

2007

# Graph Theoretic Modeling: Case Studies In Redundant Arrays Of Independent Disks And Network Defense

Sanjeeb Nanda

*University of Central Florida*



# GRAPH THEORETIC MODELING: CASE STUDIES IN REDUNDANT ARRAYS OF INDEPENDENT DISKS AND NETWORK DEFENSE

by

SANJEEB NANDA
M.S. University of Louisville, 1991
B.E. Birla Institute of Technology and Science, 1989

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2007

Major Professor: Narsingh Deo

# ABSTRACT

Graph theoretic modeling has served as an invaluable tool for solving a variety of problems since its introduction in Euler's paper on the Bridges of Königsberg in 1736 [1]. Two amongst them of contemporary interest are the modeling of Redundant Arrays of Inexpensive Disks (RAID), and the identification of network attacks. While the former is vital to the protection and uninterrupted availability of data, the latter is crucial to the integrity of systems comprising networks. Both are of practical importance due to the continuing growth of data and its demand at increasing numbers of geographically distributed locations through the use of networks such as the Internet.

The popularity of RAID has soared because of the enhanced I/O bandwidths and large capacities they offer at low cost. However, the demand for bigger capacities has led to the use of larger arrays with increased probability of random disk failures. This has motivated the need for RAID systems to tolerate two or more disk failures, without sacrificing performance or storage space. To this end, we shall first perform a comparative study of the existing techniques that achieve this objective. Next, we shall devise novel graph-theoretic algorithms for placing data and parity in arrays of $n$ disks ($n \geq 3$) that can recover from two random disk failures, for $n = p - 1$, $n = p$ and $n = 2p - 2$, where $p$ is a prime number. Each shall be shown to utilize an optimal ratio of space for storing parity. We shall also show how to extend the algorithms to arrays with an arbitrary number of disks, albeit with non-optimal values for the aforementioned ratio.

The growth of the Internet has led to the increased proliferation of malignant applications seeking to breach the security of networked systems. Hence, considerable effort has been focused on detecting and predicting the attacks they perpetrate. However, the enormity of the

Internet poses a challenge to representing and analyzing them by using scalable models. Furthermore, forecasting the systems that they are likely to exploit in the future is difficult due to the unavailability of complete information on network vulnerabilities. We shall present a technique that identifies attacks on large networks using a scalable model, while filtering for false positives and negatives. Furthermore, it also forecasts the propagation of security failures proliferated by attacks over time and their likely targets in the future.

I dedicate this thesis to my son Alex. He has been the source of my inspiration during the course of my doctoral research. His smile, laughter and mischief has made each day special, and helped me to persevere.

# ACKNOWLEDGMENTS

I thank my advisor, Dr. Narsingh Deo for his mentorship that has been vital to my growth as a scholar, and for his continued support and encouragement during the course of my research. I am grateful to Dr. Charles Hughes for enabling me to rejoin the doctoral program and helping me to reach many significant academic milestones. I also thank the members of my research committee, professors David Workman and Xin Li for their help and advice.

I thank my fellow researchers in the Center for Parallel Computation, Hemant Balakrishnan, Aurel Cami, and Zoran Nikoloski, for sharing their research experience over the years with me. I finally thank Jenny Shen for supporting me on various administrative matters during the pursuit of my degree.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1. INTRODUCTION

From its inception, graph theory has proved invaluable in modeling numerous problems and deriving their solutions. Two contemporary ones of practical interest are the design of Redundant Arrays of Independent Disks (RAID) and, the identification of network attacks and the forecasting of their propagation.

## 1.1 A case for RAID tolerant to two or more random disk failures

In contrast to the growth of CPU throughputs, enhancements to I/O bandwidth have been relatively modest. In fact, I/O limitations continue to bottleneck the performance of many applications to this day. This fact motivated the introduction of Redundant Arrays of Inexpensive Disks (RAID) [14, 32, 33] that substitutes large expensive disks with arrays of independent disks to provide greater effective I/O bandwidth. Note that, the word *independent* in RAID may be found substituted by *inexpensive* in literature on this area. However, an array's probability of failure exceeds that of each disk comprising it. Consider the disks that are manufactured with some of the most stringent requirements on durability, such as those with SCSI and Fibre Channel interfaces. They have mean time between failures (MTBF) of at most 1,400,000 hours. Manufacturers list the MTBF of a disk to be *nh* hours if, upon running an array of *n* such disks, the first disk fails after *h* hours. Thus, if an array is comprised of 1024 disks, with each having an MTBF of 1,400,000 hours, one disk may be expected to fail approximately every 57 days. However, this is a misleading measure of reliability for several reasons. First, such tests use disks with pristine electromechanical components that, unlike in older disks, have not degraded due to overheating, vibration, and shock from rough handling. Second, a large number of disks, such as

those with ATA and SATA interfaces, utilize less resilient components than SCSI and Fibre Channel ones to reduce cost. Hence, arrays comprised of such disks are likely to experience failures sooner than equal sized ones with more robust disks.

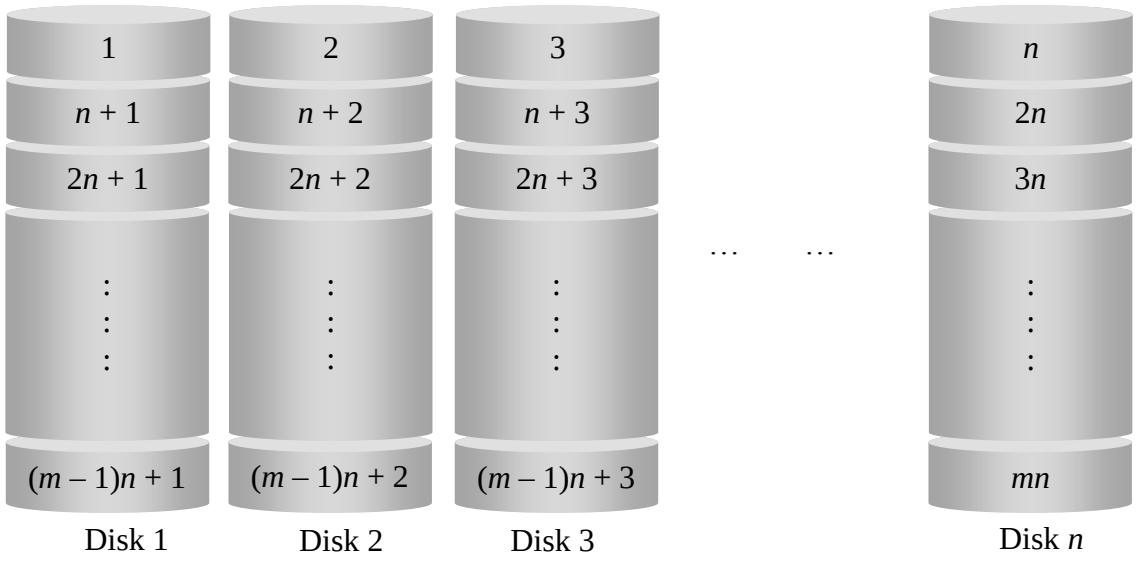


Figure 1: RAID level 0 with *n* disks and *m* blocks per disk.

A RAID level 0 with *n* disks may partition each disk into equal sized blocks and order the blocks in a round-robin fashion. This enhances I/O throughput by permitting up to *n* blocks to be accessed concurrently when reading from, or writing to the array sequentially. Figure 1 displays the order of the data blocks in a RAID level 0 with *n* disks and *m* data blocks per disk. But, RAID level 0 does not offer any redundancy. In contrast, levels 1 through 5 provide tolerance to the failure of exactly one disk in the array. RAID level 1 is simply a pair of disks with the data on any one being mirrored on the other, and RAID level 5 consists of at least three disks with rotating bit-wise parity placement to provide redundancy.

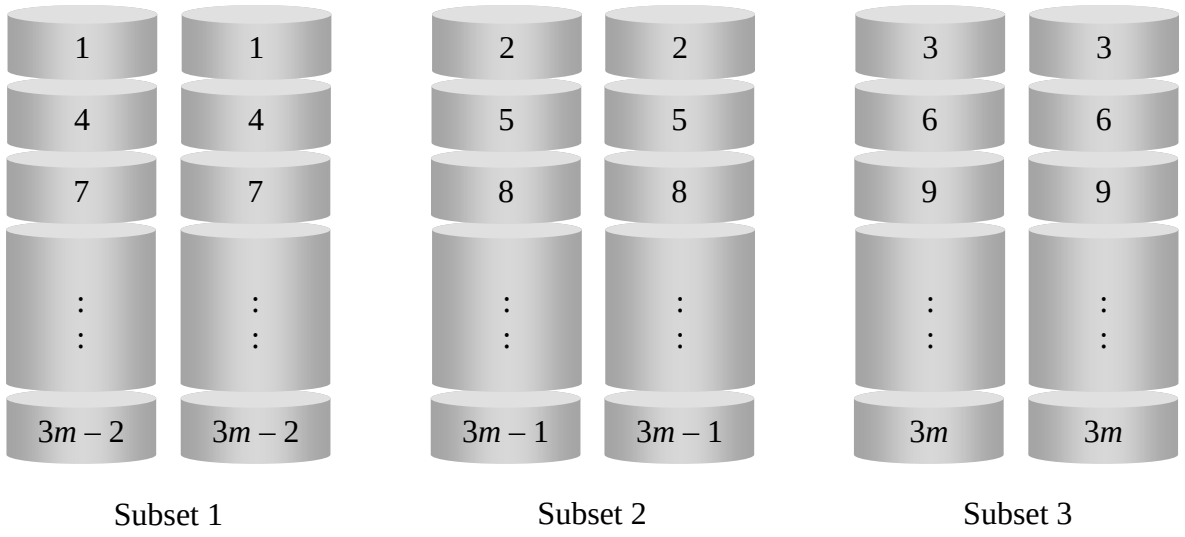


Figure 2: A RAID level 10 with three virtual disks, each being a RAID 1.

Since RAID levels 1 through 5 can tolerate only one disk failure, their use is limited to small arrays where the probability of a second disk failing before a previously failed disk has been reconstructed, is negligible. A popular technique that addresses this challenge partitions a set of disks into subsets, with each being a single-disk fault-tolerant RAID that is treated as a disk. However, the likelihood of this event is greater for larger arrays. We note that, arrays with over 1000 disks are already offered by vendors such as Hewlett Packard, Hitachi and EMC. Such arrays may need to tolerate two or more simultaneous disk failures. For example, RAID Level 10 is obtained in this manner by treating subsets of RAID Level 1 as disks. Figure 2 displays a RAID Level 10 composed of three subsets of RAID Level 1. The blocks in the array are represented by cylindrical segments, with their typical size in practice being 128 KB. Their labels indicate the sequential order in which data is written to them, with identically labeled blocks containing duplicate data.

Although such arrays can tolerate as many disk failures as there are subsets, each failure is restricted to a unique subset. Thus, there is a compelling need for arrays that can tolerate multiple arbitrary disk failures. For most practical array sizes, the probability of occurrence of more than two simultaneous disk failures is generally small enough to be ignored, and a lesser number of failures can be tackled by the use of RAID systems resistant to two disk failures.

## 1.2 A case for attack graphs

The rapid growth of the Internet has triggered an explosion in the number and use of networked applications. Unfortunately, many are maliciously designed to harm network infrastructure and systems. Hence, considerable effort has been focused on detecting and predicting the breaches in security produced by them. However, the enormity of the Internet poses a formidable challenge to representing and analyzing such attacks using scalable models. Furthermore, the unavailability of complete information on network vulnerabilities makes the task of forecasting the systems that are likely to be exploited by such applications in the future equally hard.

Directed graphs serve as an intuitive way to represent network attacks. Existing formulations [35, 37, 39] use vertices to represent a tuple of attributes comprised of a source system, target system, a vulnerability that exists on the target as a precondition, and the postcondition of an atomic attack from the source to the target using that vulnerability. Then, an arc exists from one exploit to another if and only if an atomic attack can leverage the postcondition in the former to utilize the precondition in the latter, and the corresponding systems are connected. To produce such attack graphs, the set of vulnerabilities at each system in the network is first obtained using scanners such as Nessus, Saint, ISS' Internet Scanner and the

CISCO Security Scanner. Then, a model checker such as NuSMV or Spin is applied to a graph on all possible exploits to generate every path that breaches an explicitly stated security condition that is stated as the goal. Finally, the result produced by the model checker is rendered using a visualization application such as GraphViz. However, such attack graphs suffer from two major drawbacks. First, they have very large orders and lack scalability. As a result, they cannot be used to model and visualize attacks on networks in practice. To alleviate the severity of this problem, mechanisms have been proposed to reduce the order of such graphs [3] and to represent them succinctly to facilitate their comprehension by users [27, 29]. Second, a given graph describes the attack paths comprised of sequences of exploits that breach a target security condition, and therefore, its use is confined to the analysis of attacks pertaining to the violation of that condition alone. To address this challenge, attack graphs have been constructed using a large set of attack goals [30].

However, attack graphs merely provide a roadmap of the attacks that can occur based on the vulnerabilities exposed by the systems under consideration. In isolation, they cannot establish if an exploit corresponding to an intrusion detection system (IDS) alert truly constitutes an attack. There are various reasons that can make this determination difficult. Firstly, any given network-based IDS incorporates rules that are designed to search the payload of network packets for signatures that portend threat and generates an alert when a match is found. However, such rules are independent of the potentially differing sets of vulnerabilities admitted by various hosts on that network. That is, the payload of a packet arriving at a system may ostensibly exploit a vulnerability *v*, but that system may not admit *v*. For example, the Lion worm exploits vulnerabilities in Linux only. However, a typical IDS generates an alert even when this worm

attempts to exploit a Windows-based system. However, such an exploit is guaranteed to have no ill effect on a targeted Windows system nor serve as an intermediate step towards harming any peers, and is therefore defined to be irrelevant. Secondly, an event by itself may have ambiguous implications. For example, a TCP-SYN packet received by a host may be a legitimate request by a client to open a streaming connection to that host, or it may constitute a deliberate TCP-SYN flooding being carried out by that client using spoofed IP addresses to achieve denial of service. An alert generated in response to the occurrence of the former event, termed as a false positive, is undesirable since they drain network administration resources away from the investigation of meaningful events. While methods to identify false positives such as TCP-SYN flooding already exist, they are based on time-consuming heuristics that track the activity of malicious clients over periodic intervals of time [38]. Thirdly, an event may be incorrectly judged to be benign. For example, an ftp request that appears to originate from a trusted system within a firewall to a peer within it may be considered harmless, and therefore, ignored by the existing set of IDS rules. However, in reality the actual source may reside outside the firewall, and the system appearing to initiate the ftp request may have been compromised earlier to allow a port-forwarding program on it to masquerade as the source. The lack of an alert in such an instance is termed as a false negative and it poses a far greater danger to the network than false positive or irrelevant alerts. It is therefore vital to correlate alerts to meaningfully identify attacks.

# 2. A SURVEY

## 2.1 Salient methods to realize RAID tolerant to two or more random disk failures

Several schemes to recover from random two-disk failures have been developed, each with their advantages and disadvantages. Blaum's technique for recovery against double disk failures [6] uses bit-wise exclusive-OR (XOR) of data blocks to calculate parity, and stores it using an optimal fraction of the total space of the array. However, it requires the array size to be a prime number. This is commercially unappealing since consumers typically demand considerable flexibility in choosing the size of the array that meets the needs of their respective businesses.

The EVENODD scheme [7] requires the array to have ($p$ + 2) disks, where $p$ is a prime number. Data is stored exclusively on $p$ disks while parity is stored on the remaining two disks. This scheme allows non-prime values for the number of data disks $n$ by assuming the presence of ($p - n$) additional imaginary data disks in the array containing 0's. This scheme too, uses bit-wise XOR of data blocks to calculate parity, and stores it using an optimal fraction of the total space of the array. However, it stores parities exclusively on two disks. This chokes throughput when writing to the array since parity updates are confined to two disks. Furthermore, read throughput from the array does not utilize the cumulative I/O bandwidth of all disks since the two parity disks do not contain any data to be read.

