$ for $1 \leq i \leq N$, then for any fixed $i$ with $1 \leq i < N$ and $x_i \neq x_{i+1}$, a node $v \in V$ such that $v \in x_i \cap x_{i+1}$. Moreover, for $x_N \neq x_1$, $\exists v \in V$ such that $v \in x_N \cap x_1$.

Traditionally, one could also view a cycle as a sequence of nodes where any two adjacent nodes are contained by an edge in the graph, and where the first and last node are the same and contained in an edge in the graph. This is an intuitive definition, but we adopt the former because of its usefulness later on.

A Hamiltonian cycle $H$ is a cycle that visits every node in the graph exactly once before returning to the source node. Alternatively, a Hamiltonian cycle can be described as a sequence of edges where any two
adjacent edges share exactly one unique node in the graph, where the first and last edge in the cycle share exactly one unique node in the graph, and where the number of edges in \( H \) equals the number of nodes in the graph.

Using the latter definition, consider \( H = (h_1, h_2, \ldots, h_N) \) where \( h_i \in E \) for \( 1 \leq i \leq N \), then for any fixed \( i \) with \( 1 \leq i < N \) with adjacent edges \( h_i, h_{i+1} \) in the cycle, there exists exactly one unique \( v \in V \) such that \( v \in h_i \cap h_{i+1} \). Next, there also exists exactly one unique \( w \in V \) such that \( w \in h_N \cap h_1 \). In the context of Hamiltonian cycles \( N = |V| \), hence we have exactly \( N \) unique, shared values. In other words, we see that \( V = \left( \bigcup_{i=1}^{N-1} h_i \cap h_{i+1} \right) \cup (h_N \cap h_1) \). In terms of network topology, a Hamiltonian cycle is analogous to a ring network where every site in the region is member of the ring.

Now we discuss what it means to have edge-disjoint Hamiltonian cycles. For edges \( e_1 = (a, b) \) and \( e_2 = (c, d) \) to be “disjoint”, with \( a \neq b \) and \( c \neq d \), it means that at least three nodes from \( a, b, c, d \) must be unique. To denote “disjoint” we say \( e_1 \neq e_2 \). We now state the definition of edge-disjoint Hamiltonian cycle:

**Definition 1.** Hamiltonian cycle \( H^{(1)} \) is edge-disjoint from different Hamiltonian cycle \( H^{(2)} \), if every edge in \( H^{(1)} \) is disjoint with every edge in \( H^{(2)} \). In other words, \( \forall e \in H^{(1)}, e \neq d, \forall d \in H^{(2)} \).

Edge-disjoint Hamiltonian cycles are useful in computer networks to provide data fault-tolerance and computer security. In terms of fault-tolerance, if a link is broken on one cycle, another edge-disjoint cycle with redundant information should still be able to relay the message since no edges are shared between cycles. In terms of computer security, one can divide up data values such that the original value can only be restored with information from each cycle. We appeal to the latter use.

### 2.2 Generating Edge-disjoint Hamiltonian Cycles

The problem of generating edge-disjoint Hamiltonian cycles is dependent upon one’s network topology constraints, or put mathematically, it depends upon the graph properties. The most relaxed view of a network is to assume full connectivity. In this case there exists a connection between each and every node.

#### 2.2.1 Fully-Connected

The number of edge-disjoint Hamiltonian cycles we can compute is dependent upon the number of nodes in the graph. For a fully-connected graph or network, this node-dependent relationship can be expressed as \( C = \text{floor}(\frac{M-1}{2}) \) where \( C \) is the number of edge-disjoint Hamiltonian cycle and \( M \) is the number of nodes. We include this as a theorem:
**Theorem 1.** Let $M \geq 3$ be the number nodes in a fully connected graph $G$, let $k \in \mathbb{N}$. If the number of nodes $M = 2k + 1$ is odd, there exists $k$ edge-disjoint Hamiltonian cycles in $G$. Otherwise, $M = 2k$ is even and there exists $k - 1$ edge-disjoint Hamiltonian cycles.

Hence, for 3 and 4 nodes we have 1 cycle, for 5 and 6 nodes we have 2 cycles, and so on. Thus the number of cycles scales linearly with respect to the number of nodes.

Now to generate a set of edge-disjoint Hamiltonian cycles for a fully connected graph we may use the mathematical cycle permutation $\sigma$ as found in (Alspach et al., 1990) where $\sigma(H) = (N_1)(N_2N_3N_4 \ldots N_{M-1}N_M)(N_1)$. Note that $\sigma(H)^k$ means to apply the permutation $\sigma$ to the initial Hamiltonian cycle $H$ iteratively $k$ times where $k \in \mathbb{N}$. The Hamiltonian cycle generation then occurs according to the following pattern:

$$H^{(k)} = \begin{cases} N_1N_2N_3N_4N_{M-1} \cdots N_1 & \text{if } k = 1 \\ \sigma(H^{(1)})^{k-1} & \text{if } 1 < k \in \mathbb{N} \end{cases}$$

We have derived an algorithm for this permutation and it is listed in Algorithm 1.

---

**Algorithm 1** Edge-disjoint Hamiltonian Cycle Generation

**Require:** Given $M$ nodes with $M \geq 3$ and a fully connected network.

**Ensure:** Generate $C$ edge-disjoint hamiltonian cycles, where $C = \lfloor \frac{M-1}{2} \rfloor$.

1: $C \leftarrow \text{floor}(\frac{M-1}{2})$
   \{Compute cycle $H^{(1)}$\}
2: $H^{(1)}_1 \leftarrow 1$ \{Cycle $H^{(1)}$ begins with node $N_1$.\}
3: $H^{(1)}_2 \leftarrow 2$
4: $H^{(1)}_3 \leftarrow 3$
5: for $j = 4$ to $M$ do
6:   if even($j$) then
7:      $H^{(1)}_j \leftarrow M - (\frac{j}{2} - 2)$
8:   else
9:      $H^{(1)}_j \leftarrow \text{floor}(\frac{j}{2}) + 2$
10: end if
11: end for
12: $H^{(1)}_{M+1} \leftarrow 1$ \{Close the cycle by returning to node $N_1$.\}
   \{Generate subsequent cycles using the permutation: $(N_1)(N_2N_3N_4 \ldots N_{M-1}N_M)(N_1)$.\}
13: for $i = 2$ to $C$ do
14:   $H^{(i)}_1 \leftarrow 1$ \{Cycle $H^{(i)}$ begins with node $N_1$.\}
15: for $j = 2$ to $M$ do
16:   if $H^{(i-1)}_j \neq M$ then
17:      $H^{(i)}_j \leftarrow H^{(i-1)}_j + 1$
18:   else
19:      $H^{(i)}_j \leftarrow 2$
20: end if
21: end for
22: $H^{(i)}_{M+1} \leftarrow 1$
23: end for
As an example of finding edge-disjoint Hamiltonian cycles, suppose we have a fully connected graph with \( M = 5 \) nodes. Therefore, according to Theorem 1 there must exist 2 edge-disjoint Hamiltonian cycles \( H^{(1)} \) and \( H^{(2)} \) since \( \text{floor}\left(\frac{5-1}{2}\right) = 2 \). The output for Algorithm 1 with \( M = 5 \) nodes would be \( H^{(1)} = (1, 2, 3, 5, 4, 1) \) and \( H^{(2)} = (1, 3, 4, 2, 5, 1) \). A diagram of this example is listed in Figure 2.1. Reducing a graph to its edge-disjoint Hamiltonian cycles may be referred to as its Hamiltonian decomposition or, in our language, its cycle-partitioning. A table of examples is listed in Table 2.1.

<table>
<thead>
<tr>
<th>Table 2.1: Example Cycle Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M )</td>
</tr>
<tr>
<td>( H^{(1)} )</td>
</tr>
<tr>
<td>( H^{(2)} )</td>
</tr>
<tr>
<td>( H^{(3)} )</td>
</tr>
</tbody>
</table>

In Figure 2.1, “cycle 1” (dashed) corresponds to \( H^{(1)} \) and “cycle 2” (dotted) corresponds to \( H^{(2)} \) from our previous example. Our source node or site is “N1” and each other site in the network must be visited before either cycle returns to the source node. The edges are diagramed as directed edges although one could traverse the cycle in both directions. However, for use in later algorithms, we consider only the digraph version.

Figure 2.1: Edge-disjoint Hamiltonian Decomposition of a Fully-Connected Graph, \( M = 5 \)
This approach is mathematically sound. Indeed, it has been proven in (Alspach et al., 1990) that there exists a Hamiltonian decomposition for complete (fully-connected) graphs of any size $M$.

### 2.2.2 Edge-disjoint Hamiltonian Cycles in Torus

A fully-connected network is not always available or preferred in some situations. For example, some useful torus network architectures are cited in (Bae & Bose, 2003): Cray T3D, the Mosaic, iWarp, and Tera Parallel Computer. On the other hand, our network might contain a number of missing links between nodes such that we lose full connectivity. In that case, we may be able to use the remaining edges to establish some known architecture where Hamiltonian cycles can be found. Indeed, there exist distributed algorithms for finding Hamiltonian cycles in $k$-ary $n$-cubes and hypercubes in polynomial time (Stewart, 2006).

We are interested primarily though in producing a set of edge-disjoint Hamiltonian cycles. Some methods for finding edge-disjoint Hamiltonian cycles in Torus and $k$-ary $n$-cubes using Gray Codes are given in (Bae & Bose, 2000) and a good example for finding cycles in a binary cube is found in (Bae & Bose, 2003).

So there is no confusion for the reader, a torus network is a wrap-around mesh, or, rather, a graph of degree $2n$ arranged in a grid where every node joins $2n$ neighboring nodes (unless on the edge, in which case the connection wraps around). A $k$-ary $n$-cube is a subgraph of torus where $n$ dimensions of the torus are each $k$ in size. A further subgraph of that is the binary cube where $k = 2$. A hypercube is an $n$-dimensional version of the binary cube.