Similarly, the RDP scheme [9] requires the array to have ($p$ + 1) disks, where $p$ is a prime number. Data is stored exclusively on ($p$ – 1) disks while parity is stored on the remaining two disks. As in the EVENODD scheme, RDP allows values for the number of data disks $n$ not equal

to ($p - 1$) by assuming the presence of ($p - 1 - n$) additional imaginary data disks in the array containing 0's. RDP too, uses bit-wise XOR of data blocks to calculate parity, and stores it using an optimal fraction of the total space of the array. Unfortunately, it too, stores parities exclusively on two disks that stifles read and write throughput.

The X-code scheme [43] requires the array to have $p$ disks, where $p$ is a prime number. It distributes parity over all the disks enabling efficient disk utilization. However, its requirement on the number of disks in the array being equal to a prime number makes it commercially undesirable.

The B-code scheme [44] describes the methods for the placement of data and parity in arrays with $p$ and ($2p - 1$) disks by deriving the desired maximum distance separable B-codes from the equivalent perfect 1-factorizations of the complete graphs $K_{p+1}$ and $K_{2p}$.

The STAR code [13] is an extension of the EVENODD scheme that enables arrays having ($p + 3$) disks, where $p$ is a prime number, to recover from up to three random disk failures. As in the case of EVENODD, it places parity exclusively in three disks, and thus suffers from the same I/O inefficiency as its predecessor.

Techniques based on Reed-Solomon codes [36] such as DATUM [2] allow arbitrary array sizes. However, their use of Galois Field arithmetic to compute parities makes them computationally more expensive than those using only XOR operations as demonstrated by projects such as Oceanstore [17]. Hence, they are implemented using costly specialized hardware that makes the solution economically unattractive.

Park's RM2 algorithm [31] determines the configuration of data and parity, when given the desired number of disks $n$ in the array and the ratio of disk space $r$ used for storing parity.

However, the algorithm does not guarantee a solution for all values of *r* and *n*. As a result, for a given value of *n*, several values of *r* may have to be tested, other than the optimal value of 2/*n*, in decreasing order of optimality to find one for which the algorithm can yield the placement of data and parity that can tolerate the failure of two random disks. The only known enhancement to RM2 [21] has unfortunately only addressed the low write I/O performance in arrays tolerant to two disk failures. This shortcoming arises from the need to update two distinct parity blocks whenever a block of data is written into such arrays.

Our research proposes to address the various drawbacks observed in the aforementioned schemes. To that end, our schemes attempt to meet the following requirements.

- Parities are computed using computationally less expensive XOR operations only.
- Parities are distributed evenly over all disks to enhance data throughput.
- Space consumed for storing parity is optimal.
- Arrays can be comprised of an arbitrary number of disks.

### 2.2 Salient methods to analyze and forecast network attacks

A number of methods have been proposed to meaningfully correlate alerts with attacks. One salient approach [40] fuses together alerts that are generated within a finite window of time and possess common values for a specified set of attributes to form a meta-alert. However, the drawback of this technique is that it can fuse together uncorrelated alerts when they have common attributes such as source and destination addresses, and can ignore related alerts that are spaced farther apart in time. Furthermore, it fuses meta-alerts constituting multi-step attacks in a similar manner with the same deficiencies.

Another technique [30] creates paths comprised of observed events such that, each contains events whose corresponding exploits in the attack graph have a distance less than a specific threshold to other exploits whose corresponding events lie on the same path. The relevancy of a path to an attack is then obtained by first applying a moving average filter to the distances between the adjacent events in each path, and then calculating the ratio of the number of events in a given path to the sum of the filtered distances between all pairs of events in that path. The drawback of this scheme is that, it uses conventional non-aggregated attack graphs, which as stated earlier, have extremely large orders, which makes distance calculation between alerts computationally nontrivial.

Several methods have also been proposed to estimate the likelihood of an attack with specified goals. One prominent scheme [10] accomplishes this by modeling attacks using trees where each vertex represents a network/attacker state and each edge represents an exploit that leverages a non-unique vulnerability with a given probability. The probability of the goal state is defined to be the sum of the probabilities of each non-intersecting path that has the vertex representing that goal state as an end, where the probability of each path is the product of the probabilities of the edges comprising it. However, it does not address the mechanism by which the probability of an exploit can be derived.

Another technique [28] computes the closure of the adjacency matrix $A$ of the vertices of an attack graph, where $A^k$ shows the clusters of systems that are at risk from one another after $k$ attack steps. However, this indicates the reachability of an attack initiated from a given system to others in the network and not necessarily the path that is taken by an attack.

# 3. MODELING RAID TO TOLERATE TWO RANDOM DISK FAILURES

The graph theoretic schemes we have developed to model the placement of data and parity in disk arrays have several advantages over the schemes investigated in our survey. First, the computational expense incurred by our schemes for deriving parities and reconstructing failed disks is relatively low. This is because they use XOR operations only to calculate parity. Second, they distribute parity uniformly over all the disks, permitting data to be read from, and parity to be written to all disks concurrently. This enhances I/O throughput of the array. Finally, the arrays to which our schemes can collectively be applied have disk numbers encompassing a relatively dense subset of the set of integers, while consuming an optimal ratio of space for storing parity. In each scheme, the fault-tolerant array is modeled as a graph $G = (V, E)$ with a self-loop at each vertex, where each disk corresponds to a unique vertex in $V$, each data block corresponds to a simple edge in $E$, and each parity block corresponds to a self-loop.

## 3.1 Approach using complete graphs

Two graph-theoretic algorithms for arrays of $n$ disks, $n \geq 4$, that can recover from the failure of any two arbitrary disks have been proposed [11, 12]. The first has $n = p - 1$, and the second $n = 2p - 2$, where $p$ is a prime number. Both algorithms generate solutions with at most $(n/2 - 1)$ equal sized blocks of data and exactly one block of parity per disk. Each parity block's value is then obtained by computing the bit-wise XOR of data blocks specified by the respective algorithm. In practice, the solutions repeat the configuration of the data and parity blocks an integral number of times.

### 3.1.1 *Using $K_p$, where p is a prime number, to model arrays with (p – 1) disks*

Given $p$ a prime number, this scheme admits arrays with $(p - 1)$ disks having $(p - 3)/2$ equal sized blocks of data and one block of parity per disk. Each data block is labeled $\{u, v\}$, $0 \leq u, v \leq p - 1$, $u \neq v$, and its contents used for computing the bit-wise parity stored in the parity blocks labeled $\{u, u\}$ and $\{v, v\}$. Thus, the parity in the block labeled $\{v, v\}$ equals the bit-wise XOR of the data in each data block labeled using $v$. Also, the parity-block in disk $d$ is labeled $\{d, d\}$. For brevity, a block labeled $\{u, v\}$ is hereafter referred to as, block $\{u, v\}$.


Figure 3: A 6-disk array modeled using $K_7$.

Consider the array of six disks shown in Figure 3. In this array, each parity-block $\{d, d\}$ contains the bit-wise XOR of all data-blocks labeled using $d$, where $0 \leq d \leq p - 2$. For instance, the parity in the block {3, 3} equals the bit-wise XOR of the data in blocks {3, 4}, {1, 3}, {3, 5} and {0, 3}. This array can tolerate two random disk failures. For example, if disks 0 and 2 say fail, then we can reconstruct them as follows. We can first reconstruct the block {1, 3} on disk 2 because no other blocks labeled using 1 are on the failed disks. Therefore, its content can be derived by taking the bit-wise XOR of the blocks {1, 1}, {1, 5}, {0, 1} and {1, 2}. Next, we can reconstruct the block {3, 4} on disk 0 because all other blocks labeled using 3 have been already reconstructed or lie on disks that are intact. Similarly, we can then reconstruct the block {0, 4}

followed by the block {0, 0}. In the aforesaid sequence of reconstructed blocks {1, 3}, {3, 4}, {0, 4} and {0, 0}, each block's label, other than that of the first, has an element in common with the label of the previously reconstructed block. However, we cannot reconstruct any farther in this manner because no unreconstructed blocks labeled using 0 remain on the failed disks. Next, we can reconstruct the remaining blocks on disks 0 and 2 as follows. We reconstruct the block {2, 5} because no other blocks labeled using 5 are on the failed disks. Hence, its content can be derived from the bit-wise XOR of the blocks {4, 5}, {1, 5}, {3, 5} and {5, 5}. Finally, we can reconstruct the block {2, 2}.

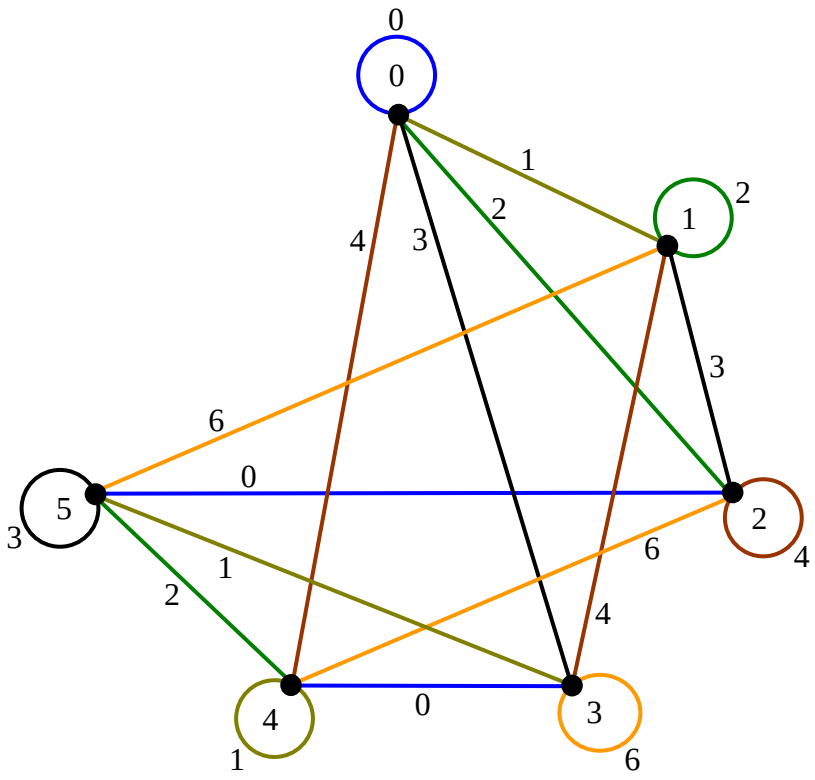


Figure 4: The graph $G(V, E)$ that models the six-disk RAID.

The array shown in Figure 3 is modeled by the graph $G(V, E)$ shown in Figure 4. Each data-block $\{u, v\}$ in the array corresponds to the edge $(u, v)$ in $E$. Thus, each parity-block $\{v, v\}$ corresponds to the self-loop $(v, v)$. Note that, all the edges and self-loops in $E$ are colored, with the blocks corresponding to identically colored edges being on the same disk. For example, the blocks in the array corresponding to edges (3, 4), (2, 5) and (0, 0), each colored 0, are all on the leftmost disk.

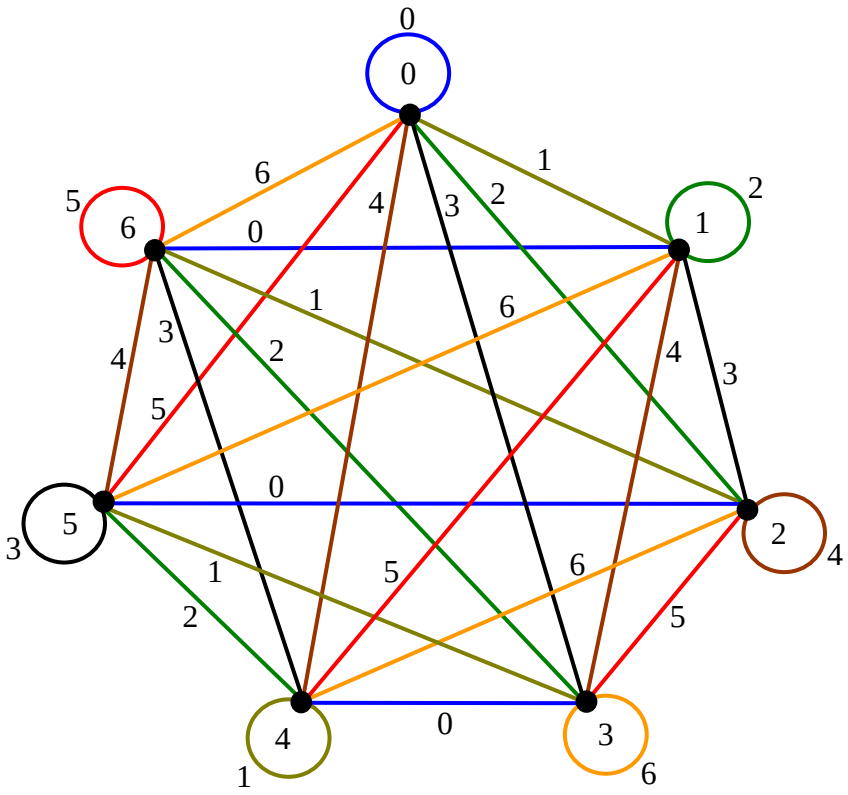


Figure 5: The graph $Q_7$ with its edges colored to yield a perfect near-1-factorization.

$G$ is obtained from the complete graph $K_7$ in the following manner. First, a self-loop is added to each vertex of $K_7$ to yield the graph $Q_7$. Next, the edges of $Q_7$ are colored to produce a

perfect near-1-factorization. A *near*-1-*factor* of a graph $G$ is a spanning subgraph of $G$ in which one vertex is isolated and all others have degree 1. (Note that the order of $G$ must then be odd.) A *perfect near*-1-*factorization* of $G$ is a partitioning of its edges into near-1-factors such that the union of any two near-1-factors induces a Hamiltonian path in $G$. Now, $Q_7$ admits such a coloring, as shown in Figure 5, since any complete graph of prime order admits a perfect near-1-factorization [4]. One such coloring is where edge $(u, v)$ is colored $u + v \pmod{p}$. Let $\mathcal{O}$ be the set of seven near-1-factors and their associated self-loops in $Q_7$. Finally, from $Q_7$ we remove vertex 6, the edges incident on vertex 6, and all the edges of the near-1-factor colored the same as the self-loop on vertex 6.

The result of removing vertex 6 and the edges incident on it from $Q_7$ results in each of the remaining six near-1-factors in $\mathcal{O}$ losing their respective edge incident on vertex 6 to yield a set of six matchings $\mathcal{M}$. Each matching corresponds to a disk, each edge $(u, v)$ in a matching represents the data block $\{u, v\}$ in the corresponding disk, and each self-loop $(v, v)$ associated with a matching, represents the parity-block $\{v, v\}$ in the disk corresponding to that matching.

Now, suppose two disks have failed in the array. Then, we can reconstruct the data blocks in those disks as follows. Select an edge $(u, v)$ in a near-1-factor in $\mathcal{O}$ corresponding to a failed disk such that $u$ is adjacent to the vertex in $W$ in the near-1-factor corresponding to the other failed disk. Then, no edge other than $(u, v)$ in the two matchings in $\mathcal{M}$ corresponding to the failed disks can be incident to the vertex $u$. Hence, there is no data block other than $\{u, v\}$ in the two failed disks that contributes to the contents of the parity-block $\{u, u\}$. Furthermore, the parity-block $\{u, u\}$ cannot be on either failed disk for the following reason. Parity-blocks correspond to self-loops, and self-loops are on the ends of the Hamiltonian path formed by the edges in the

union of any pair of near-1-factors in $\mathcal{G}$. If the parity-block $\{u, u\}$ were on a failed disk, $u$ would be an end of the Hamiltonian path formed by the edges in the near-1-factors in $\mathcal{G}$ corresponding to the failed disks. This would be a contradiction, since by assumption $(u, v)$ is an edge.

Thus, we can reconstruct the data block $\{u, v\}$ by taking the XOR of the data in all blocks labeled using $u$. We can then reconstruct a data block labeled using $v$, $\{v, w\}$ say, by taking the XOR of the data in all blocks labeled using $v$. This is possible because a pair of disks can have at most two data blocks that contribute to the contents of any given parity-block. Next, we can reconstruct a data block labeled using $w$ different from $\{v, w\}$. Thus, in each iteration we select a data block for reconstruction whose label shares a common element with the label of the data block reconstructed in the previous iteration. Thus, a data block $\{u, v\}$ can be reconstructed if the edge $(u, v)$ is on a path comprised of the edges in the matchings in $\mathcal{M}'$ corresponding to the failed disks, and the path has an end $x$, with $x$ adjacent to vertex 6 in the corresponding near-1-factor in $\mathcal{G}$. Since the edges corresponding to the data-blocks of two failed disks form a Hamiltonian path minus those edges incident to vertex 6, each edge is therefore on a path having an end that is adjacent to vertex 6. Hence, we can reconstruct each and every data block on a pair of failed disks.

Since a graph with prime order admits a perfect near-1-factorization, the steps involved in the modeling of the six disk RAID can be generalized using the complete graph on $p$ vertices $K_p$, to yield a $(p - 1)$ disk RAID, where $p$ is a prime number. Figure 6 illustrates the algorithm that may be used to this end, while Figure 26 in Appendix A describes the algorithm for reconstructing a pair of failed disks in such an array.

```
for u ← 0 to (P – 1) do
        for v ← u to (P – 1) do
                Color (u, v) with c(u, v) ← (u + v) mod p
        endfor
endfor

Let W be a singleton subset of V

for each disk d in V – W do
        b ← 0
        for u ← 0 to (p – 2) do
                for v ← (u + 1) to (p – 1) do
                        if (u ∈ V – W) and (v ∈ V – W) and (c(u, v) = c(d, d)) then
                                Block b in disk d is assigned the data-block label {u, v}
                                b ← b + 1
                        endif
                endfor
        endfor
        Block b of disk d is assigned the parity-block label {d, d}
endfor
```

Figure 6: Algorithm to place data and parity blocks in a ($p$ – 1) disk array

Data may be requested from arrays with a pair of failed disks that have not been reconstructed. To minimize the number of blocks reconstructed prior to the one containing the desired data, it is necessary to find the sequence of blocks containing that block. There are only two sequences of data blocks comprising a pair of disks, with each sequence starting on a unique disk. Hence it suffices to find the disk on which the desired reconstruction sequence starts. To this end, Figure 28 in Appendix B describes the algorithm to find the disk from which to start reconstruction.

### 3.1.2 *Using $K_{2p-1}$ to model arrays with (2p – 2) disks*

The preceding result demonstrates how any complete graph with a self-loop at each vertex models the placement of data and parity in a two-disk fault tolerant array if it admits a perfect near-1-factorization. This is stated by the following theorem.

**Theorem 1.** Let $Q_m$ be the graph formed by adding a self-loop at each vertex of $K_m$. If $K_m$ admits a perfect near-1-factorization, then $Q_m$ models the placement of data and parity in an array of ($m$ – 1) disks to tolerate two random disk failures.

Perfect 1-factorizations of $K_{2p}$ have been given by Anderson [4] and Nakamura in Japanese [15]. Nakamura defines a 1-factor of $K_{2p} = (V, E)$ where $V = \{0, 1, \ldots, 2p - 1\}$ with edges colored $c$ as,

$$F_c = \begin{cases} \{(i, j) \mid i + j \equiv c \pmod{2p}\} \cup \{(\frac{c}{2}, \frac{c}{2} + p)\}, & \text{if } c \text{ is even} \\ \{(i, j) \mid i \text{ is odd and } i - j \equiv c \pmod{2p}\}, & \text{if } c \text{ is odd and } c \neq p \end{cases}$$

The (2$p$ – 1) 1-factors defined by this construction have 2$p$ edges each. Now, we remove vertex (2$p$ – 1) and all edges incident to it from $K_{2p}$ to obtain (2$p$ – 1) near-1-factors of $K_{2p-1}$, each having (2$p$ – 1) edges. The union of the edges of any pair of these near-1-factors must form a Hamiltonian path. Next, we add a self-loop at each vertex of $K_{2p-1}$ to obtain $Q_{2p-1}$. By Theorem 1, $Q_{2p-1}$ must model the placement of data and parity in an array of (2$p$ – 2) disks to tolerate two disk failures using an optimal ratio of space for parity. Using $Q_{2p-1}$, we place data and parity in an array of (2$p$ – 2) disks as described by the algorithm in Figure 7.

```
Step 1:  for u ← 0 to (2p – 3) do
             for v ← (u + 1) to (2p – 2) do
                 if (u + p = v) then
                     Color (u, v) with c(u, v) ← 2u (mod 2p)
                 else if (u + v ≡ 0 (mod 2)) then
                     Color (u, v) with c(u, v) ← u + v (mod 2p)
                 else if (u ≡ 1 (mod 2)) then
                     Color (u, v) with c(u, v) ← u – v (mod 2p)
                 else
                     Color (u, v) with c(u, v) ← v – u (mod 2p)
                 endif
             end for
         end for

Step 2:  for u ← 0 to (2p – 2) do
             S ← {0, …, p – 1, p + 1, …, 2p – 1}
             for v ← 0 to (2p – 2) do
                 if ( u ≠ v ) then
                     S ← S – c(u, v)
                 end if
             end for
             k ← 0
             while (k ≤ (2p – 2) and k ∉ S) do
                 k ← k + 1
             end while
             Color (u, u) with c(u, u) ← k
         end for

Step 3:  Let W be a singleton subset of V

Step 4:  for each disk d in V – W do
             b ← 0
             for u ← 0 to (2p – 3) do
                 for v ← (u + 1) to (2p – 2) do
                     if u ∈ V – W and v ∈ V – W and c(u, v) = c(d, d) then
                         Label data-block b in d as {u, v}
                         b ← b + 1
                     end if
                 end for
             end for
             Label parity-block b in d as {d, d}
         end for
```

Figure 7: Algorithm to model a $(2p – 2)$ disk array that tolerates two disk failures.

As an example, the eight-disk array with data and parity assignments obtained upon executing the given algorithm using $W = \{8\}$ is shown in Figure 8.

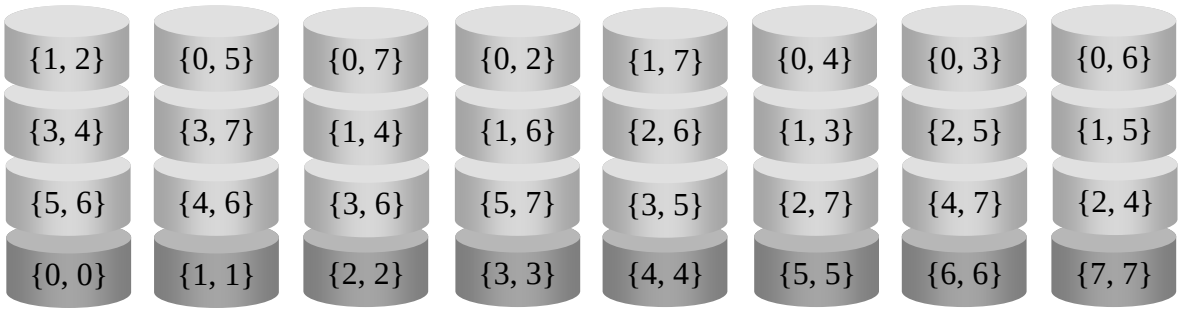


Figure 8: An eight-disk array modeled using $Q_9$

### 3.1.3 *Additional values of n for which $K_n$ models RAID*

In addition to the infinite families of graphs on $p$ and $(2p - 1)$ vertices, there are graphs on $n$ vertices that admit perfect near-1-factorizations, where $n$ equals, 15, 27, 35, 39, 49, 169, 243, 343, 729, 1331, 1369, 1849, 2197, 3125, and 6859 [41]. It is therefore possible to model arrays with $n$ disks that can tolerate two random disk failures for each of the aforementioned values of $n$ [25]. An important observation in this context is that, the existence of 1-factorizations for every value of $2n$, $n \geq 2$, as conjectured [16], would enable us to model arrays with any desired even number of disks.

### 3.1.4 *Extending the model to arrays of arbitrary size*

We note that the union of any two matchings in $\mathcal{M}$ is obtained by removing the edges incident to the vertex in $W$ from the Hamiltonian path formed by the union of the corresponding near-1-factors in $\mathcal{O}$. If $W$ were to have two or more vertices, the union of any two matchings in $\mathcal{M}$

would be obtained by removing the edges incident to all the vertices in $W$ from the Hamiltonian path formed by the union of the corresponding near-1-factors in $\mathcal{O}$. It is then clear that, the union of any two matchings in $\mathcal{T}$ would still yield paths, each with a self-loop on at most one end. The disks corresponding to a pair of such matchings can then be reconstructed in the manner described for the six-disk RAID. We can therefore extend the algorithm in Figure 7 to create arrays of $n$ disks to tolerate two disk failures by choosing $W$ such that, $|W| = p - n$ and $|W| = 2p - n - 1$ respectively, for $n \geq 3$. We illustrate an array with $n = 8$ disks using $p = 11$ in Figure 9 created in this manner.

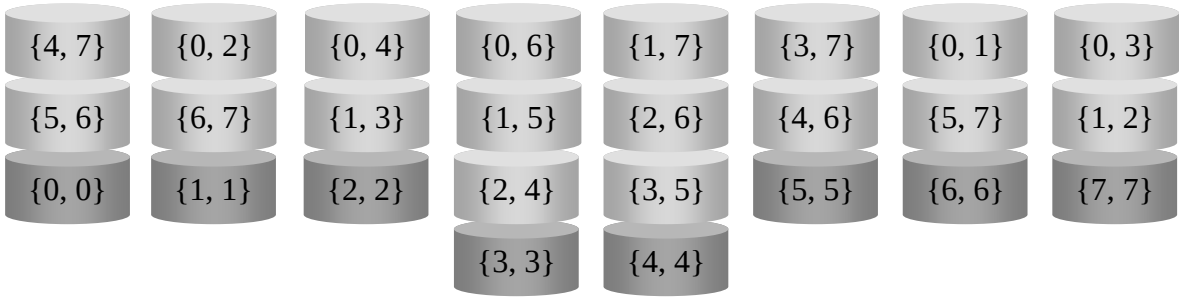


Figure 9: Algorithm to create an eight-disk array with $V - W = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

We derive the ratio of space used for parity in such an array with $n$ disks by first determining the number of data blocks in it. Earlier, we described the manner in which we delete edges from the graph $Q_m$ to obtain a 2-disk fault-tolerant array with $n = m - k$ disks, where $m = p$ or $m = 2p - 1$, and $K_m$ admits a perfect near-1-factorization. We noted that the number of data blocks in such an array equals the number of remaining edges. We now determine that number by iteratively removing a vertex $v$ and all the edges incident to it, followed by the edges in the matching that are colored the same as the self-loop at $v$.

Suppose that in the first iteration we remove vertex $u$ from $Q_m$, the $(m-1)$ edges incident to $u$ and the ½$(m-1)$ edges colored the same as the self-loop at $u$. The remaining vertices are each of degree $(m-3)$. Suppose that in the second iteration we remove vertex $v$ from $Q_m$, the $(m-3)$ edges incident to $v$ and the $\lfloor \frac{1}{2}(m-2) \rfloor$ edges colored the same as the self-loop at $v$. Since $m$ is odd, $\lfloor \frac{1}{2}(m-2) \rfloor = \frac{1}{2}(m-3)$. At this point, one vertex is of degree $(m-4)$ and the remaining ones are of degree $(m-5)$ for the reason that, one vertex does not have an edge incident to it that is colored the same as the self-loop at $v$, and therefore must have degree $(m-4)$. Now, to ensure that we delete the least number of edges in the next iteration, we choose for deletion the vertex $w$ with the least degree i.e., $(m-5)$ in addition to its $(m-5)$ incident edges, and the $\frac{1}{2}(m-3)$ edges colored the same as the self-loop at $w$. Proceeding in this manner, the maximum number of edges removed in each iteration is as follows.

Iteration 1: $(m-1)+\frac{1}{2}(m-1)$
Iteration 2: $(m-3)+\frac{1}{2}(m-3)$
Iteration 3: $(m-5)+\frac{1}{2}(m-3)$
Iteration 4: $(m-7)+\frac{1}{2}(m-5)$
Iteration 5: $(m-9)+\frac{1}{2}(m-5)$
: :
Iteration $i$: $(m-2i+1)+\frac{1}{2}(m-\frac{1}{2}(2i+1+(-1)^i))$

Then, the total number of edges removed up to and including iteration $k$ is:

$$\leq \sum_{i=1}^{k}(m-2i+1)+\tfrac{1}{2}(m-\tfrac{1}{2}(2i+1+(-1)^i))$$
$$= \tfrac{1}{4}[k(6m-5k-2)+k \bmod 2]$$

The number of edges remaining at the end of $k$ iterations is then:

$$\geq \tfrac{1}{2}m(m-1) - \tfrac{1}{4}[k(6m-5k-2) + k \bmod 2]$$

The ratio of space used for parity in the array is then

$$\leq \frac{(m-k)}{\tfrac{1}{2}m(m-1) - \tfrac{1}{4}[k(6m-5k-2) + k \bmod 2] + (m-k)}$$

It is easy to verify that this ratio has the optimal value of $2/(m-1)$ when $n = m - 1$ by observing that the optimal ratio of space used for storing parity in an $n$ disk array equals $2/n$.

Table 1 shows the number of arrays with $(p - 1)$ and $(2p - 2)$ disks in the second and third columns respectively for the corresponding interval of $n$ in the first column. The last column shows the percentage of even values of $n$ covered by their sum in the corresponding interval.

Table 1: Even numbers admitted by $(p - 1)$ and $(2p - 2)$.

| **Interval of *n*** | **Number of arrays with (*p* – 1) disks** | **Number of arrays with (2*p* – 2) disks** | **Percentage of even *n* covered** |
|---|---|---|---|
| 4 – 128 | 30 | 12 | 66.66 |
| 130 - 256 | 20 | 14 | 53.13 |
| 258 – 512 | 43 | 18 | 47.65 |
| 514 – 1024 | 76 | 34 | 42.96 |
| 1026 – 2048 | 138 | 60 | 38.67 |

Figure 10 displays the percentage difference in the ratio of space used for parity by the algorithm compared to the optimal value for array sizes between 5 and 255. Note that, prime numbers occur more sparsely in intervals spanned by larger integers, and therefore, the frequency at which the algorithm uses optimal values decreases as the size of the arrays increases.

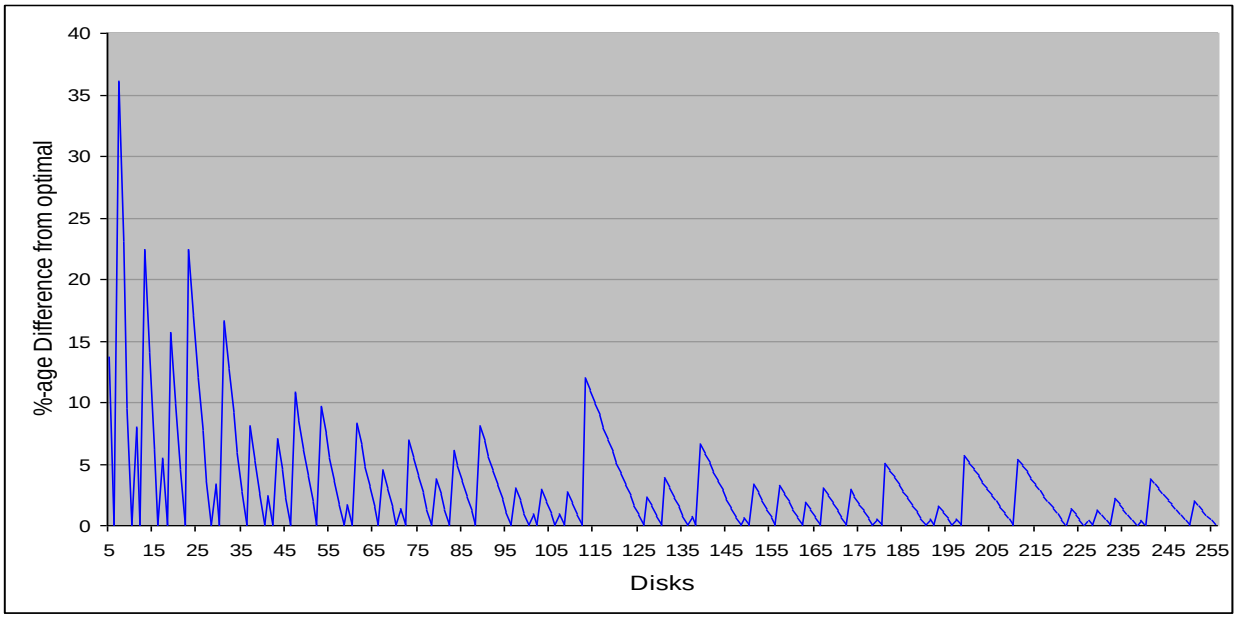


Figure 10. Difference in the ratio of space used for parity by the algorithm to the optimal value.