Bae and Bose (2000) map mixed radix notation positions of the nodes to Gray codes in order to generate a Hamiltonian cycle. A simple permutation of their mapping function may produce a new independent Gray code generating another Hamiltonian cycle. Furthermore, they prove that the independent set of Gray codes over $Z_k^n$ is equivalent to the set of edge-disjoint Hamiltonian cycles in the graph of a $k$-ary $n$-cube (Bae & Bose, 2003).

A Gray code is a sequence of numerical values or words where each succeeding value or word is a distance of one from the previous value. (The distance measure used by Bae and Bose is Lee distance.) Two Gray codes are independent if any two adjacent words in one Gray code are not adjacent in another different Gray code. For example, let $G_1, G_2$ be Gray codes. Suppose 001100, 000100 are adjacent in $G_1$. Then they are not adjacent in $G_2$.

For simplicity, we reproduce **Method 4** from (Bae & Bose, 2000) as one example of how one can generate edge-disjoint Hamiltonian cycles in a non-fully-connected network. The example uses a $3 \times 5$ dimension torus network. The algorithm is reproduced in Method 4; moreover, figure 3a from (Bae & Bose, 2000) is reproduced below in Figure 2.2.

Method 4 works on the assumption that dimension sizes $k_i$ for $0 \leq i \leq n - 1$ are either all odd or
all even and that dimension sizes are ordered thusly: \( k_{n-1} \geq k_{n-2} \geq \cdots \geq k_0 \). Our example torus \( T_K = (T_5, T_3) \) is indeed all odd (dimensions must be arranged in descending order here). The algorithm is outlined mathematically below.

**Method 4. From** (Bae & Bose, 2000).

\[
\tilde{r}_i = \begin{cases} 
   r_i, & \text{if } r_{i+1} \text{ is odd} \\
   k_i - 1 - r_i, & \text{otherwise}
\end{cases}
\tag{2.1}
\]

Let the gray code word \( g_{n-1} = r_{n-1} \). Next, for \( 0 \leq i \leq n - 2 \):

\[
g_i = \begin{cases} 
   (r_i - r_{i+1}) \mod k_i, & \text{if } r_{i+1} \text{ is odd} \\
   \tilde{r}_i, & \text{otherwise}
\end{cases}
\tag{2.2}
\]

For example, starting at the top left node we have \( R_1 = 00 \) for the mixed radix notation. Observe that \( K = (k_1, k_0) = (5, 0) \). So our first gray code word \( G_1 = g_{n-1}g_0 \) can be computed by \( g_{n-1} = r_{n-1} = 0 \) and \( g_0 = (r_i - r_{i+1}) \mod k_i = (0 - 0) \mod 3 = 0 \) since \( 0 = r_1 < k_0 = 3 \). Thus, \( R_1 \to G_1 \) maps \( 00 \to 00 \). For a complete example, Table 2.2.2 finishes the computation with one intermediate step to calculate \( g_0 \) since we already have \( g_1 = r_1 \) for every node in this example. Summing up, the full sequence of nodes is \((00, 01, 02, 12, 10, 11, 21, 22, 20, 30, 31, 32, 42, 41, 40)\), where each element of the sequence is the ID of a node in mixed radix notation.

The other edge-disjoint Hamiltonian cycle can be generated by using all remaining edges in the torus. The reader will notice that generating edge-disjoint Hamiltonian cycles using this method is a trivial computation. Other methods for finding edge-disjoint Hamiltonian cycles given in (Bae & Bose, 2003) are of similar complexity for varying graph constraints.

In this chapter, we have described fully-connected networks and torus networks, given some example
Table 2.2: Method 4—Example Cycle Generation

<table>
<thead>
<tr>
<th>Radix: ( r_1 r_0 )</th>
<th>Computation for ( g_0 )</th>
<th>Gray Code: ( g_1 g_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>((0 - 0) \mod 3 = 0)</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>((1 - 0) \mod 3 = 1)</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>((2 - 0) \mod 3 = 2)</td>
<td>02</td>
</tr>
<tr>
<td>10</td>
<td>((0 - 1) \mod 3 = 2)</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>((1 - 1) \mod 3 = 0)</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>((2 - 1) \mod 3 = 1)</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>((0 - 2) \mod 3 = 1)</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>((1 - 2) \mod 3 = 2)</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>((2 - 2) \mod 3 = 0)</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>(3 - 1 - 0)</td>
<td>42</td>
</tr>
<tr>
<td>41</td>
<td>(3 - 1 - 1)</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>(3 - 1 - 2)</td>
<td>40</td>
</tr>
</tbody>
</table>

methods for finding edge-disjoint Hamiltonian cycles from the literature, and reviewed some basic graph theory and notation for the reader. Having gained some confidence in being able to generate cycles, we next apply the theory of edge-disjoint Hamiltonian cycles to privacy-preserving data mining.
We have seen in Section 1.5 that Secure Sum (SS) is indeed vulnerable to collusion. Recall that collusion resistance, in terms of SS, means keeping the values generated at any individual node secret, such that if other nodes share information together or collude, the secret values cannot be determined.

Consider a simple analogy for collusion resistance. Bob, Eve, Ray, and Alice each contribute anonymously to their mother’s birthday party. Their father thanked them each for the $35 it cost all of them to put on the party.

Eve is jealous of Alice and would like to know if she gave more to the party than her sister, so she convinces Ray to collude with her and to tell her how much he gave. Ray agrees. Eve gave $15 and Ray gave $10, so Eve knows Bob and Alice gave $10 = $35 − ($15 + $10). Therefore she correctly concludes she gave more than Alice.

However, Eve would also like to rub it in, so she asks Bob to tell her how much he gave too. Bob refuses, leaving Eve without any method to compute Alice’s exact contribution to the party. So we learn from this story that the key to collusion resistance involves increasing the number of people (or nodes) required to compute the private data values of the remaining persons or nodes.

This intuition is indeed present in the literature. Clifton and Vaidya (2004) lay the foundation for a cycle-partitioned secure sum algorithm by noting how any node in a secure sum can divide its value into random “shares” (we use the word “partition”) such that each share is passed along a different secure sum circuit or cycle. Collusion protection is then achieved when no node has the same neighboring nodes twice for each cycle.

This observation is carried out in a collusion resistant PPDM algorithm implicitly based on two edge-disjoint hamiltonian cycles (Wang et al., 2006). Here each node divides its count value into two random partitions and performs a SS operation for each partition—making sure its neighbors in the algorithm are
all different.

The work from (Wang et al., 2006) was then extended in (Urabe et al., 2004) to work with $D$ “routes” where each route is equivalent to edge-disjoint Hamiltonian cycles—meaning each count value is divided up into $D$ random partitions at each site and passed along the SS cycle.

Another unique method to providing collusion resistance by creating shares but without the need for edge-disjoint Hamiltonian cycles was given in (Urabe et al., 2007). This last method also relies on SS and on the need for a fully-connected network but sends data to nodes in a tree-like pattern for each round of the algorithm.

We implement our own version of a cycle-partitioned secure sum algorithm, similar to (Urabe et al., 2004) but provide a mathematical proof for its collusion resistance based on edge-disjoint Hamiltonian cycles. We then extend this algorithm to include an “opt-out” feature to address some additional security concerns. Finally, it should be noted that the collusion resistance we achieve is similar to the resistance achieved by the algorithms in (Urabe et al., 2004) and (Urabe et al., 2007).

### 3.1 Cycle-partitioned Secure Sum

We have discussed in Chapter 2 the generation of edge-disjoint Hamiltonian cycles. The reader will recall that each edge-disjoint Hamiltonian cycle is a unique ordering of the sites or nodes such that no node has the same adjacent nodes in any two different Hamiltonian cycles. As we will see, this property is very useful for collusion resistance.

A cycle-partitioned secure sum (CPSS) algorithm is similar to SS in that values are added sequentially along a cycle that starts and ends with an initiating site or node, $N_1$. At the beginning of the cycle, $N_1$ adds a random number to its count value before sending it to the next node in the cycle. $N_1$ must also subtract that random number at the end of the cycle to obtain the global sum (the sum of all count values for any particular itemset).

CPSS differs from secure sum in that instead of just using one cycle for summing, the nodes are arranged into $C$ different edge-disjoint Hamiltonian cycles. At each node, the count value is then divided into $C$ random, non-zero, integer partitions—one partition for each edge-disjoint Hamiltonian cycle. Each partition is then sent to the next node in the appropriate cycle.

In the same fashion, the initiating node adds its count value to a random number $R$ and then randomly partitions this value to send along each cycle. When $N_1$ receives the subtotals from each circuit or cycle at the end of the computation, the subtotals are added together and the random number $R$ is subtracted out to obtain the global sum. The pseudocode for CPSS is given in Algorithm 2.
Algorithm 2 CPSS: Cycle-Partitioned Secure Sum

Require: Given \( M \) nodes and \( C \) edge-disjoint Hamiltonian cycles.

Ensure: \( \sum_{k=1}^{M} V_k = \text{GlobalSum} \)

For \( N_1 \), the initiating node.

1: \( R \leftarrow \text{NewRandomNumber}() \)
2: \( \text{RandomlyPartition}(V_1 + R, C) \{ \sum_{i=1}^{C} V_{1}^{(i)} = V_1 + R \neq V_{1}^{(i)} \in \mathbb{Z} \} \)
3: for \( i = 1 \) to \( C \) do
4: Send \( V_{1}^{(i)} \) to the next node in cycle \( i \).
5: end for

For each node, partition & add to subtotals in each cycle.