## 3.2 Approach using complete bipartite graphs

Algorithms have been proposed to model data and parity placement in arrays with $(n - 1)$ disks to tolerate two disk failures using complete bipartite graphs on $2n$ vertices, for $n = p$ and $n = 2p - 1$, where $p$ is a prime number [24].

### 3.2.1 *Using $K_{p, p}$ to model $(p - 1)$ disk arrays*

As shown in the preceding algorithms, a two-disk fault-tolerant array is modeled by a graph satisfying the following condition. The blocks in a disk correspond to the edges of a matching in the graph, and the blocks in any pair of disks correspond to paths in the graph, each having an end of degree 1. Such graphs can be derived from complete bipartite graphs with $2p$ vertices as follows.

Let $Q_{p, p}$ be the complete bipartite graph on $2p$ vertices, $p$ a prime, with a self-loop at each vertex. Suppose that each edge $(u, v)$ in it is colored $u + v \pmod{p}$. Then, $Q_{p, p}$ can be factored into $p$ 1-factors such that the simple edges in the union of any two 1-factors form a Hamiltonian cycle. This may be proved as follows.

Let $Q_{p, p} = (V, E)$ where $V_1 = \{0, 1, \ldots, p - 1\}$ and $V_2 = \{p, p + 1, \ldots, 2p - 1\}$ are the bipartition subsets of $V$. Color each edge $(u, v)$ of $Q_{p, p}$ with $u + v \pmod{p}$ (including the self-loop when $u = v$). Then, we shall show that, (a) the graph on the vertices of $Q_{p, p}$ formed by its proper edges with a common color is a 1-factor of $Q_{p, p}$, and (b) the proper edges in the union of any two 1-factors contains a Hamiltonian cycle. For the sake of brevity, we shall henceforth refer to proper edges simply as edges and to self-loops explicitly.

(a) First, we show that there are $p$ independent edges with a common color by proving that no two edges having a common color can be adjacent. Suppose to the contrary that, there exist a pair of adjacent edges $(u, v)$ and $(v, w)$ both colored $e$. Then, we must have $e \equiv u + v \pmod{p}$ and $e \equiv v + w \pmod{p}$. That is, $u + v \equiv v + w \pmod{p}$ and therefore, $u \equiv w \pmod{p}$. Since $u$ and $w$ are both adjacent to $v$, both belong to the same bipartition subset, and therefore $u = w$. A contradiction. Now, if there are less than $p$ edges colored $e_1$ say, then there must be more than $p$ edges colored $e_2$, where $e_1 \neq e_2$ and $0 \leq e_1, e_2 \leq p - 1$. This is a result of the pigeonhole principle, for there are $p^2$ edges in $Q_{p,\,p}$ and $p$ colors. Then, two edges colored $e_2$ must be adjacent. A contradiction. Hence, there must be exactly $p$ independent edges having any common color, and therefore, the graph on $V$ formed by the edges with a common color is a 1-factor of $Q_{p,\,p}$.

(b) Next, we show that the union of any two 1-factors forms a Hamiltonian cycle. Suppose to the contrary, there exists a cycle $v_1, v_2, v_3, \ldots v_t, v_1$ in the union of two 1-factors with edges alternately colored $e_1$ and $e_2$ where $t < 2p$. Without loss of generality, assume that the edges $(v_1, v_2), (v_3, v_4), \ldots (v_{t-1}, v_t)$ are colored $e_1$, and $(v_2, v_3), (v_4, v_5), \ldots (v_t, v_1)$ are colored $e_2$. Note that $t$ must be even. Therefore, let $t = 2k$, $k < p$. Then,

$$e_1 \equiv (v_1 + v_2) \,(mod\ p), \qquad e_2 \equiv (v_2 + v_3) \,(mod\ p),$$

$$e_1 \equiv (v_3 + v_4) \,(mod\ p), \qquad e_2 \equiv (v_4 + v_5) \,(mod\ p),$$

$$\vdots \qquad\qquad\qquad \vdots$$

$$e_1 \equiv (v_{t-1} + v_t) \,(mod\ p), \qquad e_2 \equiv (v_t + v_1) \,(mod\ p)$$

We can then easily prove that $t(e_1 - e_2)/2 \equiv 0 \,(mod\ p)$ and therefore, $k(e_1 - e_2) \equiv 0 \,(mod\ p)$. This is a contradiction since $p$ is a prime, and $k$, $e_1$ and $e_2$ are each less than $p$.

From (a) and (b) we see that $Q_{p,\,p}$ admits a factorization into $p$ 1-factors such the union of any pair has $2p$ edges without a cycle having less than $2p$ edges. Hence the union of any two 1-factors must form a Hamiltonian cycle. Figure 11 illustrates the Hamiltonian path formed by the 1-factors having edges colored 0 and 1.

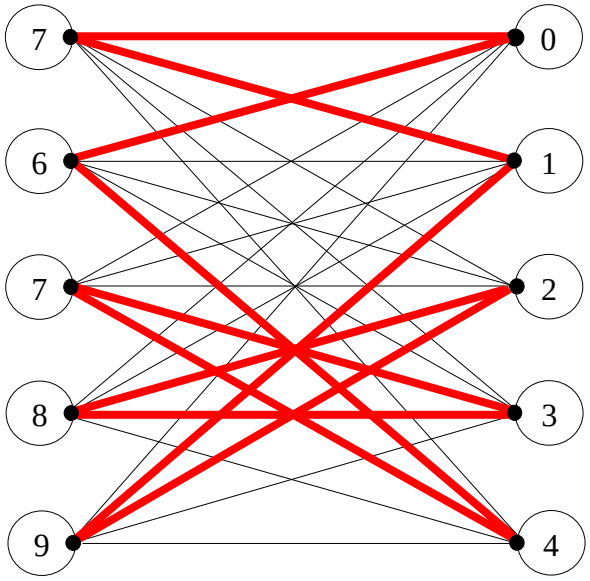


Figure 11: $Q_{5,\,5}$ and the Hamiltonian cycle formed by the 1-factors colored 0 and 1.

Given that $Q_{p,\,p}$ admits a perfect 1-factorization, we now remove edges from it such that the union of the remaining edges in any pair of its 1-factors forms paths, each having a self-loop on at most one end. Now, each Hamiltonian cycle formed by the union of a pair of 1-factors of $Q_{p,\,p}$ is of the form shown in Figure 12. The distance between vertices $u$ and $v$ on such a cycle with edges alternately colored $2v \pmod p$ and $2u \pmod p$ is $p - 1$. To show this, let us suppose that $\{u, w_1, w_2, \ldots w_{k-1}, v\}$ is a path of length $k$. Then,

$$u + w_1 \equiv 2v \pmod{p}, \qquad w_1 + w_2 \equiv 2u \pmod{p}$$

$$w_2 + w_3 \equiv 2v \pmod{p}, \qquad w_3 + w_4 \equiv 2u \pmod{p}$$

$$\vdots \qquad\qquad \vdots$$

$$w_{k-2} + w_{k-1} \equiv 2v \pmod{p}, \quad w_{k-1} + v \equiv 2u \pmod{p}$$

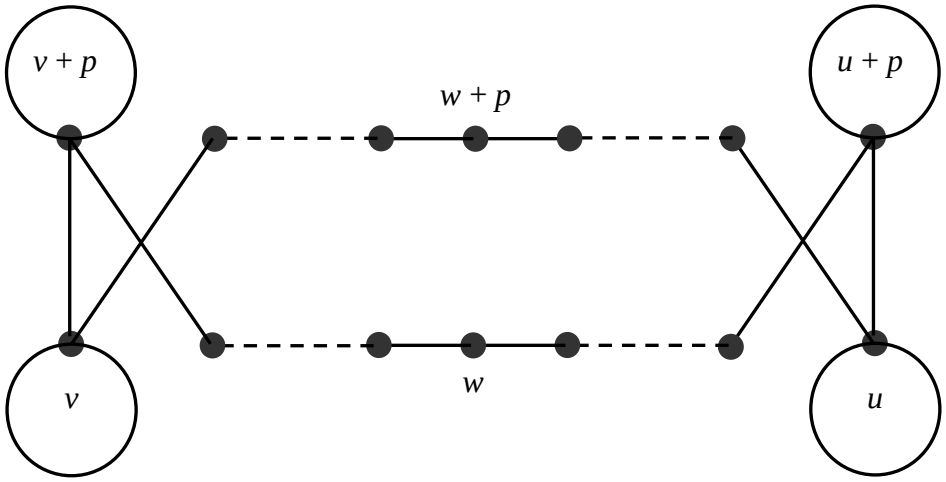


Figure 12: A Hamiltonian cycle formed by a pair of 1-factors of $Q_{p,p}$.

Then, it can be proved that $(k + 1)(v - u) \equiv 0 \pmod{p}$, which implies that $k = p - 1$. In a similar manner we can show that the distance between vertices $(u + p)$ and $(v + p)$ is also $p - 1$. Using this observation, we remove each edge $(u, u + p)$ in $Q_{p,p}$, $0 \le u \le p - 1$. Then the union of the remaining edges in any pair of 1-factors yields a pair of paths of length $p - 1$, each having a self-loop at both ends. Now, for any given vertex $w$ on one of these paths, where $0 \le w \le p - 2$, vertex $w + p$ must lie on the other path and vice versa. This is observed by enumerating the vertices in the paths starting with their corresponding ends, $u$ and $u + p$ (say). Thus, if $w$ is adjacent to $x + p$, with the edge $(x + p, w)$ colored $a$ in one path, then $w + p$ must be adjacent to $x$

with the edge ($x$, $w + p$) colored $a$ in the other. Using this observation, we next remove the vertices $w$ and $w + p$ from $Q_{p, p}$, for $w \in \{0, 1, \ldots p - 1\}$, and all the edges incident to them. Finally we remove all edges colored $2w$ (mod $p$), which leads to all the edges of that 1-factor being deleted. As a result, the union of the remaining edges in any pair of the other ($p - 1$) 1-factors must yield paths having a self-loop on at most one end as shown in Figure 13.

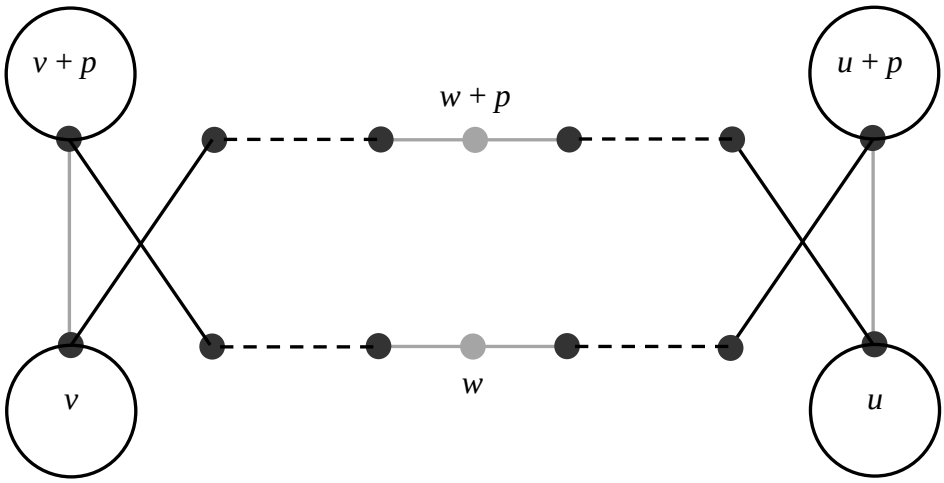


Figure 13: Paths obtained by deleting the lighter colored edges from two 1-factors of $Q_{p, p}$.

We now have the graph derived from $Q_{p, p}$ that models the placement of data and parity in an array to tolerate two disk failures based on the characteristics of such a graph stated at the beginning of this section. Figure 15 illustrates a four-disk array on the right modeled using the graph in Figure 14 that is derived in the aforementioned manner from $Q_{5, 5}$. In this array, disk $d$ contains the block $\{u, v\}$ when the corresponding edge ($u$, $v$) in the graph is colored $d$, where $1 \leq d \leq 4$. Now, if for example disks 1 and 2 have failed in this array, their blocks may be reconstructed in the order {7, 4}, {4, 8}, {8, 8}, followed by {2, 9}, {9, 3}, {3, 3}, followed by

{1, 1} and {6, 6}. Note that, each of these sequences corresponds to paths with a self-loop on exactly one end that are formed by the edges in the pair of 1-factors of $Q_{5,5}$ colored 1 and 2.

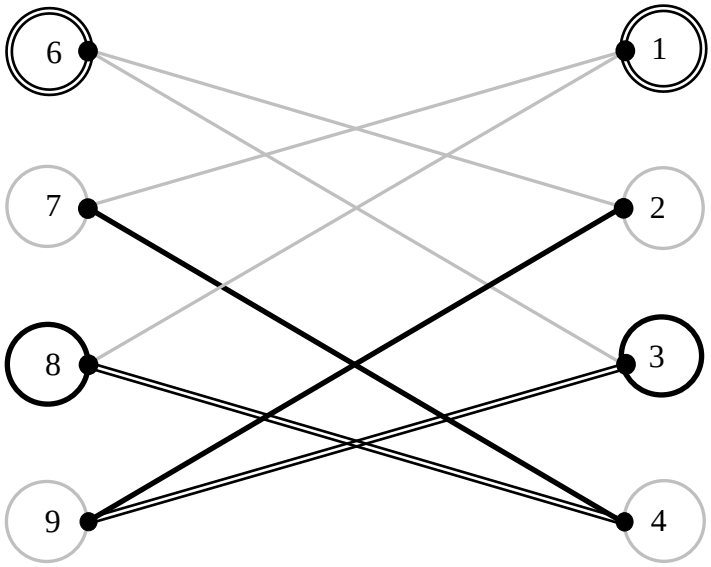


Figure 14: The graph derived from $Q_{5,5}$.

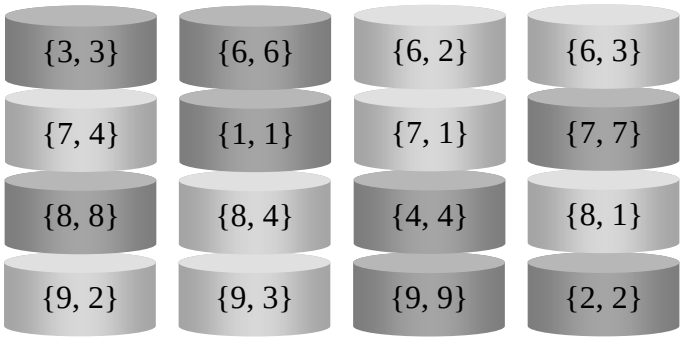


Figure 15: Four-disk array modeled from the graph derived from $Q_{5,5}$.

### 3.2.2 *Using $K_{2p-1,\,2p-1}$ to model $(2p-2)$ disk arrays*

Let $Q_{2p-1,\,2p-1}$ be the complete bipartite graph on $4p-2$ vertices, $p$ a prime, with a self-loop at each vertex. The placement of data and parity in an array with $2p-2$ disks to tolerate two disk failures using $Q_{2p-1,\,2p-1}$ can be modeled in a manner similar to $Q_{p-1,\,p-1}$. We now describe the method for obtaining a perfect 1-factorization of $Q_{2p-1,\,2p-1}$ from the perfect 1-factorization of $K_{2p}$ for which constructions have been proposed by Anderson [4] and Kobayashi [15]. Let $K_{2p} = (V, E)$ where $V = \{0, 1, \ldots, 2p-1\}$. Then, Kobayashi's construction defines a 1-factor of $K_{2p}$ having edges colored $e$ as,

$$F_e = \begin{cases} \{(i,j) \mid i + j \equiv e\} \cup \{(\frac{e}{2}, \frac{e}{2} + p)\} \; when \; e \; is \; even \\ \{(i,j) \mid i \; is \; odd, and \; i - j \equiv e \pmod{2p}\} \; when \; e \; is \; odd \; and \; e \neq p. \end{cases}$$

This defines $(2p-1)$ 1-factors for $K_{2p}$, where $0 \leq e \leq 2p-1$ and $e \neq p$, such that the union of the edges of any pair forms a Hamiltonian cycle. Using the 1-factors of $K_{2p}$, the edges of $K_{2p-1,\,2p-1}$ are colored to create a perfect 1-factorizarion using a method proposed by Laufer [18]. It is as follows. Let $K_{2p-1,\,2p-1} = (V, E)$ where $V_1 = \{0, 1, \ldots, 2p-2\}$ and $V_2 = \{2p-1, 2p, \ldots, 4p-3\}$ are the bipartition subsets of $V$. Let $c(u, v)$ be the color assigned to an edge $(u, v)$ in $K_{2p-1,\,2p-1}$, and $e(w, x)$ the color assigned to an edge $(w, x)$ in $K_{2p}$. Then, for $0 \leq u, v \leq 2p-2$,

$$c(u, v+2p-1) = c(v, u+2p-1) = \begin{cases} e(u+1, v+1), if \; u \neq v \\ e(0, v+1), if \; u = v \end{cases}$$

The proper edges of $Q_{2p-1,\,2p-1}$ are colored the same as the edges of $K_{2p-1,\,2p-1}$, and each self-loop $(u, u)$ on the vertices of $Q_{2p-1,\,2p-1}$ is colored as,

$$c(u,u) = u,\ for 0 \le u \le 2p-2$$

Each Hamiltonian cycle constructed in this manner is described by a sequence of vertices $\langle u, u + 2p - 1, \ldots, v + 2p - 1, v, \ldots, u\rangle$, where $0 \le u, v \le 2p - 2$. In such a cycle, the shortest paths between vertices $u$ and $v$, and between vertices $u + 2p - 1$ and $v + 2p - 1$ are of length $2p - 2$. These paths correspond to the subpath that is obtained from the Hamiltonian cycle in $K_{2p}$ defined by the sequence of vertices $\langle u + 1, 0, v + 1, \ldots, u + 1\rangle$, by removing vertex 0 and the edges $(0, u + 1)$ and $(0, v + 1)$.

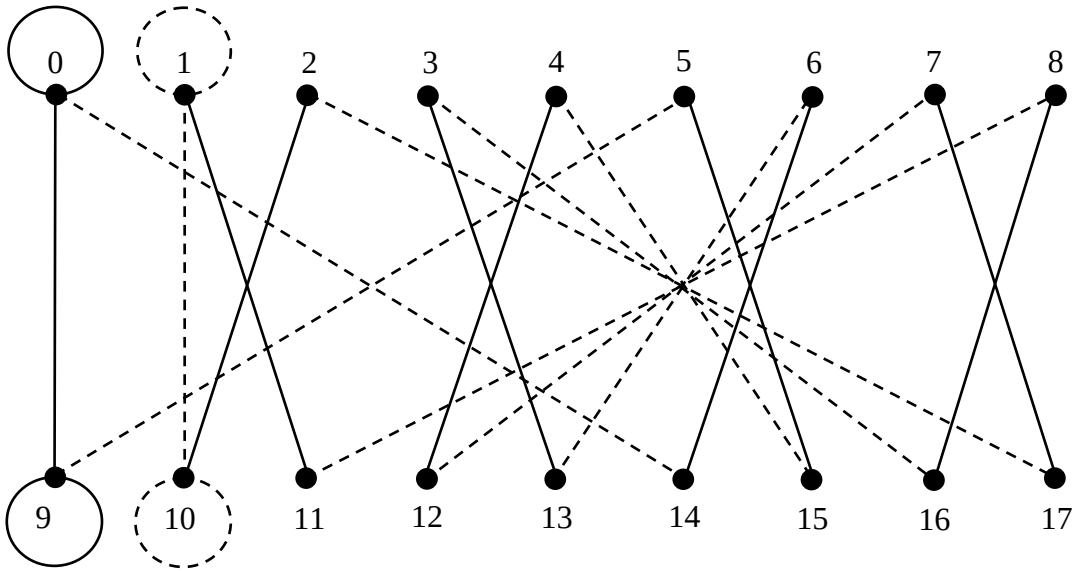


Figure 16: The Hamiltonian path formed by edges colored 0 and 1 in $Q_{9,9}$.

Figure 16 illustrates the 1-factors formed by the edges colored 1 and 2 in $Q_{9,9}$ and the Hamiltonian cycle formed by their union using the aforementioned coloring scheme. The solid lines indicate the edges colored 1, and the dotted lines indicate the edges colored 2. We observe

that the paths defined by the sequences of vertices ⟨0, 14, 6, 13, 3, 16, 8, 11, 1⟩ and ⟨9, 5, 15, 4, 12, 7, 17, 2, 10⟩ correspond to the subpath that is obtained from the Hamiltonian cycle in $K_{10}$ defined by the sequence of vertices ⟨1, 0, 2, …, 1⟩, by removing vertex 0 and the edges (0, 1) and (0, 2).

Now remove edges from $Q_{2p-1,\,2p-1}$ in the same manner as that described earlier for $Q_{p,\,p}$ such that the union of the remaining edges in any pair of its 1-factors forms paths, each having a self-loop on at most one end. To this end, we first remove each edge $(u, u + 2p - 1)$ in $Q_{2p-1,\,2p-1}$, $0 \leq u \leq 2p - 2$, such that the union of the remaining edges in any pair of 1-factors yields a pair of paths, each having a self-loop at both ends. Next, we remove the vertices $v$ and $v + 2p - 1$ from $Q_{2p-1,\,2p-1}$, for some $v \in \{0, 1, \ldots 2p - 2\}$, followed by all the edges incident to those vertices. Finally, we remove all edges colored $v$, which leads to all the edges of that 1-factor being deleted. Therefore, the union of the remaining edges in any pair of the other $(2p - 2)$ 1-factors must yield paths having a self-loop on at most one end. Hence, this graph must model the placement of data and parity in an array to tolerate two disk failures.

### 3.2.3 *Using $K_{p,\,p}$ to model p disk arrays*

The previously described models using $Q_{p,\,p}$ and $Q_{2p-1,\,2p-1}$, $p$ a prime, placed data and parity in arrays with $p - 1$ and $2p - 2$ disks respectively to tolerate two disk failures. We now show how to use $Q_{p,\,p}$ to place data and parity in arrays with $p$ disks to tolerate two disk failures.

First, color each edge $(u, v)$ in $Q_{p,\,p}$ with $u + v \pmod p$ to obtain a perfect 1-factorization. Consider the Hamiltonian cycle formed by the union of a pair of 1-factors of $Q_{p,\,p}$ having edges colored $a$ and $b$, $0 \leq a, b \leq p - 1$. Let $a \equiv 2u \pmod p$ and $b \equiv 2v \pmod p$. Then, as proven earlier,

the distance between vertices $u$ and $v$, as well as between vertices $u + p$ and $v + p$ on that cycle is $p - 1$. Now, from each 1-factor with edges colored $2v \pmod p$, $0 \leq v \leq p - 1$, we remove the edges $(v, v + p)$ and $(v + 1, (v + p - 1) \pmod p + p)$ to obtain a matching with $p - 2$ edges. Then the union of a pair of such matchings with edges colored $2u \pmod p$ and $2v \pmod p$ and their identically colored self-loops form paths, each having a self-loop on at most one end. This is because, the removal of the edges $(u, u + p)$ and $(v, v + p)$ from the Hamiltonian path formed by the union of edges colored $2u \pmod p$ and $2v \pmod p$ yields two paths, each having a self-loop at both its ends. Finally, since the distance between vertices $(u + 1) \pmod p$ and $(v + 1) \pmod p$ on that Hamiltonian path is $p - 1$, each of the aforementioned two paths must have exactly one of the edges $(u + 1, (u + p - 1) \pmod p + p)$ and $(v + 1, (v + p - 1) \pmod p + p)$ on it. Hence, upon removing them we must obtain paths, each having a self-loop on at most one end. Again, by our argument in section 2, this graph must model the placement of data and parity in an array to tolerate two disk failures.

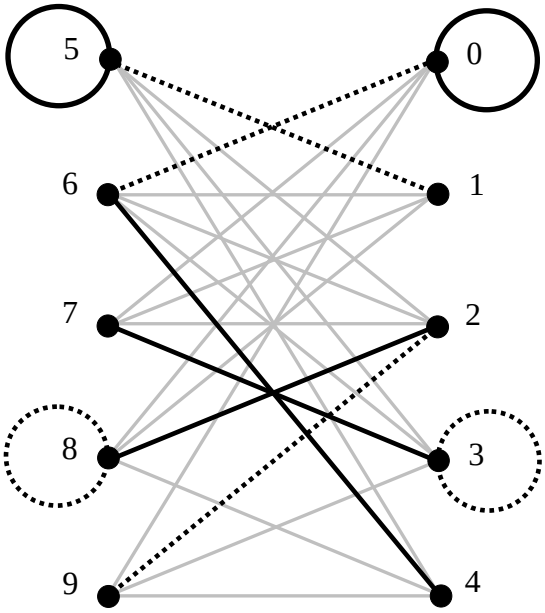


Figure 17: Paths corresponding to the reconstruction sequences of blocks on disks 1 and 2.

Figure 17 displays the pair of matchings in $Q_{5,\,5}$ corresponding to disks 0 and 1. These are the edges of the 1-factor with edges colored 0 remaining upon removing the edges (0, 5) and (1, 9), and the edges of the 1-factor with edges colored 1 remaining upon removing the edges (3, 8) and (4, 7). The union of these matchings yields four paths with vertices 1, 4, 7, and 9 as their ends, with each of degree 1, and a self-loop at its other end. Thus, reconstruction of disks 0 and 1 in the event of their failure can start with any data block corresponding to the edges incident to those vertices. Figure 18 shows the five-disk array obtained using the previously described model. If disks 1 and 2 in this array were to fail for instance, their data blocks could be reconstructed in the order given by the sequence of blocks {1, 5}, {5, 5} followed by {7, 3), {3, 3), followed by {4, 6}, {6, 0}, {0, 0} and finally {9, 2}, {2, 8}, {8, 8}.

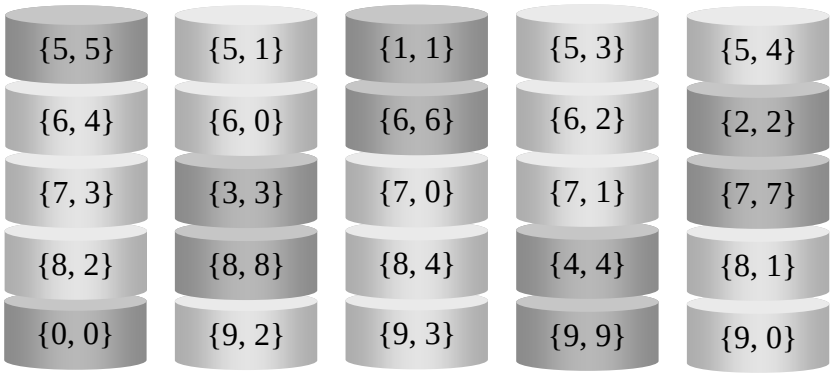


Figure 18: An array constructed from $Q_{5,\,5}$.

# 4. METHODS TO MODEL AND FORECAST NETWORK ATTACKS

We propose a model that decouples attack graphs from the networked systems on which they are described. This approach is motivated by the following reason. Each system is reachable from every other in a network. Hence, to execute the sequence of exploits needed to achieve its goal, a malware relies only on the existence of the preconditions necessary to execute each, regardless of which system provides each precondition. However, to meaningfully define an *attack graph* unique to an attack goal and an *attack path*, we first state the following definitions.

**Definition 1**: A *network* is a pair $G = (S, E)$, where $S$ is a set of systems and $E$ is a set of 2-tuples, and given $s_i$, $s_j$ in $S$, $(s_i, s_j)$ is in $E$ if and only if $s_i$ and $s_j$ can communicate with each other. We assume that $s_i$ can communicate with $s_j$ if physical cabling can be traced from $s_i$ to $s_j$, and the routers that control data flow on the cabling between those systems, permit data from $s_i$ to reach $s_j$ and vice versa.

**Definition 2**: A network *service* is an application on a system that has the ability to transmit and receive data to and from network services on other systems. Each service utilizes a unique port number for the transmission and reception of data from its peers.

**Definition 3**: A network service on a system $s$ is said to be targeted by an exploit $e$ if $e$ attempts to utilize it to enhance its privileges on $s$. Since we want to determine how applications with vulnerabilities are exploited by their peers in a network, we restrict our discussion to network services only. For brevity, we shall hereafter refer to each simply as a service.

**Definition 4**: A service $v$ on a system $s$ is said to be compromised if an exploit that targets $v$ succeeds in gaining privileges on $s$. We refer to $s$ as the host of $v$.

**Definition 5:** A service $v$ is assigned a *rank* using the set of permissions on a system hosting $v$ that can be usurped by compromising $v$. Thus, given services $v_1$ and $v_2$, the rank of $v_1$ is greater than that of $v_2$, if the set of permissions usurped by compromising $v_2$ is a subset of the set of permissions usurped by compromising $v_1$. We denote the rank of a service $v$ as $rank(v)$.

**Definition 6:** A *honeypot* is a system in a network that admits exploits that target it, and enables security experts to analyze the characteristics of such exploits to determine the behavior of the attackers that perpetrate them. In the following section we construct the network required to derive attack graphs from observed exploits.

### 4.1 A network to derive attack graphs

Suppose that $R = \{r_1, r_2, \ldots, r_n\}$ and $S = \{s_1, s_2, \ldots, s_n\}$ are two sets of honeypots where each $s_i$ in $S$ rejects any transmissions to it other than those from $r_i$ in $R$, for $1 \leq i \leq n$. This can be achieved, for instance, by confining each pair of systems $\{r_i, s_i\}$ to a unique domain, and rejecting all transmissions to $s_i$ originating from outside its own domain. However, systems outside the domain of $s_i$ can counter this by impersonating $r_i$. They can achieve this by overwriting the valid 2-tuple comprised of the IP and media access control (MAC) addresses of $r_i$ in the ARP cache of $s_i$ with the spurious 2-tuple comprised of the IP address of $r_i$ and the MAC address of the impersonator. To prevent this from occurring, the initially valid 2-tuple that associates the IP and MAC addresses of $r_i$, is never allowed to be replaced.

Each system in $R$ and $S$ is actually a virtual machine that is simulated within a physical server using a tool such as VMware WorkStation. This enables a physical server to simulate multiple systems in a network, and thereby furnish the means for realizing the large number of

systems needed to implement our proposed solution in practice. For the sake of brevity, we shall henceforth refer to each virtual machine simply as a system.

Initially, the honeypots are initialized such that, no service on any system in *R* transmits data to any service on any system in *S*. Any transmission that is observed thereafter is scanned to determine if it is harmless. For example, a SYN packet that is typically used to initiate a TCP connection or to scan a port is judged to be harmless. On the other hand, an unfamiliar transmission is considered to be an exploit. In the latter case, the system in *R* from which the transmission originated is deemed to have been compromised.