6: for \( k = 2 \) to \( M \) do
7: \( \text{RandomlyPartition}(V_k, C) \{ \sum_{i=1}^{C} V_{k}^{(i)} = V_k \neq V_{k}^{(i)} \in \mathbb{Z} \} \)
8: for \( i = 1 \) to \( C \) do
9: \( V_{k}^{(i)} \leftarrow V_{k}^{(i)} + V_{\text{received}}^{(i)} \)
10: Send \( V_{k}^{(i)} \) to the next node in cycle \( i \).
11: end for
12: end for

\( N_1 \) receives \( C \) values.

13: \( \text{GlobalSum} = \sum_{i=1}^{C} V_{\text{received}}^{(i)} - R \)
14: Broadcast \( \text{GlobalSum} \) to each other node.

We also define the function \( \text{RandomlyPartition} \) in Algorithm 3, which is needed by Algorithm 2. Suppose, for example, that we have \( C = 5 \) cycles and wish to randomly partition the count value of \( A = 7 \) at some fixed node to send along each cycle. First we generate 4 random numbers: \( 16, -2, 3, -15 \). We also call these random numbers “random partitions.” Next, we see that \( 7 - (16 + -2 + 3 + -15) = 7 - 2 = 5 \), so the fifth random partition is 5. Finally, we may observe that our five random numbers \( (16, -2, 3, -15, 5) \) do in fact partition our count value:

\[ 16 + -2 + 3 + -15 + 5 = 7 = A. \]

Algorithm 3 RandomlyPartition(A, C)

Require: Given integer value \( A \).

Ensure: \( \sum_{i=1}^{C} A_i = A \).

1: \( \text{Sum} \leftarrow 0 \)
2: for \( i = 1 \) to \( C - 1 \) do
3: \( A_i \leftarrow \text{NewRandomNumber}() \{ 0 \neq \text{NewRandomNumber}() \in \mathbb{Z} \} \)
4: \( \text{Sum} \leftarrow \text{Sum} + A_i \)
5: end for
6: \( A_C \leftarrow A - \text{Sum} \)

We see that CPSS differs from SS in terms of its privacy-preserving features. While SS protects the subtotals, and, indeed, the count values from being known by any particular node, it is weak to the collusion
of any two nodes.

For example, if nodes $A, B, C$ are in a SS cycle in the order ($\ldots, A, B, C, \ldots$), then the value $C$ receives from $B$ less the value $A$ sends to $B$ is the count value of $B$. That is, if the count values of $A, B, C$ are 5, 7, 4 respectively and $T$ is the subtotal so far, then $A$ sends $5 + T$ to $B$ and $C$ receives $5 + 7 + T$ from $B$. When $A$ and $C$ collude, we may compute the count value of $B$ by doing a simple subtraction: $7 = (5 + 7 + T) - (5 + T)$.

It is clear that $B$ could easily tell any other site its count value. This contingency is trivial and we assume all non-colluding nodes prefer to keep their count values private. We make this assumption for CPSS as well. Figure 3.1 shows an example running of CPSS, Algorithm 2, for $M = 5$ nodes. Notice that for each node in this example there are a total of four adjacent nodes: two receiving and sending.

![Figure 3.1: Cycle-partitioned Secure Sum, $M = 5$](image)

Suppose we wished to compute the count value ($V_2$) of node 2 from Figure 3.1 by collusion. In order to compute $V_2 = 7$, observe that it requires the two incoming nodes and the two receiving nodes to collude. That is, it requires $N_3$ and $N_5$ on the receiving end and $N_1$ and $N_4$ on the sending end respectively: $7 = (5 + 13) - (2 + 9)$.

Thus, we observe from this example that 4 nodes were required to compute the remaining node’s count value. It is clear then that we need a good definition to describe what it exactly means to be collusion resistant. We adopt the following formal definition to describe collusion resistance for CPSS.
**Definition 1.** CPSS is $K$-collusion resistant if no $K$ nodes or fewer working together in a set can compute the count value of any node not belonging to the $K$-node set.

Using the above definition, we will show that CPSS confers $K$-collusion resistance proportional to the number of edge-disjoint Hamiltonian cycles in the network.

**Theorem 1.** Given any network with $C$ edge-disjoint Hamiltonian cycles and $M$ nodes such that $M > 4$, CPSS is $K$-collusion resistant with $K = 2C - 1$.

**Proof.** Given a graph with $M \geq 5$ nodes decomposed into $C \geq 2$ edge-disjoint Hamiltonian cycles, and given any fixed node $N_k$ with value $V_k$, where $1 \leq k \leq M$, show that no set of $2C - 1$ nodes can compute $V_k$ when $N_k$ is not in the set. In other words, a minimum of $2C$ nodes different from $N_k$ are required to compute $V_k$.

We denote our edge-disjoint Hamiltonian cycles as $H(i)$ with $1 \leq i \leq C$. For any fixed $i$, let $(E_p^i)_{p=1}^M$ be the sequence of edges for that cycle, and let $(H_q^i)_{q=1}^M$ be the sequence of nodes in that cycle $i$. The random number generated by $N_1$ we designate as $R$ with random partitions $R(i)$ such that $R = \sum_{i=1}^C R(i)$.

Recall that for any fixed $k$, $N_k$ partitions its value $V_k$ into $C$ non-zero, integers to send on each cycle $H(i)$, such that $\sum_{i=1}^C V_k^i = V_k$. In order to define the computability of these values, we use the weight function $w$. The weight function for a given edge is equal to the running total of values on that cycle as computed by the source node of the specified edge:

$$w(E_j^i) = R(i) + \sum_{p=1}^j V_h^i, \text{ where each } h_p = H_p^i. \quad (3.1)$$

The substitution of $h_p = H_p^i$ is given for simplicity. Furthermore, notice that $\sum_{p=1}^M V_h^i = \sum_{p=1}^M V_p^i$ for any fixed $i$ and that $w(E_1^i) = V_1 + R(i)$ for all $i$. Using (3.1), we can compute $V_k^i$ for any fixed $k$ such that $2 \leq k \leq M$ (the computation for $V_1$ is to follow):

$$V_k^i = w(E_z^i) - w(E_{z-1}) \quad 2 \leq z \leq M \quad (3.2)$$

$$= R(i) + \sum_{p=1}^z V_h^i - (R(i) + \sum_{p=1}^{z-1} V_h^i) \quad (3.3)$$

$$= V_k^i + \sum_{p=1}^{z-1} V_h^i - \sum_{p=1}^{z-1} V_h^i = V_k^i \quad V_k^i = V_h^i \quad (3.4)$$
In general, for any two consecutive edges in a Hamiltonian cycle, say $D_1$ and $D_2$, we have $D_1 = \{a,b\}$ and $D_2 = \{b,c\}$ such that $a, b, c$ are each unique nodes in the cycle. Suppose then for any fixed $i$ and fixed $k \geq 2$ that $N_k = E_k^{(i)} \cap E_{k-1}^{(i)}$ (two consecutive edges in cycle $i$), then the Hamiltonicity of our cycle implies that there exist two unique nodes different from $N_k$ in $E_k^{(i)} \cup E_{k-1}^{(i)}$. These nodes may collude together to compute $V_k^{(i)}$ as shown in (3.2)-(3.4). However, to compute $V_k$ for any fixed $k \geq 2$ we need:

$$V_k = \sum_{i=1}^{C} V_k^{(i)}$$

(3.5)

So to compute $V_k$ for any fixed $k \geq 2$ it requires $C$ partitions of $V_k$, each partition computable by exactly two unique nodes other than $N_k$. Since our choice of $H^{(i)}$ was arbitrary and any two cycles are edge-disjoint, we see that the two nodes computing each $V_k^{(i)}$ are each distinct. To see this is true, pick a node not $N_k$ to be adjacent to $N_k$ in two different cycles, then we have the same edge in two cycles, violating our edge-disjointedness. Hence, for any arbitrary $k \geq 2$, $2C$ unique nodes other than $N_k$ are required to compute $V_k$. Moreover, since each $V_k^{(i)}$ was non-zero, not less than $2C$ nodes are needed, and therefore, $2C - 1$ nodes, of which $N_k$ is not a part, cannot compute $V_k$.

Next, for $k = 1$, we have a similar situation, only that the random number $R$ is first added to $V_1$ before it is partitioned and sent to the next node in each cycle. So to find $R$, we rely on the fact that each node is sent the GlobalSum at the end of the computation:

$$R = \sum_{i=1}^{C} w(E_{M}^{(i)}) - \text{GlobalSum}$$

(3.6)

$$= \sum_{i=1}^{C} (R^{(i)} + \sum_{p=1}^{M} V_{h_p}^{(i)}) - \text{GlobalSum} \quad \text{(from 3.1)}$$

(3.7)

$$= \sum_{i=1}^{C} R^{(i)} + \sum_{i=1}^{C} \sum_{p=1}^{M} V_{p}^{(i)} - \sum_{k=1}^{M} V_k$$

(3.8)

$$= R + \sum_{p=1}^{M} \sum_{i=1}^{C} V_{p}^{(i)} - \sum_{k=1}^{M} V_k$$

(3.9)

$$= R + \sum_{p=1}^{M} V_p - \sum_{k=1}^{M} V_k = R$$

(3.10)
The edge $E_{i}^{(i)}$, for any fixed $i$ with $1 \leq i \leq C$, contains node $N_1$ and another, different node (by the definition of a Hamiltonian cycle). Since each cycle is edge-disjoint and contains $N_1$, then (3.6) implies that $C$ unique nodes other than $N_1$ are needed to compute $R$. Now we use knowledge of $R$ to compute $V_1$:

\[
V_1 = \sum_{i=1}^{C} w(E_{1}^{(i)}) - R \\
= \sum_{i=1}^{C} (R^{(i)} + \sum_{p=1}^{1} V_{p}^{(i)}) - R \\
= \sum_{i=1}^{C} R^{(i)} + \sum_{i=1}^{C} V_{1}^{(i)} - R \\
= R + V_1 - R = V_1
\]

(3.11) (3.12) (3.13) (3.14)

It is clear that $E_{1}^{(i)}$, for any fixed $i$ with $1 \leq i \leq C$, contains node $N_1$ and a different node (by the definition of a Hamiltonian cycle). Because cycles are edge-disjoint and each edge contains $N_1$, then the nodes not $N_1$ are each unique from $N_1$ and each other. From (3.11) we see that there are $C$ such nodes other than $N_1$ needed to finish the computation of $V_1$.