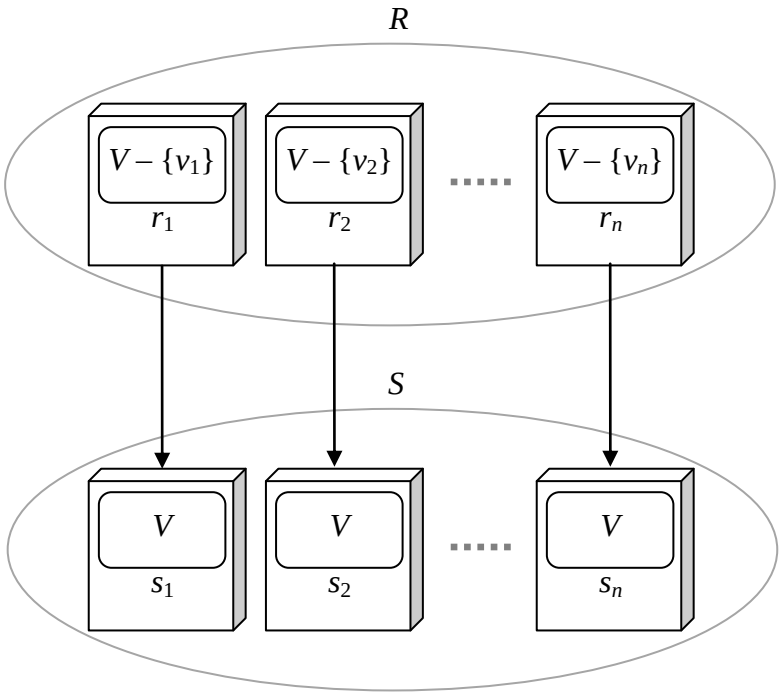


Figure 19: A network with 2n honeypots.

Suppose that $V = \{v_1, v_2, \ldots, v_n\}$ is the set of services on which attack paths are to be found. Then, we initialize each system $r_i$ in $R$ to contain each service in $V - \{v_i\}$, for $1 \leq i \leq n$.

Figure 19 illustrates the aforementioned network. The rectangle with rounded corners shown within each system specifies the services on that system, and arc ($r_i$, $s_i$) indicates the exploits that are permitted to originate from $r_i$ to target $s_i$, but not vice versa.

Legal liabilities inhibit honeypots from propagating attacks that are known to it. As a result, an attacker can identify a system *s* that has been ostensibly compromised as being a honeypot, if exploits designed by the attacker to originate from *s* do not occur. To ensure that attackers continue to launch exploits targeting systems in *S* from system in *R*, it is essential that they are not alerted to the fact that systems in *S* are honeypots. Hence, attacks are allowed to originate from systems in *S*. To this end, we allow systems in *S* to transmit only to those in another set, *T* say, where each system in *T* is protected with up-to-date internet security software.

### 4.2 Deriving an attack graph

**Definition 7:** An attack graph is a pair $H = (V, A)$, where *A* is a set of arcs, and given $v_i$, $v_j$ in *V*, ($v_i$, $v_j$) is in *A* if and only if $v_i$ can be compromised to yield the postcondition necessary to exploit $v_j$. Then, $v_i$ is called the predecessor of $v_j$. Next, we show how to derive such attack graphs.

Suppose that *k* systems in *S* are each targeted on port *p* by an instance of exploit *e*. Since, each service utilizes unique port numbers we can assume that the same service *v* has been targeted by *e* on each of the *k* systems. Now, an instance of *e* targeting a given system in *S* must originate from a unique system in *R*, since $r_i$ in *R* is permitted to transmit to $s_i$ in *S* only, for $1 \leq i \leq n$. Let *p* be the probability that a service on a system *r* in *R* has been compromised given that *r* is the source of an exploit *e*. Assuming that the probability of an internal attack on *r*, such as one perpetrated by a disgruntled employee with administrative privileges on *r*, is relatively low, we have $p \approx 1$. However, it is possible that a service has been compromised in *r* even when no

exploit is observed to originate from it. For instance, an attacker may have compromised a service on $r$ but not yet leveraged it to launch an exploit. So, let $q$ be the probability that a service in $r$ has been compromised given that $r$ is not a source of $e$. In our proposed network, each system in $S$ can be targeted by an exploit launched from a unique system in $R$ only. Hence, an attack that is keen to propagate itself to all systems in $S$ is very likely to eventually use a compromised service $u$ in each system in $R$ to launch the corresponding exploit that required the attacker to compromise $u$ in the first place. In other words, if after prolonged observation, $r$ is not found to be a source of $e$, it is very likely that no service has been compromised on $r$ to launch $e$. Thus, we have $(1 - q) \approx 1$, and therefore, $0 \approx q < p \approx 1$.

Let $C(n, k)$ denote the number of $k$-combinations of an $n$-set. Now, suppose that $e$ originates from $k$ given systems in $R$. Since, there are $C(n - k, m - k)$ $m$-sets of systems with every system in each of those $m$-sets containing each service common to those $k$ systems, the probability $p_k$ that a service common to $k$ such systems has been compromised equals $C(n - k, m - k)p^k q^{m-k}$. Then, since $m = n - 1$ in our network we have,

$$\frac{p_k}{p_{k-1}} = \frac{p(m-k+1)}{q(n-k+1)} = \frac{p(n-k)}{q(n-k+1)}$$

Now, $0 \approx q < p \approx 1$. Hence, $\frac{p_k}{p_{k-1}} > 1$, for $1 \leq k < n$.

Thus, the probability that a service has been compromised is greater when the number of systems $k$ containing it, which are the origins of a common exploit, is greater. Now consider the following cases.

Case 1 $k = (n - 1)$

The intersection of the sets of services in the $(n - 1)$ systems in $R$ from which instances of $e$ exploit the $(n - 1)$ systems in $S$, must contain exactly one service, $u$ say. By our preceding discussion, the likelihood of an attack having compromised $u$ to yield the postcondition to exploit $v$ must be high. Hence, we designate $u$ as the predecessor of $v$.

Case 2 $k < (n - 1)$

The intersection of the sets of services that are the origins of $e$ has $(n - k)$ services, $v_1$, $v_2$, …, $v_{n-k}$ say, where $(n - k) > 1$. Let $U = \{v_1, v_2, \ldots, v_{n-k}\}$. In that case, each service in $U$ may be a predecessor of $v$ in the derived attack graph. However, if $k$ is relatively small, and the number of services in $V$ that admit the precondition necessary for $e$ to occur is small, then having all services in $U$ as predecessors of $v$ in an attack graph yields larger numbers of false positives. To avoid this, we can wait till the value of $k$ is observed to be larger. In the meantime we may apply the following criteria to prune the number of services in $U$ that are predecessors of $v$.

(i) Suppose that $(n - 1)$ systems in $R$ having service $u$ have been observed to target the service $v$ as well as the service $v_i$ on systems in $S$, where $i \in \{1, 2, \ldots, k\}$. By our argument in Case 1, $u$ must then be a predecessor of both $v_i$ and $v$. Then, $v_i$ cannot be a predecessor of $v$ for the following reason. If an attacker has already found a service $u$ to compromise in order to exploit $v$, then there is no benefit in compromising $v_i$ as an intermediate step after compromising $u$ in order to exploit $v$. Hence we can eliminate $v_i$ from the set of likely predecessors of $v$.

(ii) Suppose that $(n - 1)$ systems in $R$ having service $v_i$ are observed to be targeting the service $w$ on systems in $S$, where $i \in \{1, 2, \ldots, k\}$, and $rank(w) < rank(s)$. Then, by our argument in

Case 1, $v_i$ is a predecessor of service $w$. Now an attacker that has compromised service $v_i$ earlier to exploit $w$ cannot be exploiting $v$ later if $rank(w) < rank(v)$. This is because exploits are designed by an attacker to usurp privileges with reasonable confidence of success. Hence, an attacker that intends to exploit a higher ranked service $v$ will do so without exploiting a lower ranked service $w$ first. Therefore, we can eliminate $v_i$ from the set of likely predecessors of $v$.

Case 3 $k = n$

Unfortunately, this implies that two or more services have been compromised to enable exploit $e$ to be launched from each system in $R$. However, such an occurrence is rare because of the following reason. In order to obtain the desired privileges on a system $r$ that enables the launch of $e$ from $r$, an attacker has to first find a vulnerability admitted by a service on $r$ that can be compromised to yield those privileges, and then write a program to achieve that. Neither task is trivial. Hence, upon finding a vulnerability in a service and the exploit necessary to compromise it, there is little reason for the attacker to find another service that can be compromised to enable the launch of $e$, unless the existing mechanism to compromise the former service is rendered ineffective by network security. Hence, the occurrence of this case should be limited.

Using the inferences from Case 1 and Case 2, an attack graph $H = (V, A)$ is constructed over time. A service in $V$ that does not have a successor in $H$ is then that which can be exploited by an attack to realize the postcondition satisfying its goal. Note that $H$ may have multiple such services. Also, it may admit self-loops as well as arcs having the same tail and head. Now, an attack graph may be alternatively defined as follows.

**Definition 8**: An *attack graph* is a directed graph $H = (V, A)$ where each vertex $v$ in $V$ is a vulnerability and each arc $(u, v)$ in $A$ is an action $a$ originating from a system having vulnerability $u$ that has been exploited to yield the condition for $a$ to exploit a system with vulnerability $v$. A unique vertex $v_f$ in $V$ represents the *end vulnerability* that can be exploited by the attack to realize the postcondition that immediately satisfies its goal. Thus an attack path is a unique path in $H$ that contains $v_f$ as its end.

Figure 20 illustrates an example of an attack graph described by Definition 8, where vulnerabilities $v_1$ and $v_2$ – each having in-degree 0 – are starting points for an attack, and whose goal is achieved by exploiting vulnerability $v_6$. In this graph, the sequences of vulnerabilities $v_1$, $v_3^+$, $v_6$, and $v_2$, $v_4$, $v_6$, and $v_2$, $v_5$, $v_6$ represent attack paths, where $v^+$ denotes one or more occurrences of $v$. Each has vulnerability $v_6$ as its end, which can be exploited to yield the postcondition that satisfies the goal of the attack.

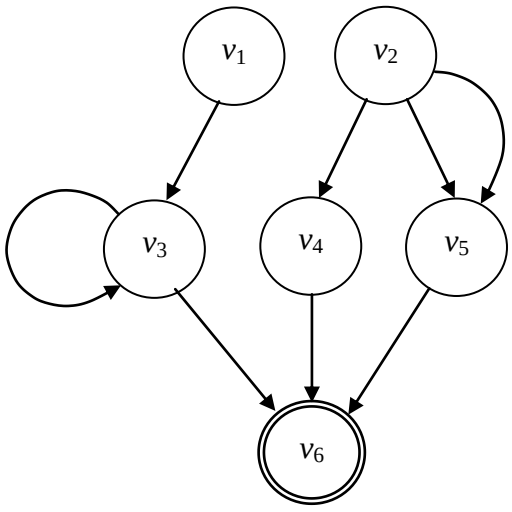


Figure 20: An attach graph with attack paths.

The graphs given by Definition 7 and Definition 8 are closely related for the following reason. A given vulnerability is unique to a service. Hence for each edge $(u, v)$ in the graph given by Definition 8, there exists an edge between the services containing vulnerabilities $u$ and $v$ respectively in the graph given by Definition 7, and vice versa. Hence, the terms service and vulnerability are interchangeable. However, a graph $H$ given by Definition 7 admits multiple services without successors. For each such service $s$ in $H$ we can obtain the graph $H_s$ that has $s$ as its only vertex without a successor, by removing any vertex from $H$ that is not an ancestor of $s$. In this manner we obtain multiple graphs from $H$ that together contain all the attack paths in $H$. Each attack graph that we refer to hereafter shall be assumed to have exactly one vertex without an ancestor.

### 4.3 Using an attack graph

We now describe how the given model of an attack graph can be used to determine the likelihood of an alert corresponding to a multi-step attack. Suppose an IDS generates an alert corresponding to an exploit originating from system $s_2$ targeting vulnerability $v_1$ at time $t_1$ on system $s_1$. Then, we determine if $s_1$ admits $v_1$. If not, we discard that alert as irrelevant, else we find a predecessor of $v_1$ in a chosen attack graph $H$, $v_2$ say, and determine if $s_2$ admits $v_2$. If true, we determine if an alert has been seen at some time $t_2 < t_1$ for an exploit targeting $v_2$ on $s_2$. Suppose this is true, and the system from which that exploit originates is $s_3$. Then we find a predecessor of $v_2$ in $H$, $v_3$ say, and determine if $s_3$ admits $v_3$. If true, we again determine if an alert has been seen at some time $t_3 < t_2$ for an exploit targeting $v_3$ on $s_3$. We continue backtracking in this manner until one of the following three cases occurs.

(a) Upon examining an alert at time $t_{k-1}$ corresponding to an exploit from system $s_k$ targeting vulnerability $v_{k-1}$ on system $s_{k-1}$, we backtrack to the predecessor of $v_{k-1}$ in the chosen attack graph, $v_k$ say, and find that $s_k$ admits $v_k$, but $v_k$ has no predecessor.

(b) After inspecting the alert corresponding to an exploit from system $s_{k+1}$ targeting vulnerability $v_k$ on system $s_k$, we backtrack to the predecessor of vulnerability $v_k$ in the chosen attack graph, $v_{k+1}$ say, and find that $v_{k+1}$ is not admitted by $s_{k+1}$. There are three possible scenarios that can cause this:

  (b.1) $s_{k+1}$ already admits the postcondition for an exploit to target $v_k$ on $s_k$. This implies that the exploits targeting the sequence of vulnerabilities $v_k$, $v_{k-1}$, …, $v_1$ represent a legitimate multi-step attack on system $s_1$.

  (b.2) The chosen attack graph may not correspond to the attack that is ostensibly occurring.

  (b.3) The attack may be exploiting an unknown vulnerability on $s_{k+1}$ to exploit the sequence of vulnerabilities $v_k$, $v_{k-1}$, …, $v_1$ thereafter.

(c) Upon examining an alert generated at time $t_{k-1}$ for an exploit from system $s_k$ targeting vulnerability $v_{k-1}$ on system $s_{k-1}$, we backtrack to the predecessor of $v_{k-1}$ in the chosen attack graph, $v_k$ say, and find that $s_k$ admits $v_k$. However, there is no alert originating from any system that targets $v_k$ on $s_k$ at time $t_k < t_{k-1}$.

### 4.4 Correlating an alert to an attack

Using the previously described model of an attack graph, we can correlate an alert to an attack in the following manner. If (a) holds true and $v_1 = v_f$ in the attack graph being used, then we have found a complete sequence of exploits targeting each vulnerability in an attack path with $v_f$ at its end. We refer to such a path as a *complete attack path* and its length as the *complete*

*attack length*. It would then be intuitive to assume that such an occurrence indicates a legitimate multi-step attack that targets system $s_1$. In the event (b.2) holds true, we can substitute the attack graph being currently used with another that contains vulnerability $v_1$ and repeat the process of backtracking along the new graph to see if (a) or (b.1) holds. We may obtain such a graph after a number of substitutions. However, if no such attack graph is found, then we have to reconcile the matching of partial sequences of vulnerabilities $v_k$, $v_{k-1}$, …, $v_1$ for each examined attack graph with the likelihood that it represents a real attack. It is intuitively appealing that the greater the length of the sequence $v_k$, $v_{k-1}$, …, $v_1$ that is matched, the greater the likelihood that the alert targeting vulnerability $v_1$ corresponds to a legitimate multi-step attack. We refer to the value of $k$ in this *matching attack subpath* as the *matching attack length*, and its difference from the complete attack length as the *remaining attack length*. Furthermore, if $u_i$, $u_{i-1}$, …, $u_1$ and $v_j$, $v_{j-1}$, …, $v_1$ are the partial sequence of vulnerabilities that are matched by employing the attack graphs $H$ and $H'$ respectively say, with $i > j$, then it is intuitive to assume that $H$ is the attack graph that best describes the perceived attack. Finally, (b.3) and (c) may be caused by the exploit on $s_k$ being stealthy, or the IDS rules not being adequately comprehensive to detect such an exploit.

Algorithm 1 uses the previously described model to determine if an alert corresponding to an exploit constitutes an attack. Step 1 is executed once for a given network, while Step 2 is executed each time an alert is generated by an IDS thereafter. If the alert is correlated to an attack, Step 2 returns the sequence of 2-tuples comprised of the systems and their corresponding vulnerabilities that have been iteratively targeted by that attack. Such a sequence forms an attack instance as defined next.

Algorithm 1

***Correlate***(vulnerability *v*, system *s*, time *t*, int *matchLen*)
{
**if** ($v \neq null$) and (*s* admits *v*) and
(alert found for exploit targeting *v* on *s* at time $t_v < t$) **then**
$e \leftarrow$ exploit at time $t_v$ for which alert is found
$t \leftarrow t_v$
$P \leftarrow P + (v, s)$
$s \leftarrow$ origin system of *e*
$matchLen \leftarrow matchLen + 1$
**if** (*v* has predecessors in $G_i$) **then**
**for** each predecessor *u* of *v* in $G_i$ **do**
*Correlate*(*u*, *s*, *t*, *matchLen*)
**end for**
**else**
*Correlate*(*null*, *s*, *t*, *matchLen*)
**end if**
**else**
$$corr \leftarrow \frac{matchLen}{matchLen + d(w, w_f)} + \frac{1}{d(w, w_f) + 1}$$
**if** ($corr > maxCorr$) **then**
$maxCorr \leftarrow corr$
$P_{max} \leftarrow P$
**end if**
**end if**
}

Step 1: *S* = Systems in a subnet
*V* = Set of vulnerabilities admitted by all systems in *S*
*n* = Size of the set of attack goals admitted by *V*
for attack goal $i \leftarrow 1$ to *n* generate attack graph $G_i$ on *V*

Step 2: $e \leftarrow$ exploit at time *t* for which alert is raised
$s \leftarrow$ system targeted by *e*
$w \leftarrow$ vulnerability targeted by *e*
$P_{max} \leftarrow \phi$
$maxCorr \leftarrow 0$
$i \leftarrow 1$
$t \leftarrow \infty$

**while** ($i \leq n$) **do**
    $d(w, w_f) \leftarrow$ distance of $w$ to *end vulnerability* $w_f$ in $G_i$
    $P \leftarrow \phi$
    *Correlate*($w$, $s$, $t$, –1)
    $i \leftarrow i + 1$
**end while**

**if** ($maxCorr > \tau$) **then**
    **return** ($P_{max}$, $i$)
**else**
    **return** *null*
**end if**

Figure 21: Algorithm to identify an attack from an alert.

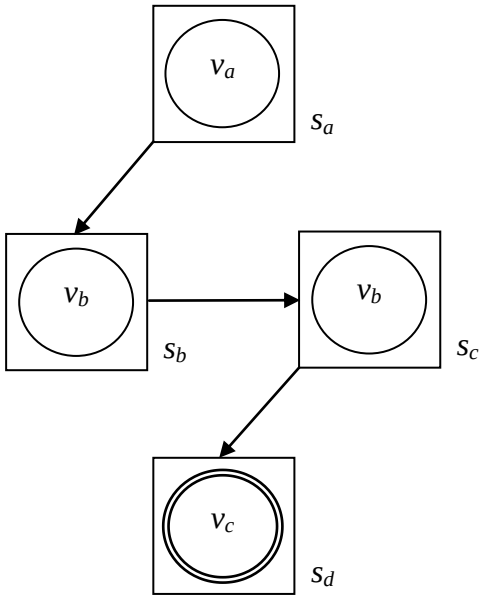


Figure 22: An attack instance with an attack subpath

**Definition 9**: Let $G = (S, E)$ be a network and $H = (V, A)$ be an attack graph. Given an attack path in $H$ defined by the sequence of vulnerabilities $v_1, v_2, \ldots, v_k$, an *attack instance* is the

sequence of tuples $(s_1, v_1)$, $(s_2, v_2)$, …, $(s_i, v_i)$ where $i \leq k$ and systems $s_1, s_2, \ldots, s_i$ in $S$ admit $v_1, v_2, \ldots, v_i$ respectively as a matching attack subpath.

Figure 22 illustrates an example of an attack instance comprised of the tuples $(s_a, v_a)$, $(s_b, v_b)$, $(s_c, v_b)$, and $(s_d, v_c)$. In this instance, systems $s_a, s_b, s_c, s_d$ admit an attack path comprised of the attack path $v_a, v_b, v_b, v_c$. The correlation factor *corr* in Algorithm 1 is defined such that it is proportional to the ratio of the matching attack length to the complete attack length. At the same time, it is inversely proportional to the remaining attack length. This enables the correlation value of an attack to be proportional to the proximity of $v_f$ to the vulnerability targeted by its most recently observed exploit. Finally, the threshold $\tau$ is a suitable real value in (0, 2.0).

Execution of Step 2 of Algorithm 1 returns the tuple $(P, i)$ if an attack instance $P$ using the attack graph $H_i$ is correlated to the investigated alert. Now, suppose that it returns $(P_1, i_1)$ in response to an alert for an exploit $e_1$ at time $t_1$, and $(P_2, i_2)$ in response to an alert for an exploit $e_2$ at time $t_2$, where $t_2 > t_1$. If $i_1 \neq i_2$ and $P_1$ is a subsequence of $P_2$, then $P_2$ must represent an attack that is different from $P_1$, since by Definition 1, each attack graph has a unique goal. On the other hand, if $i_1 = i_2$ then we obtain vindication of the algorithm's earlier result that suggested an attack with the goal of the attack graph having index $i_1$.

The depths of attack graphs are generally small since the average length of a sequence of vulnerabilities exploited to achieve a desired security breach is small. Furthermore, the system that is examined for the predecessor of a given vulnerability $v$ is specified by the alert for the exploit targeting $v$, and therefore, its discovery is achieved in constant time using suitable data structures such as a hash table. Thus, the order of Algorithm 1 is dominated by the number of attack graphs examined. If their number equals $n$, then the algorithm has order $\mathbf{O}(n)$. Moreover,

since the value of $n$ is independent of the size of the network, the proposed model with its associated algorithm to identify attacks is scalable.

Algorithm 1 identifies attack instances on a subnet when the vulnerabilities on all systems have been identified in Step 1. However, it does not identify the systems that are future targets of exploits by an identified attack. For example, suppose that $(s_1, v_1)$, $(s_2, v_2)$, …, $(s_j, v_j)$ is an identified attack instance using the attack graph with the final goal vulnerability $v_f$, where systems $s_1, s_2, \ldots, s_j$ admit vulnerabilities $v_1, v_2, \ldots, v_j$ respectively, and $v_j \neq v_f$. Then Algorithm 1 does not identify the systems that are likely to be targeted next by that attack to realize its goal of exploiting $v_f$. For instance, the attack may next target systems $s_a$ and $s_b$ that admit vulnerabilities $v_i$ and $v_{j+1}$ using exploits from systems $s_{i-1}$ and $s_j$ respectively, for some $1 \leq i < j$, as shown in Figure 23. Next, we describe the method to identify such systems.

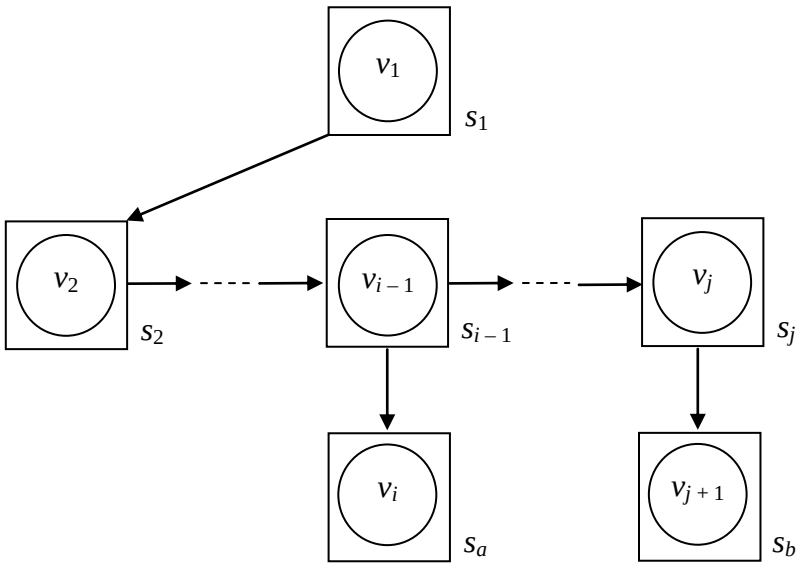


Figure 23: Future exploit targets of an identified attack.

## 4.5 Forecasting the propagation of attacks

To determine the systems that are likely to be targeted by an attack in the future utilizes the idea of matching the remaining attack length of an attack path in the direction of the goal. As described earlier, Algorithm 1 returns the index $i$ of the attack graph corresponding to the matched attack instance. Now, if the remaining attack length is zero, then the system specified in the first tuple ($v_1$, $s_1$) of an attack instance is the system targeted by the attack. On the other hand, if the remaining attack length is nonzero, $d$ say, then we iteratively find the sequence of systems $s_{k+1}, s_{k+2}, \ldots s_{k+d}$ such that, system $s_j$ admits vulnerability $v_j$, for $k + 1 \leq j \leq k + d$, $v_{j+1}$ is a child of $v_j$ in the attack graph, $v_{k+1}$ is the child of $v_1$, and $v_{k+d} = v_f$. However, in some iteration $j$, multiple systems may admit the vulnerability $v_j$ that is a descendant of $v_1$. Therefore, to definitively identify the next system that is likely to be targeted, we determine that malware's past preferences when selecting a system to exploit under such circumstances.

Consider a malware that has just compromised system $s$ within a private network by exploiting vulnerability $u$ on it, and whose eventual goal is to exploit vulnerability $v_f$ Furthermore, suppose that ($u$, $v$) is an arc in the attack graph with goal $v_f$. Since a malware propagates itself in the manner dictated by the flow of its coded logic, it can be expected to exhibit predictable traits in choosing the next system to exploit in the set of systems that all admit vulnerability $v$. For instance, that malware may be designed to next exploit, from $s$, the system that admits vulnerability $v$ and belongs to the same private network as $s$. On the other hand, if $s$ is a router, then that malware may be designed to immediately exploit another router that admits $v$. Similarly, it may choose the host having the lowest version of the operating system that admits vulnerability $v$. We now describe how such preferences are measured.

First, we define a *system state* as a tuple $z = (v, x_1, x_2, \ldots, x_m)$ in $V \times X_1 \times X_2 \times \ldots \times X_m$ of $m + 1$ independent variables that describes the state of a system, where $V$ is the set of all vulnerabilities and $X_i$ is the domain of characteristic $x_i$, $1 \leq i \leq m$, for some integer value $m$. For example, we may have $X_1 = \{a_{11}, a_{12}, a_{13}\}$, where $a_{11}$ denotes a system having a private class $C$ IP address of the form $192.168.b_1.b_2$, with $0 \leq b_1, b_2 \leq 255$; $a_{12}$ represents a system having a private class $B$ IP address of the form $172.b_1.b_2.b_3$, with $16 \leq b_1 \leq 31$ and $0 \leq b_2, b_3 \leq 255$; and $a_{13}$ represents a system having a private class $A$ IP address of the form $10.b_1.b_2.b_3$, with $0 \leq b_1, b_2, b_3 \leq 255$. Similarly, we may have $X_2 = \{a_{21}, a_{22}, \ldots, a_{2r}\}$ as the set of $r$ unique combinations of operating systems and version numbers that indicate their respective service packs or patches. Also, we may have $X_3 = \{a_{31}, a_{32}\}$, where $a_{31}$ denotes a system that is a router and $a_{32}$ denotes otherwise. In this manner, we identify $x_1, x_2, \ldots, x_m$ while ensuring that their respective domains do not omit any values that characterizes a system.

Since each variable comprising a system state has a finite domain, we must have a finite number of such states. Then, the conditional probability of an attack being in the system state $z_i$ when the preceding system state is $z_j$, denoted as $p(z_i \mid z_j)$, may be derived in the following manner. For each pair of system states $z_i$ and $z_j$ we determine over all attack instances of the form $(v_1, s_1)$, $(v_2, s_2)$, …, $(v_k, s_k)$, $k > 0$, the number of occurrences $c_j$ where a system $s_t$ is in state $z_j$, and the number of occurrences $c_{i,j}$ where a system $s_t$ is in state $z_j$ and system $s_{t+1}$ is in state $z_i$, for $t < k$. Then, $p(z_i \mid z_j) = c_{i,j}/c_j$. We may represent the probabilities of such pairs of states using a matrix $P = (p)_{i\,j}$, where entry $(i, j)$ equals $p(z_i \mid z_j)$. Now, given any system, its state must be a tuple in $V \times X_1 \times X_2 \times \ldots \times X_m$, since by construction, each domain is chosen such that each and every system is characterized by a unique value within it. As a result, a system's state is also

unique. Hence a given pair of systems $r$ and $s$ have a unique pair of corresponding states $z_r$ and $z_s$ respectively. Then the probability of an attack exploiting vulnerability $v$ on $s$ from $r$, using its prior exploitation of vulnerability $u$ on $r$ is given by the conditional probability $p(z_s \mid z_r)$. For simplicity of notation we denote this simply as $p(s, r)$.

Algorithm 2 utilizes the function *FindTarget* to perform a depth-first search for the targeted system. In each stage of recursion, it searches the branches from a system $r$ to each system $s$ in decreasing order of the conditional probability value $p(s, r)$, where $s$ admits the next vulnerability in the attack path towards $v_f$ following that admitted by $r$. If a target is not found by proceeding down a branch, *FindTarget* backtracks to $r$ and resumes its search on the branch having the next lower conditional probability value. It continues in this manner until a branch returns a target system, or all branches have been exhausted.

Algorithm 2

```
FindTarget(system s, set of systems R, vulnerability u)
{
    if (u = v_f) then
        return s
    else
        R ← R – s
        target ← null
        for each successor v of u in attack graph G_i do
            Let R_v = {r_1, …, r_j} be the set of systems in R that admit v,
            with p(r_i, s) ≥ p(r_{i+1}, s), 1 ≤ i < j
            if (R_v ≠ empty and target = null) then
                i ← 1
                while (target = null and i < j) do
                    target ← FindTarget(r_i, R_v, v)
                    i ← i + 1
                end while
            end if
```

```
            end for
        end if
        return target
}
```

Execution Step

$S$ = Systems in a subnet
$R \leftarrow S$
Suppose Algorithm 1 returns ($P$, $i$)
Let $P = (v_1, s_1), (v_2, s_2), \ldots, (v_k, s_k)$

**for** $j \leftarrow 1$ to $k$ **do**
    $R \leftarrow R - s_j$, where $s_j$ is a member of a tuple in $P$
**end for**
*FindTarget*($s_k$, $R$, $v_k$)

Figure 24: Algorithm to identify the target of an attack

# 5. CONCLUSION

The first objective of this research is to utilize graph theoretic techniques to model RAID capable of tolerating two random disk failures using bit-wise XOR operations only, and using an optimal ratio of space for parity. The techniques shown in this thesis place data and parity in arrays of $n$ disks to tolerate two random disk failures for the values of $n = (p - 1)$, $p$ and $(2p - 2)$. These values cover significant numbers of the disks that may constitute arrays in practice, but not all. Hence, extending the coverage to additional values of $n$ is of great practical interest.

The second objective of this research is to devise more powerful techniques for correlating alerts with attacks in the context of network defense. The technique shown in this thesis correlates alerts to attacks with relatively high certainty by examining a sufficient number of the former after they have occurred. While that number is relatively small, the systems on which those alerts are generated are implicitly sacrificed to the attack. This is undesirable if those systems contain critical data that is shared by multiple clients, and whose availability cannot be interrupted. Hence, extending our technique to achieve correlation before network resources have been compromised is of significant practical value.

## 5.1 Modeling two-disk fault-tolerant RAID with additional numbers of disks

In section 0, we demonstrated the technique for using the complete bipartite graph on $2p$ vertices $K_{p, p}$ to model an array with $p$ disks that is capable of tolerating two random disk failures. In a similar manner, it may be possible to use the complete bipartite graph on $(4p - 2)$ vertices $K_{2p - 1, 2p - 1}$ to model the placement of data and parity in an array with $(2p - 1)$ disks.