To show that the set of $C$ nodes different from $N_1$ found in (3.6) and the other set of $C$ nodes different from $N_1$ found in (3.11) are all distinct, notice that $N_1 = E_{1}^{(i)} \cap E_{M}^{(i)}$ and that the definition of Hamiltonian cycle implies that, for any fixed $i$ with $1 \leq i \leq C$, there exist two unique nodes not $N_1$ in $E_{1}^{(i)} \cup E_{M}^{(i)}$. Since cycles are edge-disjoint and because the first and last edge of every cycle each contains $N_1$, every node not $N_1$ must be unique. Hence a total of $2C$ nodes are needed to compute $V_1$. Since each $V_{1}^{(i)}$ is non-zero, not less than $2C$ nodes are required for this computation, that is, $2C - 1$ nodes not $N_1$ cannot compute $V_1$.

Thus, we have shown that for a node $N_k$ with value $V_k$, for any fixed $k$ with $1 \leq k \leq M$, a minimum of $2C$ nodes not $N_k$ are required to compute $V_k$, that is, $2C - 1$ nodes not $N_k$ cannot compute $V_k$. So by the arbitrariness of $k$ and the definition of $K$-collusion resistance, CPSS has a collusion resistance of $K = 2C - 1$. □
3.2 Problems with CPSS: Statistical Disclosure

We have shown that algorithms like CPSS can reliably protect users against $K$-colluding parties or fewer in an exact sense, but there remain scenarios where disclosure of private, sensitive data may still be leaked to other parties even when $K$ or fewer parties collude. One such scenario depends on the availability of publicly accessible information, that is, knowledge exogenous to the data mining operation itself.

Consider a distributed data mining operation using CPSS that consists of 3 national retailers $A, B, C$ and a half dozen small ones, say $D, \ldots, I$. Since $m = 9$ we may find $c = 4$ edge-disjoint Hamiltonian cycles if we assume a fully-connected network. By Theorem 1, we have a collusion resistance of 7 retailers, meaning 7 or fewer retailers cannot compute any remaining retailer’s sales count for a particular item. Said another way, 8 or more retailers are needed to compute the sales count of any remaining retailer.

However, suppose for some particular itemset $U = \{x, y, z\}$, only retailers $A, B, C$ have the rights or resources to sell the items in $U$ for any significant profit. Although the itemset $U$ has support globally because it meets the global support threshold, the other retailers $D, \ldots, I$ lack local support for $U$, and may have count values for those items close to or equal to 0.

With the foreknowledge of a big retailer monopoly at selling $U$, $A$ and $B$ may collude together to approximate or, if all other retailers cannot sell the product at all, find exactly what retailer $C$ sold for itemset $U$. So in this example, $K$ or fewer retailers not retailer $C$ colluded together and did compute the count value of retailer $C$.

Observe that whenever a small group of nodes have too great a contribution to the sum, or rather, if too many nodes contribute too little to the sum, the possibility to disclose private information exists even if $K$ or fewer nodes collude.

This problem is under the subject of statistical disclosure control. A very good book on techniques used in statistics to prevent “re-identification” and information leakage is written by Willenborg and de Waal (2000). We adapt their definition of the $(n, k)$-dominance rule from an earlier work of theirs to privacy-preserving data mining:

**Definition 2.** $(n, k)$-dominance rule—for privacy preserving data mining. An itemset is regarded as sensitive if the sum of the largest $n$ contributions from $n$ different nodes account for more than $k\%$ of the global sum (Willenborg & Waal, 1996).

Using our previous example, suppose retailers $A, B, C$ make up more than 95% of the support for some itemset $U$, then itemset $U$ is sensitive under the $(n, k)$-dominance rule with $n = 3$ and $k = 95$. Said another way, 3 different retailers out of the 9 had a cumulative sum making up 95% of the global sum for itemset $U$. 
Now let us examine the properties of sensitive itemsets.

Let $U$ be any itemset and let $U'$ be any superset containing $U$. (For example, suppose $U = \{x, y, z\}$ where $x, y, z$ are items, then we might choose $U' = \{w, x, y, z\}$, where $w$ is an item.) It is clear that for any fixed site in the network, the count value of an itemset is greater than or equal to the count value of any of its supersets, that is, $\text{count}(U) \geq \text{count}(U')$, for any arbitrary $U'$. Consider a simple analogy as an example of this property: if Fancy Grill sells 100 hamburgers and it sells 250 fries, it cannot sell more than 100 hamburgers and fries together.

Next, recall that if an itemset is $(n, k)$-sensitive, it means that $n$ different sites have $k$ percent of the global sum. If a frequent itemset is sensitive, the superset containing that itemset may also be sensitive, especially if $k$ is large. This is because the count value of an itemset has nowhere to go but down when one goes to count its superset. Indeed, at each fixed site, the small count values may go to zero while the larger count values, the dominant ones, may simply be reduced—retaining the proportion of inequity that defines a sensitive itemset. Hence, if a node learns an itemset it has is sensitive, it may desire to prevent the computation of any subsequent supersets.

This begs the question, how does one deal with sensitive itemsets? Karr et al. (2004) suggest the possibility of using an anonymous “opt-out” method to prevent statistical disclosure for their secure sum-based distributed secure regression. It is our opinion that this is indeed the simplest way to deal with the problem of disclosure without perturbing data values. We develop an anonymous opt-out method for integration with CPSS in order to deal with statistical disclosure within the context of privacy-preserving data mining.

### 3.3 Anonymous Opt-Out

Anonymous opt-out allows any node to withdraw its contribution from the CPSS computation without any other node knowing which nodes, if any, have withdrawn. We begin our results by introducing the terms $K$-collusion resistant for anonymous opt-out and non-viable global sum and finish by describing some algorithms that meet these criteria. Our anonymous opt-out algorithm is integrated with CPSS and therefore any notation related to CPSS is also assumed. Now to the definitions.

**Definition 3.** A global sum is non-viable if for $M$ nodes with values $V_k$, $\text{GlobalSum} \neq \sum_{k=1}^{M} V_k$.

A non-viable global sum simply means our cycle-partition secure operation operation failed to produce the correct output. This can be done accidentally or intentionally. With the proper code and non-faulty network links, we may avoid the former. For the latter, creating a non-viable global sum is important to opting out
algorithms because a global sum that is garbage cannot be useful to other nodes—effectively canceling out the sum operation. However, this is not a sufficient definition upon which to build an anonymous opt-out algorithm. Let us therefore define $K$-collusion resistance for anonymous opt-out.

**Definition 4.** An algorithm is “$K$-collusion resistant for anonymous opt-out” if (a) the global sum is non-viable upon opt-out, (b) any set of $K$ or fewer nodes cannot calculate the value of any node not in the $K$-node set, and (c) any set of $K$ nodes or fewer cannot determine if some other node not in the $K$-node set has opted out.

To satisfy Definition 4 we first develop a rudimentary algorithm meeting each criterion of the definition. A naïve approach then to integrate anonymous opt-out into CPSS is to simply have the withdrawing node add an extremely large random value to its count value upon opt-out (we refer to this value as the “opt-out value”). When the initiating node calculates the global sum it will notice that the global sum is above some pre-set “opt-out threshold” and will declare this particular itemset invalid to all other nodes. The psuedocode for the naïve approach is given in Algorithm 4:
Algorithm 4 CPSS with Naïve Opt-Out

Require: Given $M$ nodes and $C$ edge-disjoint Hamiltonian cycles.

Ensure: $\sum_{k=1}^{M} V_k = \text{GlobalSum}$ OR opt-out: $\sum_{k=1}^{M} V_k \neq \text{GlobalSum}$

{For $N_1$, the initiating node.}

1: $R \Leftarrow \text{NewRandomNumber}()$
2: if $\text{OptOut}(N_1) = \text{true}$ then
3: $V_1 \Leftarrow R$
4: else
5: $V_1 \Leftarrow V_1 + R$
6: end if
7: $\text{RandomlyPartition}(V_1, C) \{\sum_{i=1}^{C} V_1^{(i)} = V_1, 0 \neq V_1^{(i)} \in \mathbb{Z}\}$
8: for $i = 1$ to $C$ do
9: Send $V_1^{(i)}$ to the next node in cycle $i$.
10: end for
{For each node, partition & add to subtotals in each cycle.}
11: for $k = 2$ to $M$ do
12: if $\text{OptOut}(N_k) = \text{true}$ then
13: $V_k \Leftarrow \text{LargeRandomNumber}()$
14: end if
15: $\text{RandomlyPartition}(V_k, C) \{\sum_{i=1}^{C} V_k^{(i)} = V_k, 0 \neq V_k^{(i)} \in \mathbb{Z}\}$
16: for $i = 1$ to $C$ do
17: $V_k^{(i)} \Leftarrow V_k^{(i)} + V_{\text{received}}^{(i)}$
18: Send $V_k^{(i)}$ to the next node in cycle $i$.
19: end for
20: end for
{N_1 receives $C$ values.}
21: $\text{GlobalSum} = \sum_{i=1}^{C} V_{\text{received}}^{(i)} - R$
22: if $\text{GlobalSum} \geq \text{Threshold}$ OR $N_1$ Opted Out then
23: Broadcast $\text{SumCancelled}$ to each other node.
24: else
25: Broadcast $\text{GlobalSum}$ to each other node.
26: end if

An example of Algorithm 4 is given in Figure 3.2. Here node $N_3$ replaces its count value with a random value (1000) denoted “Opt-out”, divides that value into two partitions, and sends them along the edge-distjoint Hamiltonian cycles (we chose the random number to be 1000, the same as our opt-out threshold, for simplicity). When the initiating node receives and adds all the subtotals at the end of the computation, it notices that the global sum is greater than its opt-out threshold (also 1000). Hence, the global sum is invalidated for this itemset and $N_1$ broadcasts an opt-out message to all other nodes.

Now let us examine how Algorithm 4 satisfies Definition 4. First, since our naïve approach has not
modified the CPSS algorithm except for the minor addition of an opt-out value and threshold, we can use Theorem 1 to see that \( K \) or fewer nodes cannot deduce the count value of any node not in the \( K \)-node set, meeting the requirements of part (b).

Furthermore, it follows that if \( K \)-nodes or fewer cannot deduce any remaining nodes count value, they also cannot determine which node, if any, added a *large* opt-out value to the sum. This satisfies part (c) of the definition. Finally, since the large value added was a random and not a count value, the global sum will be non-viable, satisfying part (a) and completing the requirements of Definition 4.

However, while Algorithm 4 is both simple and can effectively satisfy Definition 4 (most of the time), there are scenarios where it fails due to its dependency on an arbitrary threshold value. For example, if the threshold value is too high, then one can approximate what the sum of all remaining node should be, violating part (a) of our definition. This can be done by simply subtracting out the most significant digits of the global sum and thereby eliminating the opt-out value. Opting out should trash the global sum, but in our example, see that we may simply subtract out significant digits to approximate the real value of the global sum: \( 1041 - 1000 = 41 \approx 42 = \text{GlobalSum} \).

On the other hand, the opposite problem may occur: if the threshold is too low, and in rare cases in which it is still high, a global sum may be mistaken as non-viable and thrown out. For instance, if we had chosen 843 instead of 1000 as our opt-out value in Figure 3.2 (still a large random number with respect to our count...
values), then our non-viable global sum would have been: $884 = (5 + 7 + 843 + 20 + 9) \leq 1000 = \text{Threshold}$. Similarly, if we choose a large, negative random number as our opt-out value, then if our global sum is large we may again undershoot the opt-out threshold and end up with a non-viable global sum being accepted as true.

Finally, if we choose the opt-out values to be both positive or negative, then we have an additional possibility of two withdrawing nodes canceling out each other's opt-out values. So we see then that in general, this method has a non-zero probability of mistaking non-viable global sums for real ones even though it satisfies the criteria for Definition 4 for some choices of opt-out threshold. Hence, a better opt-out algorithm that does not give false positives must be sought.

### 3.4 Anonymous ID Assignment

In our first attempt to develop an anonymous opt-out feature, we have shown how Algorithm 4 has the problem of sometimes allowing non-viable global sums (trash) to be accepted as legitimate data. This occurs because its method of making the global sum non-viable is also its method for anonymously declaring that there is an opt-out. In order to address this concern, it is necessary to separate our mechanism for declaring opt-out anonymously from making the global sum non-viable.

Consider, for example, that each node were given a unique, anonymous ticket (or ID). Upon choosing to opt-out, a node could simply turn in their ticket to declare opt-out and then concurrently clobber the global sum with a large random number to make it non-viable. If the ticket were turned in anonymously and if the ticket could not be traced to the node turning it in, then the opt-out would be truly anonymous. Moreover, since the opt-out method is separated from making the global sum non-viable, one would never mistakenly accept illegitimate data as useful.

Therefore, the first step in developing an anonymous opt-out feature is to show how to anonymously assign these tickets or IDs to each node. We describe just such a method in Algorithm 5, which anonymously assigns each node a unique ID using an array version of SS, that is, an array of sums or slots (slot array $S$ of length $N$) is passed around the circuit rather than a single scalar value.
Algorithm 5 Simple Anonymous ID Assignment

Require: Given $M$ nodes.
Ensure: Each node gets a unique, anonymously assigned ID $\in [1, M]$.

1: $Len \leftarrow N$ {For some choice of $N \geq M$}
2: $NumOnes \leftarrow 0$
3: repeat
4: $R \leftarrow ArrayOfRandomNumbers(Len)$ {For some choice of $R_j = NewRandomNumber()$, for $1 \leq j \leq Len$}
5: $S \leftarrow R$ {For each node, if necessary, select a random slot.}
6: for $k = 1$ to $M$ do
7:   $index \leftarrow RandomNumberFromRange(1, Len)$
8:   $S_{index} \leftarrow S_{index} + 1$
9:   {Send $S$ to next node in the secure sum.}
10: end for
11: for $j = 1$ to $Len$ do
12:   $S_j \leftarrow S_j - R_j$ {Remove random number from the slot.}
13: if $S_j = 1$ then
14:   $NumOnes \leftarrow NumOnes + 1$
15: end if
16: end for
17: until $NumOnes = M$

Algorithm 5 works by adding a random number to each element in the slot array $S$ at the source node. The slot array is then sent around the circuit of nodes whereupon each node may select a slot element from the array at random by adding 1 to that element. At the end of the secure sum computation, the source node removes the random numbers from the slot array and declares all the indices that have a value of 1. The relative order of the declared indices will indicate the ID number of the node choosing it. If a collision occurs (a value of 2 or more) the algorithm will repeat until all nodes have chosen an anonymous unique ID.

For example, suppose we have 3 nodes $N_1, N_2, N_3$. The source node, $N_1$, creates a slot array $S$ of length 5 and sets each element to a different random number, also storing the random values redundantly in an array called $R$: $S = R = (3, 4, 10, -2, 1)$. Next, $N_1$ randomly select index 3 of $S$ and adds 1 to it before sending $S$ on to $N_2$.

$N_2$ receives $S$, randomly selects slot 5, adds 1 to it, and passes $S$ on to $N_3$. Similarly, $N_3$ randomly selects slot 1 and adds 1 to it before sending $S$ back to $N_1$. Now $N_1$ has an updated slot array $S$ with $S = (4, 4, 11, -2, 1)$. It may subtract $R$ from $S$ to find which slots have been chosen: $S - R = (4, 4, 11, -2, 1) - (3, 4, 10, -2, 1) = (1, 0, 1, 0, 1)$. Lastly, $N_1$ broadcasts to each node that slots $\{1, 3, 5\}$ have been chosen.

The node choosing a particular slot index sets their ID number to the index of the slot in terms of their relative order being broadcast, that is, $indices = \{1, 3, 5\}$ means ID1 is chosen by $N_3$, ID2 is chosen by $N_1$, and ID3 is chosen by $N_2$. Thus, the IDs of nodes $N_1, N_2, N_3$ are ID2, ID3, ID1 respectively. An illustration
of this example is shown in Figure 3.3.

Figure 3.3: Simple Anonymous ID Assignment, 3 nodes.

Unfortunately, because this simple ID assignment algorithm relies upon SS to anonymously assign unique IDs, it is weak to the collusion of any two nodes, even as we have previously discussed at the end of Chapter 1 (see second to last paragraph). Therefore, we next develop an anonymous ID assignment algorithm (AIDA) that is $K$-collusion resistant. At the end of the AIDA algorithm, each node will also have a unique ID but one that is anonymous with $K$-collusion resistance.

AIDA works by distributing a slot array, $S$, to each node, where $\text{length}(S) = N$, $N \geq M$ nodes. As before, each node randomly selects exactly one slot from the array and adds 1 to that slot. When the slot array returns to the initiating node, the initiating node broadcasts to each other node the element indices for all 1 values found in ascending order.

If a collision occurs ($\text{value} > 1$) or a 0 is found in the $S$ array, that element index will not be broadcast. The node that chose an index in the broadcast will then set their ID number to the position of their index relative to the ordering of the broadcast (this relative order spans each round of the algorithm). For example, if $\text{indices} = \{23, 57, 89, 110\}$ have been broadcast, then the node choosing index 89 in $S$ will be ID #3—a unique, anonymous ID. This process may be repeated until the number of indices broadcast matches the number of nodes in the data mining operation.

$K$-collusion resistance for the anonymity in AIDA is conferred by sending $S$ via CPSS, that is, the slot array is itself randomly partitioned and sent along each edge-disjoint Hamiltonian cycle. The choosing of a slot in the array is equivalent to adding a count value of one in CPSS, except that in AIDA we have an array instead of a scalar. See Algorithm 6 for pseudocode to this approach.

The length of the slot array $S$ shown in Algorithm 6 should be chosen to be an integer value $N \geq M$, where $M$ is the number of nodes. Indeed, the choice of $N$ depends on the size of $M$. We discuss the choosing
of $N$ as it relates to efficient AIDA termination in Chapter 4.

**Algorithm 6** AIDA: Anonymous ID Assignment, with $K$-collusion Resistance

**Require:** Given $M$ nodes and $C$ edge-disjoint Hamiltonian cycles.
**Ensure:** Each node gets a unique, anonymously assigned ID $\in [1, M]$.

{For $N_1$, the initiating node.}

1: $Len \leftarrow N$ \{For some choice of $N \geq M$\}
2: Initialize $S$ array. \{$S_j(i) = 0$, for $1 \leq i \leq C$ and $1 \leq j \leq Len$\}
3: $NumOnes \leftarrow 0$
4: repeat
5: \begin{itemize}
   \item $R \leftarrow ArrayOfRandomNumbers(Len)$ \{ $R_j = NewRandomNumber()$, for $1 \leq j \leq Len$ \}
   \item \textbf{RandomlySelectSlot ($S, R, Len, C$)} \{$S = S + R$, increment $S_{selected}$, RandomlyPartition($S_j, C$) $\forall j$.} 
   \item for $i = 1$ to $C$ do
   \begin{itemize}
      \item Send $S^{(i)}(j)$ to the next node in cycle $i$. \{$S^{(i)}_j(i) \in S^{(i)}(i)$, for $1 \leq j \leq Len$\}
   \end{itemize}
5: end for
6: \end{itemize}
7: for $k = 2$ to $M$ do
8: \begin{itemize}
   \item Initialize $Z$ array. \{$Z_j = 0$, for $1 \leq j \leq Len$\}
   \item \textbf{RandomlySelectSlot ($S, Z, Len, C$)} \{$S = S + Z$, increment $S_{selected}$, RandomlyPartition($S_j, C$) $\forall j$.} 
   \item for $i = 1$ to $C$ do
   \begin{itemize}
      \item Send $S^{(i)}(j)$ to the next node in cycle $i$. \{$S^{(i)}_j(i) \in S^{(i)}(i)$, for $1 \leq j \leq Len$\}
   \end{itemize}
8: end for
9: end for
10: for $j = 1$ to $Len$ do
11: \begin{itemize}
   \item $Sum \leftarrow 0$
   \item \{Recover the value at $S_j$ and see if it is selected.\}
   \item for $i = 1$ to $C$ do
   \begin{itemize}
      \item $Sum \leftarrow Sum + S^{(i)}_j$
   \end{itemize}
11: end for
12: \end{itemize}
13: \begin{itemize}
   \item $Sum \leftarrow Sum - R_j$ \{Remove random number from the sum.\}
   \item if $Sum = 1$ then
   \begin{itemize}
      \item $NumOnes \leftarrow NumOnes + 1$
   \end{itemize}
13: end if
14: \end{itemize}
15: end for
16: until $NumOnes = M$

The pseudocode for the AIDA subroutine, RandomlySelectSlot, is given in Algorithm 7. In turn, the function RandomlyPartition called by RandomlySelectSlot is described earlier in Algorithm 3.
Algorithm 7 RandomlySelectSlot(S, A, Len, C)

Require: Given array S and A of length Len, with S partitioned into C partitions.
Ensure: Randomly select one slot in the C-partitioned array S.

1: selectedSlot $\leftarrow$ RandomNumberInRange(1, Len)
   \{ThisNode is the node calling this function.\}
2: for $j = 1$ to Len do
3:   if ThisNode has no assignment AND $j = selectedSlot$ then
4:     $X \leftarrow 1 + A_j$
5:   else
6:     $X \leftarrow A_j$
7: end if
8: RandomlyPartition($X, C$) \{$\sum_{i=1}^{C} X^{(i)} = X, 0 \neq X^{(i)} \in \mathbb{Z}$\}
9: for $i = 1$ to $C$ do
10:   $S_j^{(i)} \leftarrow S_j^{(i)} + X^{(i)}$
11: end for
12: end for

3.5 CPSS with Opt-out

Recall that Algorithm 4 suffered from the complication of sometimes passing non-viable global sums as legitimate data because it failed to separate the method for declaring opt-out from making the global sum non-viable. In the previous section, we invented the AIDA algorithm to allow us to randomly assign unique, anonymous IDs to each node with $K$-collusion resistance—now enabling the design of an opt-out method that depends on anonymous IDs only and not on making the global sum non-viable.

Therefore, we next develop a CPSS algorithm with an opt-out feature satisfying the criteria of Definition 4 but without the problems of Algorithm 4. We term this algorithm: “cycle-partitioned secure sum with opt-out” or CPSOO.

Using Algorithm 6 we have seen that we are able to assign each node in our data mining operation a unique ID that is anonymous with $K$-collusion resistance. By integrating AIDA into CPSOO we may create an array of bits $B$, where the index of each element in $B$ corresponds to a node’s unique ID number in the data mining operation. The purpose of the bit array $B$ is to allow nodes to anonymously opt-out without interfering with another node’s opt-out declaration and to avoid error by separating the opt-out declaration from the method for making the global sum non-viable. We see that the anonymous, $K$-collusion resistant IDs allow this non-interference by making sure each node has its own private element that cannot be mistakenly cancelled out by two nodes XORing over the same bit.

In the tradition of CPSS, each element of $B$ is partitioned and sent along each cycle with the traditional sum values. However, this partitioning is a random bit-partition instead of an integer one. For example, suppose again we have $M$ nodes and that for some node $N_j$ with ID $\#k$ ($1 \leq k \leq M$ and $j \neq k$) there is a
bit value $B_k$ in array $B$. Then if we bit-partition $B_k$ for $C$ cycles, we have $B_k = \bigoplus_{i=1}^{C} B_k^{(i)}$ where $\oplus$ is the bitwise eXclusive-OR (XOR) operation. (Recall that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $0 \oplus 0 = 0$.)

The opt-out mechanism for CPSOO follows very naturally from this data structure. If a node wishes to opt-out, it simply sets all bit-partitions such that their XOR is equal to 1, that is, its bit-value in $B$ is set to 1. Additionally, the node opting out adds a random number instead of its real count value to the sum in order to ensure that the global sum will be non-viable. More importantly, the true value of the withdrawing node cannot be known by any other nodes. The psuedocode for CPSOO is given in Algorithm 8.
Algorithm 8 CPSOO: Cycle-partitioned Secure Sum with Opt-out

Require: Given $M$ nodes and $C$ edge-disjoint Hamiltonian cycles.

Ensure: $\sum_{k=1}^{M} V_k = \text{GlobalSum}$ OR opt-out: $\sum_{k=1}^{M} V_k \neq \text{GlobalSum}$

1: AnonymousIDAssignment() \{For each $k$, $ID_k$ is the unique ID for node $N_k$, $1 \leq ID_k \leq M$\}
   \{For $N_1$, the initiating node.\}
2: Initialize array $B$. $\{B_j^{(i)} = 0$, for $1 \leq i \leq C$ and $1 \leq j \leq M\}$
3: $R \leftarrow \text{NewRandomNumber}()$
4: for $j = 1$ to $M$ do
5: $T_j \leftarrow \text{NewRandomBit}()$
6: $B_j \leftarrow \text{RandomlyBitPartition}(T_j, C)$ \{$T_j^{(1)} \oplus T_j^{(2)} \oplus \cdots \oplus T_j^{(C)} = T_j$\}
7: if $j = ID_1$ AND $N_1$ opts out then
8: $V_1 \leftarrow R$
9: $B_j \leftarrow \text{RandomlyBitPartition}(T_j \oplus 1, C)$
10: else if $j = ID_1$ then
11: $V_1 \leftarrow V_1 + R$
12: end if
13: end for
14: RandomlyPartition($V_1, C$) \{$\sum_{i=1}^{C} V_1^{(i)} = V_1$, $0 \neq V_1^{(i)} \in Z$\}
15: for $i = 1$ to $C$ do
16: Send $V_1^{(i)}$ & $B_j^{(i)}$ to the next node in cycle $i$. \{$B_j^{(i)} \in B^{(i)}, \forall j \in [1, M]$\}
17: end for
18: for $k = 2$ to $M$ do
19: for $j = 1$ to $M$ do
20: if $j = ID_k$ AND $N_k$ opts out then
21: $B' \leftarrow \text{RandomlyBitPartition}(1, C)$ \{$1 = \bigoplus_{i=1}^{C} B'_i$\}
22: $V_k \leftarrow \text{NewRandomNumber}()$
23: else
24: $B' \leftarrow \text{RandomlyBitPartition}(0, C)$ \{$0 = \bigoplus_{i=1}^{C} B'_i$\}
25: end if
26: for $i = 1$ to $C$ do
27: $B_j^{(i)} \leftarrow B_j^{(i)} \oplus B'_i$
28: end for
29: end for
30: RandomlyPartition($V_k, C$) \{$\sum_{i=1}^{C} V_k^{(i)} = V_k, 0 \neq V_k^{(i)} \in Z$\}
31: for $i = 1$ to $C$ do
32: $V_k^{(i)} \leftarrow V_k^{(i)} + V_{\text{received}}^{(i)}$
33: Send $V_k^{(i)}$ & $B_j^{(i)}$ to the next node in cycle $i$. \{$B_j^{(i)} \in B^{(i)}, \forall j \in [1, M]$\}
34: end for
35: end for
36: $\text{GlobalSum} = \sum_{i=1}^{C} V_{\text{received}}^{(i)} - R$
37: $\text{BitSum} \leftarrow \sum_{j=1}^{M} ((\bigoplus_{i=1}^{C} B_j^{(i)}) \oplus T_j)$ \{XOR bits from each cycle, cancel random bit $T_j$, add array.\}
38: if $\text{BitSum} \geq 1$ then
39: Broadcast $\text{SumCancelled}$ to each other node.
40: else
41: Broadcast $\text{GlobalSum}$ to each other node.
42: end if

The function $\text{RandomlyBitPartition}$ is also defined in Algorithm 9.
Algorithm 9 RandomlyBitPartition(B, Q)

Require: Given bit value \( B \) and length \( Q \).
Ensure: \( \bigoplus_{i=1}^{Q} B'_i = B \). Returns bits \( B' \).

1: \( Parity \leftarrow 0 \)
2: for \( i = 1 \) to \( Q - 1 \) do
3: \( B'_i \leftarrow \text{NewRandomBit}() \)
4: \( Parity \leftarrow Parity \oplus B_i \)
5: end for
6: \( B'_Q \leftarrow B \oplus Parity \)
7: return \( B' \)

A diagram of some typical inputs and outputs for an arbitrary node “N2” in mid-computation of CPSOO is given in Figure 3.4. N2’s anonymous ID is 3. Here there are \( C = 2 \) cycles and \( M = 5 \) nodes. The two edge-disjoint cycles are represented by the dotted and dashed lines, with two distinct sending and two distinct receiving nodes assumed for each ray. The bracketed numbers are the partitioned running subtotals of the count value of the itemset whereas the parenthetical numbers are for the 5 opt-out bits of the bit-array \( B \). A normal and opt-out example are shown.

See that in the normal “no opt-out” scenario, the count value 7 is randomly partitioned into 4 and 3
and added to the (incoming) 9 and 2 subtotal respectively. The incoming bit-array partitions (01101) and (00001) are modified to (10100) and (11000), but their XOR is not modified, that is: 01101 ⊕ 00001 = 01100 = 10100 ⊕ 11000.

However, in the “opt-out” scenario of Figure 3.4, a large random number is generated (456), randomly partitioned into 430 and 26, and added to each subtotal to make the global sum non-viable. To signal for opt-out, the node modifies the incoming bit-array partitions (01101) and (00001) such that the XOR of the incoming partitions is different from the XOR of the outgoing partitions, that is: 01101 ⊕ 00001(= 01100) ≠ 11100 ⊕ 10100(= 01000).

Notice also that 01000 ⊕ 01100 = 00100. The third bit is set because “N2” has ID3. Since only “N2” owns ID3 and the ID is anonymous, the opt-out may occur anonymously and without the possibility of collision, that is, no opt-out can cancel out another opt-out by both nodes XORing the same bit (which if it did occur would result in the same problem as described Algorithm 4 where a non-viable global sum is taken to be legitimate).

We have already discussed how AIDA is equivalent to CPSS in terms of collusion resistance. The summing of count values in CPSOO also uses CPSS without alteration, and so it is also K-collusion resistant. The prime difference between CPSOO and CPSS is the addition of an opt-out feature involving bit partitions. We assert that CPSOO’s opt-out feature is also K-collusion resistant.

**Theorem 2.** Given any network with C edge-disjoint Hamiltonian cycles and M nodes such that M > 4, CPSOO is “K-collusion resistant for anonymous opt-out” where K = 2C − 1.

**Proof.** We provide a sketch of the proof—showing that CPSOO satisfies the criteria for Definition 4. As before, let there be a network of M ≥ 5 nodes with C ≥ 2 edge-disjoint Hamiltonian cycles.

Part (a) of Definition 4 requires that the global sum become “non-viable” upon opt-out, that is, ∑_{k=1}^{M} V_k ≠ GlobalSum, where each V_k is the count value of node N_k. According to Algorithm 8, a node opting out, say N_{k_o}, replaces its count value with some large positive random number R_{k_o} such that R_{k_o} ≠ V_{k_o}. See that R_{k_o} ≠ V_{k_o} implies that GlobalSum = V_1 + V_2 + ⋯ + R_{k_o} + ⋯ + V_{M-1} + V_M ≠ M \sum_{k=1}^{M} V_k. Thus, the global sum is non-viable upon opt-out.

Part (b) says that any set of K nodes or fewer cannot calculate the count value of any remaining node. By carefully examining CPSOO we see that the portion of code for summing count values is equivalent to the code implemented for CPSS (Algorithm 2). Hence, Theorem 1 implies that CPSOO is also K-collusion resistant with respect to summing count values based upon this equivalence, where K = 2C − 1. Therefore
part (b) is satisfied.

Finally, part (c) of the definition specifies that any set of \( K \) or fewer nodes cannot determine if some other node not in the \( K \)-node set has opted out. In order to prove this we must first break down the opt-out feature into its two components: AIDA and the cycle-partitioned secure bit-sums (XORs).

For AIDA, the reader has in fact already observed how the anonymity of each ID is \( K \)-collusion resistant. Indeed, as in part (b), AIDA’s choosing of slots is actually implemented by specially interpreting the results of a array-based CPSS (each element of the array may be seen as its own CPSS operation). Therefore, by Theorem 1 and the equivalence of AIDA to CPSS in terms of summing values (choosing slots), we see that the anonymity of each node’s ID is indeed \( K \)-collusion resistant where \( K = 2^{C-1} \).

Next we turn to the cycle-partitioned secure bit-sums (or bitwise exclusive-ORs). It remains to show that XOR’ing bits is equivalent to summing count values in terms of \( K \)-collusion resistance for CPSOO, where \( K = 2C - 1 \). Let \( B \) be an array of \( M \) bits with \( B_k \) being the bit for node \( N_k \). Moreover, let \( B_k = \bigoplus_{i=1}^{C} B_k^{(i)} \), where \( \oplus \) is the exclusive-OR bitwise operation. (We refer to node \( N_k \) as having opt-out bit \( B_k \) in bit array \( B \) for convenience; however, each node is randomly assigned a unique ID and therefore such a mapping is not guaranteed in practical use.)

A node \( N_k \) has opted opt if its bit \( B_k \) is set to 1 and not 0 (\( 1 \leq k \leq M \)). For any \( i \) and \( k \geq 2 \), see that the bit set by \( N_k \) for its bit value \( B_k^{(i)} = B_{k,\text{prev}}^{(i)} \oplus B_{k,\text{next}}^{(i)} \), where \( B_k^{(i)} \) is the bit value of \( B_k \) received by the next node in cycle \( i \), and where \( B_{k,\text{prev}}^{(i)} \) is the bit value of \( B_k^{(i)} \) sent to \( N_k \) by its previous node in the cycle.

By the Hamiltonicity of our cycle, the node sending \( B_{k,\text{prev}}^{(i)} \) and the node receiving \( B_{k,\text{next}}^{(i)} \) are different from \( N_k \) and each other. To calculate \( B_k \) recall that we need each \( B_k^{(i)} \) since \( B_k = \bigoplus_{i=1}^{C} B_k^{(i)} \). The neighboring nodes previous to and following any fixed \( N_k \) are each unique between cycles since we choose our cycles to be edge-disjoint. Hence, a total of \( 2C \) nodes are needed to determine any fixed \( B_k \) where \( k \geq 2 \), that is, \( 2C - 1 \) or fewer nodes different from \( N_k \) cannot calculate \( B_k \).

For \( B_1 \), notice that in Algorithm 8 the initiating node, \( N_1 \), masks its bit-partitioned array \( B \) with random bits before sending them along each cycle. Since \( N_1 \) never exposes its random bits, as it does in CPSS indirectly (see Equation 3.6), and since opt-out is declared anonymously, no node not \( N_1 \) can determine
if $N_1$ has set its opt-out bit $B_1$.

On the other hand, if $M - 1$ nodes collude they can trivially determine $N_1$ has opted out (or any other fixed node for that matter). Notice from Theorem 1 that the maximum value of $C$ for $M \geq 3$ is $p$ when $M = 2p + 1$ is odd and $p - 1$ when $M = 2p$ is even. So if $M$ is even, we have $M - 1 = 2p - 1 = 2(C + 1) - 1 = 2C + 1$ and $M - 1 = 2p = 2C$ when odd. This implies that $M - 1 \geq 2C - 1$ for any choice of $M \geq 3$ and that . Thus, the anonymity of the opt-out for $N_1$ has a $K$-collusion resistance no weaker than $K = 2C - 1$.

Now we see that for all nodes our cycle-partitioned secure bit-sums are $K$-collusion resistant where $K = 2C - 1$. Hence, by collusion resistance of AIDA and our bit partition opt-out, we have shown that no $K$ or fewer nodes can deduce that any node not in the $K$-node set has opted out, completing part (c) of Definition 4 and finishing the proof.

Having developed a useful algorithm for privacy-preserving data mining that both resists collusion and provides an opt-out feature against unwanted statistical disclosure, we now turn to the finer points of its practical use, addressing communication complexity and numerical issues.
CHAPTER 4

ADDITIONAL CONSIDERATIONS

We have described algorithms like CPSS and AIDA in the previous chapter and their usefulness as tools for privacy-preserving data mining. We next review some of the practical concerns of running these algorithms, namely algorithmic complexity in the form of communication complexity, program termination for AIDA, and numerical issues involved in the selection of random numbers.

4.1 Numerical Considerations

The use of random numbers is vital to the operation of the SS, CPSS, and AIDA algorithms. In AIDA, random numbers are used to choose one out of \( N \) slots. This use is straightforward and requires little consideration. In SS and CPSS a random number is added to the value of the initiating node or site to mask that value.

This masking must be done carefully. If the sum is large and our random number is small then the sum can be estimated to a certain degree which may defeat the purpose of masking our sum. Hence, we need to choose a random number which is non-zero and sufficiently large. If our choice of random number is too large; on the other hand, there exists a risk of creating an overflow situation.

For example, consider our value \( V = 110051 \) and our random number \( R = 2 \), then \( sum = 110053 \). Clearly, the sum is not well masked. So we might like to choose a larger random number, like \( R = 1123453 \) such that \( sum = 1233504 \). On the other hand, to contrive the opposite extreme, suppose \( R = 1000000 \) is the maximum positive number on our machine, then \( sum \) is going to overflow.

Our intuition then is to choose a random number no less than ten times the average value in our sum, and no greater than a few orders of magnitude greater than the global sum. It may not be obvious to the
miner, however, how close the local count value is to the average. To overcome this, the miner might average the counts of all frequent 1-itemsets in their local database. The miner might use the maximum 1-itemset count times the number of nodes in the network as an estimation for global sum. Taking a few orders of magnitude above each of these values might produce a fine range for random number generation.

For example, suppose we have $M = 5$ nodes and that the average count of all frequent 1-itemsets is 542 and the maximum count is 4573 in our local database. Then the miner might conclude a good range for choosing random numbers would be $R \in [5420, 22865000]$, where 5420 is $\text{RandMin}$ and 22865000 is $\text{RandMax}$.

Suppose the maximum value for our machine is 999999999, then $999999999 - (5 + 1) \cdot 22865000 = 862809999 > 0$ means our range for $R$ is estimated to be safe from overflow. Why? If the initiating node chooses $\text{RandMax}$ to mask its value and if the value at each site is no more than $\text{RandMax}$ (a large over-estimate), then we would expect the global sum to be no more than $(M + 1) \cdot \text{RandMax}$.

CPSS also randomly partitions the count values at each site such that the sum of all the partitions equals the count value. The number of partitions equals the number of edge-disjoint Hamiltonian cycle in the algorithm. For example, let $\text{count} = 3$, we may partition it into $C = 5$ values 2, $-1$, 25, $-20$, $-3$ such that $3 = 2 + (-1 + 25 + (-20) + (-3)$.

Algorithmically, we actually choose $C - 1$ random integers and let the last value be our count value less the sum of those numbers. Using our last example, see that if we choose 2, $-1$, 25, $-20$ randomly then $3 - (2 + (-1 + 25 + (-20)) = -3$. So 2, $-1$, 25, $-20$, $-3$ are our 5 random partitions.

To address the numerical concerns mentioned above for making random partitions of count values, notice that for each individual cycle, we have $M$ partition values being added by each node in that cycle, plus one random value partition. Hence, we may choose our random partitions like before, but such that the absolute value of the partition is no larger than $\text{RandMax}$.

Having addressed the numerical concerns of CPSS and SS, we now turn again to the element of randomness in AIDA. Although the choosing or random numbers in AIDA is straightforward and introduces no issues with overflow and the like, this randomness does introduce a non-zero probability that AIDA may, in fact, never terminate. This unlikely event is described in the next section and addressed by practical use.

### 4.2 Anonymous ID Assignment Termination

The reader will remember that the Anonymous ID Assignment algorithm, Algorithm 6, relies on randomly choosing as many as $M$ slots (the number of nodes) from a slot array of size $N$ for each round of selection until each node (hopefully!) has been given a unique, anonymous ID. Unfortunately though, as many as $M$
nodes have the possibility of randomly colliding; that is, each node may randomly choose the same slot in single selection round.

Thus, we have a non-zero probability that no progress is made in each selection round and subsequently the non-zero probability that the ID assignment algorithm may never terminate. However, given a good choice of $N$ for each $M$, we may, in practical use, show that AIDA terminates in short order, or at worst, terminates eventually.

![Figure 4.1: AIDA—Maximum Iterations for 10,000 trials.](image)

We ran a simulation for AIDA varying both the slot size $N$ and the node number $M$ and observed the maximum number of iterations needed to terminate the program, performing 10,000 trials. The results are presented in Figure 4.2 for $M = 5$ to 30 node in increments of 5, $N = 30$ to 505 slots in increments of 25. The maximum number of iterations for each $(M, N)$ pair is recorded on the Z axis. Figure 4.2 graphs this data using a 3D surface.

We also recorded the average number of iterations needed to complete the AIDA algorithm, again taken
over 10,000 trials. We use the same layout as before—varying nodes 5 to 30 by 5 and slots 30 to 505 by 25. See Figure 4.2 for the 3D graph of our simulation results.

Observing the graphical data, we see that in practical use, AIDA terminates within a maximum of 7 rounds or less, given a good choice of $N$. On average AIDA terminates with 4 rounds or less, with the majority of the $(M,N)$ node/slot pairs needing only 1 to 2 rounds to terminate. Given our choice of 10,000 trials we are confident that this data is consistent and indicative of practical use of AIDA.

While we have shown that, in practical use, AIDA terminates fairly quickly, the next most natural question is how much computational power does it require for the program to run each round? Given our algorithms are not computationally heavy, we choose communication as a metric for studying the complexity. Communication is likely to be the bottleneck in any algorithm that is computationally light but has moderate network dependence.
4.3 Communication Complexity

We discuss the computational complexity of our algorithms in terms of communication. The basic unit for communication complexity we define as a single message sent from one site in the network to one other different site. The size of a message is also useful in considering communication cost, but in terms of communication complexity it is not.

For example, the sending of an array of values at stage 1 of an algorithm is equivalent, in terms of communication complexity, to the sending of a boolean value at stage \( N \) of an algorithm given that both values were sent from a single node to another different single node and as part of the same message.

We believe our choice of communication complexity here is valid since the size of the messages in our algorithms are all relatively small and may be sent in a packet or two.

4.3.1 CPSS, Algorithm 2

We are given \( M \) nodes and \( C \) cycles. Along each cycle, \( M \) messages are passed from the source node to each other node and then back again, so we have \( C \cdot M \) messages total. Additionally, at the end of the algorithm, GlobalSum is broadcast to each other node, for another \( M - 1 \) messages. Let the communication complexity be \( c(M) \) where \( c(M) = C \cdot M + (M - 1) \). We show that \( c(M) = O(M^2) \).

Proof. Recall that we assume our nodes \( M \geq 3 \). Moreover, observe that the greatest number of cycles we may obtain for any such \( M \) is \( C = \text{floor}(\frac{M-1}{2}) \) assuming a fully-connected network.

\[
c(M) = C \cdot M + (M - 1) \quad (4.1)
\]

\[
< C \cdot M + M \quad \text{since } M - 1 < M \text{ for } M \geq 3 \quad (4.2)
\]

\[
< \frac{M}{2} \cdot M + M \quad \text{since } \lfloor \frac{M - 1}{2} \rfloor < \frac{M}{2} \text{ for } M \geq 3 \quad (4.3)
\]

\[
= \frac{1}{2}M^2 + M < M^2 \quad \text{for } M \geq 3 \quad (4.4)
\]

So we see that \( c(M) \approx \frac{1}{2}M^2 + M \) and that \( c(M) = O(M^2) \). \qed

4.3.2 Anonymous ID Assignment, Algorithm 6

Consider \( M \) nodes and \( C \) cycles with \( I \geq 1 \) selection iterations of the algorithm. Like CPSS, we have \( C \) values sent along each cycle, in this case our “slot” array. Hence, we have \( C \cdot M \) messages. At the end
selection iteration of the algorithm, the source node must broadcast the chosen ID’s (M ID’s) to each other node (M − 1), meaning we have a worst case scenario of M · (M − 1) messages for the algorithm to complete.

Thus, we may define the communication complexity \( c(M) = I \cdot C \cdot M + M \cdot (M - 1) \). We show that \( c(M) \in O(M^2) \).

**Proof.** We assume \( M \geq 3 \). From our previous proof we see that \( C < \frac{M}{2} \), so we have:

\[
c(M) = I \cdot C \cdot M + M \cdot (M - 1) \tag{4.5}
\]
\[
< I \cdot \frac{M}{2} \cdot M + M \cdot (M - 1) \mid \text{ as before} \tag{4.6}
\]
\[
= I \cdot \left( \frac{1}{2} M^2 \right) + M^2 - M \tag{4.7}
\]
\[
< I \cdot \left( \frac{3}{2} M^2 \right) \tag{4.8}
\]
\[
\text{since } M \geq 3 \tag{4.9}
\]

Thus we prove that in the worst case the communication complexity \( c(M) \approx I \cdot \left( \frac{3}{2} M^2 \right) \) and that \( c(M) \in O(M^2) \).

We have considered the computational complexity of CPSS and AIDA and shown them to be of practical usefulness. We have also simulated AIDA to demonstrate that it, in general, terminates in a reasonable amount of time given a good choice of slots \( N \). Finally, we have shown how local database information can be used to estimate a good range for choosing random numbers in CPSS. In the next chapter we summarize our contribution to privacy-preserving data mining and suggest some future work.
CHAPTER 5

CONCLUSION

We discuss our contribution to privacy-preserving data mining and give some suggestions for future work.

5.1 Contribution

First, we have given the reader some examples for generating edge-disjoint Hamiltonian cycles in fully-connected and torus graphs, which are useful to our anonymous opt-out algorithm (see Chapter 2 for more details). Our algorithm (CPSOO, see Algorithm 8 in Chapter 3) was then placed solidly in the mathematical context of edge-disjoint Hamiltonian cycles which we used to prove mathematically that CPSOO is \( k \)-collusion resistant where \( k < 2C \) and \( C \) is the number of edge-disjoint Hamiltonian cycles. This collusion resistance is comparable to other algorithms listed in the literature.

Unlike other works, however, CPSOO anonymously assigns unique IDs to each node (see AIDA, Algorithm 6). The anonymous IDs are then used to implement an anonymous opt-out feature for any node wishing to withdraw from the data mining operation. The anonymous opt-out feature in CPSOO is useful in addressing the security concerns involved in statistical disclosure scenarios, such as when a node discovers a \( k \)-itemset is sensitive and does not want to mine \( k+x \)-itemset supersets \((x = 1, 2, \ldots)\) or for when exogenous information reveals that there are nodes contributing too much or too little to the global sum. Indeed, unwanted statistical disclosure can break \( k \)-collusion resistance of CPSS-like algorithms that do not have anonymous opt-out.

We have also discussed some numerical consideration useful to implementing our algorithm. In particular, we have given guidelines for the use of random numbers in our algorithms and shown that in practical use, AIDA terminates. Moreover, we have also demonstrated that our algorithms fall under \( O(M^2) \), where \( M \) is the number of nodes in the network, in terms of communication complexity, which we feel is quite reasonable.
considering the collusion-resistance afforded.

5.2 Future Work

We mentioned earlier in Chapter 3 that an algorithm described in (Urabe et al., 2007) achieves a similar degree of collusion resistance but does not use edge-disjoint Hamiltonian cycles. For future work, we believe that a similar opt-out feature as employed in CPSOO could also be applied to their research efforts in order to address the additional privacy concerns raised by statistical disclosure.

Moreover, although we have developed a method for anonymous ID assignment (AIDA) that terminates in a reasonable amount of time in practical use, there still exists a non-zero probability that AIDA may never terminate. It may be useful to construct a method for anonymous ID assignment that always terminates in a fixed numbers of iterations. Such an algorithm in practical use, however, should be tested for efficiency versus our AIDA algorithm.

Finally, we recommend a full study of the use of random numbers in secure sum. One might determine how our guidelines and any new inventions fair in practical usage and explore a broader range of security concerns related to these numerical issues.
References


Kantarcioglu, M. (2004). Privacy-preserving distributed mining of association rules on horizontally partitioned data. IEEE Transactions on Knowledge and Data Engineering, 16(9), 1026–1037. (Senior Member-Chris Clifton)