Perfect 1-factorizations are known to exist for the complete bipartite graph on $2n$ vertices $K_{n,n}$, where $n = p^2$, $p$ a prime [8]. Therefore, similar techniques may be used to obtain solutions for data and parity assignment in arrays with $p^2$ or $(p^2 - 1)$ disks to tolerate two disk failures. Developing such techniques is an area for future research.

### 5.2 Modeling RAID to tolerate three random disk failures

Arrays deployed in practice generally have smaller numbers of disks. Hence, their ability to tolerate the failure of up to two random disks is adequate. However, as increasingly larger arrays are deployed, the need for tolerance to more than two random disk failures is compelling. A scheme [22] that achieves this is illustrated in Figure 25.

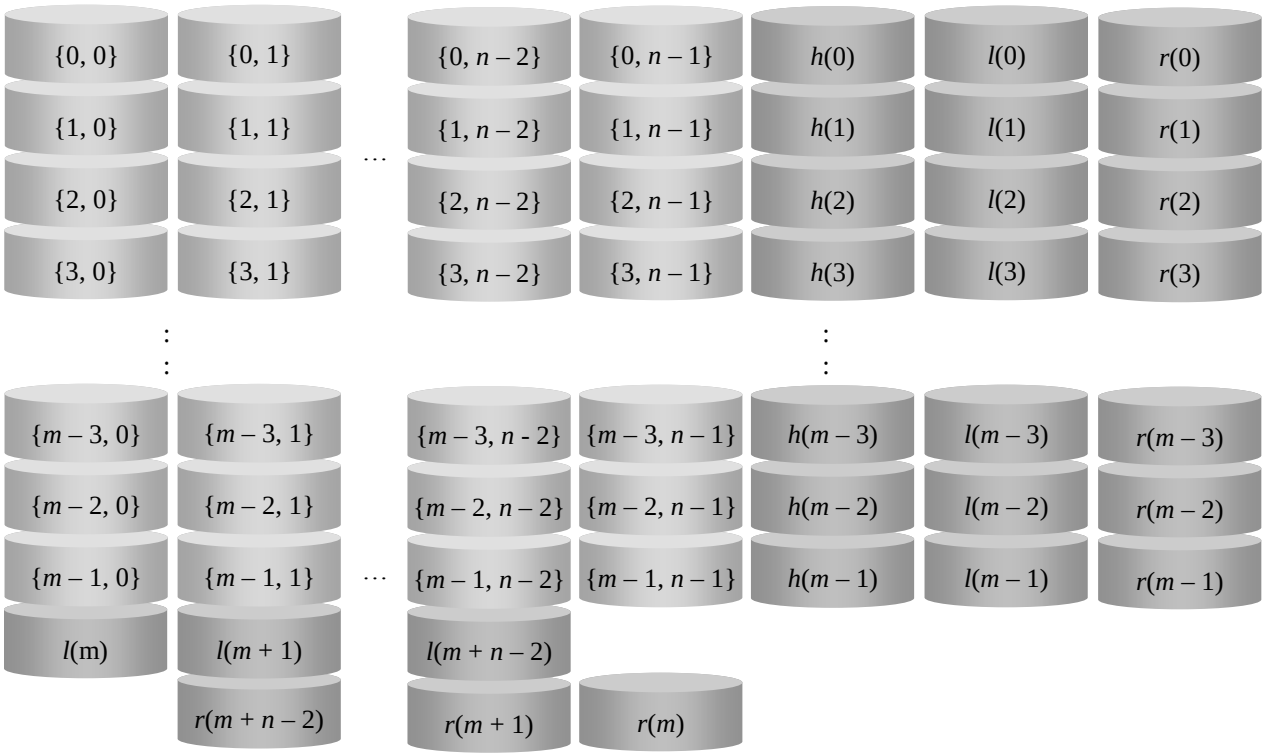


Figure 25: A three-disk fault tolerant array

In the previously illustrated array, parity is stored in the blocks $h(i)$, $l(j)$ and $r(k)$, for $0 \leq i \leq m-1$, $0 \leq j, k \leq m+n-2$, and $m > n$. These parities are calculated in the following manner.

$$h(i) = \{i, 0\} \oplus \{i,1\} \oplus \ldots \oplus \{i, n-1\}$$

$$l(j) = \begin{cases} \{0, j\} \oplus \{1, j-1\} \oplus \ldots \oplus \{j, 0\} & \text{if } 0 \leq j \leq n-1, \\ \{j-n+1, n-1\} \oplus \{j-n+2, n-2\} \oplus \ldots \oplus \{j, 0\} & \text{if } n \leq j \leq m-1, \\ \{j-n+1, n-1\} \oplus \{j-n+2, n-2\} \oplus \ldots \oplus \{m-1, j-m+1\} & \text{if } m \leq j \leq m+n-2 \end{cases}$$

$$r(k) = \begin{cases} \{0, n-k-1\} \oplus \{1, n-k\} \oplus \ldots \oplus \{k, n-1\} & \text{if } 0 \leq k \leq n-1, \\ \{k-n+1, 0\} \oplus \{k-n, 1\} \oplus \ldots \oplus \{k, n-1\} & \text{if } n \leq k \leq m-1, \\ \{k-n+1, 0\} \oplus \{k-n, 1\} \oplus \ldots \oplus \{m-1, m+n-k-1\} & \text{if } m \leq k \leq m+n-2 \end{cases}$$

Now, suppose that three disks $i$, $j$ and $k$ have failed in this array. Then, we prove by induction that the data in those disks can be iteratively reconstructed as follows. Assume without loss of generality that, $i < j < k$. In the first iteration, we reconstruct blocks $\{0, i\}$, $\{0, k\}$ and $\{0, j\}$. Block $\{0, i\}$ is reconstructed using the parity in block $l(i)$. This is possible because the blocks used for computing the parity in block $\{0, i\}$ all occur to the left of disk $i$, which are all intact. Next, block $\{0, k\}$ is reconstructed using the parity in block $r(k)$. This is possible because the blocks contributing to the parity in block $\{0, k\}$ all occur to the right of disk $k$, which are all intact. Now block $\{0, j\}$ is reconstructed using the parity in block $h(0)$.

Suppose that in the $(t-1)^{th}$ iteration, we can reconstruct blocks $\{t-1, i\}$, $\{t-1, k\}$ and $\{t-1, j\}$. Then, we can reconstruct blocks $\{t, i\}$, $\{t, k\}$ and $\{t, j\}$ in the $t^{th}$ iteration as follows. A data block needed to reconstruct $\{t, i\}$ is of the form $\{t', i'\}$ where $t' < t$ if $i' > i$, and $t' > t$ if $i' < i$. Now if $i' < i$ then $\{t', i'\}$ must be intact since $j$ and $k$ are both greater than $i$. On the other hand, if $i' > i$ then $\{t', i'\}$ must be intact if $i' \neq j$ and $i' \neq k$. Now, if $i' = j$ or $i' = k$, $\{t', i'\}$ would have

been already reconstructed since, by assumption, we have reconstructed all blocks $\{t', j\}$ and $\{t', k\}$ for $t' \leq t - 1$. Finally, the parity block $l(i + t)$ that is used for reconstructing the block $\{t, i\}$ is either located on a disk containing parities only, or is located on a disk with index less than $i$, which are both intact by assumption.

Similarly, a data block needed to reconstruct $\{t, k\}$ is of the form $\{t', k'\}$ where $t' < t$ if $k' < k$, and $t' > t$ if $k' > k$. If $k' > k$ then $\{t', k'\}$ must be intact since $i$ and $j$ are both less than $k$. On the other hand, if $k' < k$ then $\{t', k'\}$ must be intact if $k' \neq i$ and $k' \neq j$. Now, if $k' = i$ or $k' = j$, $\{t', k'\}$ would have been already reconstructed since, by assumption, we have reconstructed all blocks $\{t', i\}$ and $\{t', j\}$ for $t' \leq t - 1$. Finally, the parity block $r(i + t)$ that is used for reconstructing the block $\{t, k\}$ is either located on a disk containing parities only, or is located on a disk with index greater than $k$, which are both intact by assumption.

Having reconstructed blocks $\{t, i\}$, $\{t, k\}$, we can reconstruct $\{t, j\}$ using the parity in block $h(t)$. Thus we have reconstructed all the desired blocks in iteration $t$, and thereby shown that the array can tolerate the failure of any three arbitrary disks.

Using this scheme, the ratio of space used for storing parity in an array of $n$ disks as a function of $m$ is $f(m) = \frac{3m + 2n - 2}{3m + 2n - 2 + mn}$. Now, $f'(m) = \frac{n(2 - 2n)}{(3m + 2n - 2 + mn)^2}$. Since $n > 1$, $f'(m) < 0$ for all $m \geq 1$. Therefore, $f(m)$ is monotonically decreasing in the interval $[1, \infty)$. Furthermore, $\lim_{m \to \infty} \frac{3m + 2n - 2}{3m + 2n - 2 + mn} = \frac{3}{n + 3}$. Hence, $f(m)$ is closest to the optimum value of $3/(n + 3)$ when $m$ is infinitely large. That is turn implies that parity cannot be rotated to enhance data throughput, for otherwise, this ratio assumes a relatively larger non-optimal value. For this reason, this scheme is not attractive in practice.

Some of the alternatives that may be investigated for the placement of data and parity in arrays to tolerate three disk failures are the factorization of graphs and hypergraphs such that the union of any three 1-factors yields a Hamiltonian path.

### 5.3 Correlating alerts to attacks without compromising systems

As shown earlier, honeypots provide a method for deriving attack graphs without putting system resources at risk. Honeypots also offer the potential for being used as targets to correlate alerts with attacks in the manner shown earlier. However, a challenge posed by such an approach is that, the presence of longer attack paths requires relatively large numbers of honeypots to ensure that sufficient numbers of correlated alerts are generated corresponding to an attack. Methods to prune the number of honeypots required for this purpose is an area of future research.

# APPENDIX A: RECONSTRUCTING IN ($p$ – 1) DISK ARRAYS

Suppose that disks $i$ and $j$ have failed in an array with $(p - 1)$ disks. Then their reconstruction is achieved in the following manner. Each block $\{u, v\}$ on disk $i$ is colored $u + v \pmod{p}$, and therefore satisfies the equation $u + v \equiv 2d \pmod{p}$. Now, block $\{u, v\}$ on disk $i$ can be reconstructed if, for some vertex $w$ in $W$, the edge $(u, w)$ is in a near-1-factor whose corresponding blocks lie on disk $j$. The edge $(u, w)$ is colored $u + w \pmod{p}$. Thus, we can start reconstruction of a block $(u, v)$ on disk $i$ that satisfies $u + w \pmod{p} \equiv 2j \pmod{p}$. That is, $u \equiv 2j - w \pmod{p}$. Conversely, reconstruction can be started on a block $\{u, v\}$ on disk $j$ that satisfies $u \equiv 2i - w \pmod{p}$.

We illustrate the aforementioned idea using the six-disk array shown in Figure 3. This array was created using $W = \{6\}$ and the complete graph on $p$ vertices, where $p = 7$. Now, suppose that the disks 0 and 1 have failed. Thus, $i = 0$ and $j = 1$. Then, the block $\{u, v\}$ on disk 0 from which reconstruction can be started is the one where $u \equiv 2 \cdot 1 - 6 \ (mod\ 7) = 3$. This is the block $\{3, 4\}$. Thereafter, each block reconstructed corresponds to the edge that is incident to the edge corresponding to the previously reconstructed block. Similarly the block $\{u, v\}$ on disk 1 from which reconstruction can be started is the one where $u \equiv 2 \cdot 0 - 6 \ (mod\ 7) = 1$. This is the parity block $\{1, 1\}$.

Figure 26 illustrates the pseudo code of the function *Repair*(disk $i$, disk $j$) that repairs failed disks $i$ and $j$. This function in turn calls the function *Reconstruct*(block $\{u, v\}$, parity group $v$) shown in Figure 27 to reconstruct the data block $\{u, v\}$ by taking the XOR of all blocks in the parity group $v$.

```
Repair (disk i, disk j)
{
        for d ← i and j do
                k ← d
                u ← 2k – w (mod p)

                if (k = i) then
                        k ← j
                else
                        k ← i
                endif

                v ← 2k – u (mod p)
                continue ← true

                do
                        Reconstruct(block {u, v}, parity group v)

                        if (u = v) then
                                continue ← false
                        else
                                if (k = i) then
                                        k ← j
                                else
                                        k ← i
                                endif

                                u ← v
                                v ← 2k – u (mod p)

                        endif

                while (continue = true)
        endfor
}
```

Figure 26: Repairing two failed disks

```
Reconstruct(block {u, v}, parity group v)
{
        result ← 0
        for disk d ← 0 to (n – 1) do
                x ← 2d – v (mod p)
                if (x ∉ W) and (x ≠ u) do
                        result ← result ^ data in block {v, x}
                endif
        endfor
        block {u, v} is assigned result
}
```

Figure 27: Computing the data in a block from its parity group

Note that, each parity group has $(n - 2)$ data blocks and one parity block, with disk $d$ not containing any block from parity group $v$ if $2d \equiv v + w \pmod{p}$, where $w \in W$.

# APPENDIX B: READING FROM FAILED DISKS

An important function of RAID systems is to provide uninterrupted access to data on failed disks that have not been reconstructed. Hence, a block on which requested data resides must be reconstructed prior to those without it. Now, the preceding algorithm reconstructs two failed disks in two iterations of the outermost for-loop, where the disk index $d$ assumes the values $i$ and $j$, without constraints on their order. Each iteration reconstructs the data blocks corresponding to the edges of a path terminating with a self-loop. However, the order in which $k$ assumes the values $i$ and $j$ is important when a data block on a failed disk has to be reconstructed to enable its contents to be read expeditiously. This is because, in an array with $(p - 1)$ disks, the average number of blocks that have to be reconstructed in an iteration to read a given data block is $(p - 3)/4$. That in turn requires $(p - 3)^2/4$ XOR operations on blocks. Hence, for large value of $p$, it is desirable to reconstruct a given data block in the first iteration. To this end, the function *FindReconstructionDiskIndex*(block $\{u, v\}$) given in Figure 28 computes the disk from which to start reconstruction such that the failed data block $\{u, v\}$ having the requested data is reconstructed more quickly. Note that, the function executes at most two iterations of the outermost while loop. Without loss of generality, if the value returned is $i$, then the first block reconstructed is $\{2i - w \pmod p, 2j - (2i - w) \pmod p\}$.

In practice, data is requested from blocks corresponding to the edges on both paths. However the order and frequency with which data is requested from a block may be utilized to prioritize it.

```
FindReconstructionDiskIndex(block {u, v}, disk i, disk j)
{
        found ← false
        d ← i

        while (found = false) do
                k ← d
                x ← 2k – w (mod p)

                if (k = i) then
                        k ← j
                else
                        k ← i
                endif

                y ← 2k – x (mod p)
                continue ← true

                do
                        if (u = x) and (v = y) then
                                continue ← false
                                found ← true
                        else
                        if (u = v) then
                                continue ← false
                        else
                                if (k = i) then
                                        k ← j
                                else
                                        k ← i
                                endif

                                x ← y
                                y ← 2k – x (mod p)
                        endif
                while (continue = true)

                if (found = true) then
                        return d
                else
                        d ← j
                endif
        endwhile
}
```

Figure 28: Finding the smallest reconstruction sequence.

# LIST OF REFERENCES


1. G. Alexanderson, “Euler and Königsberg's bridges: a historical view,” *Bulletin of the American Mathematical Society*, pp. 567 – 573, 2006.
2. G. Alvarez, W. Burkhard, and F. Cristian, “Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering”, *The 24th Annual International Symposium on Computer Architecture*, pp. 62 – 72, 1997.
3. P. Ammann, D. Wijesekera and S. Kaushik, “Scalable graph-based network vulnerability analysis,” *Proceedings of the* $9^{th}$ *ACM Conference on Computer and Communications Security*, pp. 217 – 224, 2002.
4. B. Anderson, “Symmetry groups of some perfect 1-factorizations of complete graphs,” *Discrete Mathematics*, vol. 18, no. 3, pp. 227 – 234, 1977.
5. B. Anderson, “A class of starter-induced 1-factorizations,” *Graphs and Combinatorics, Lecture Notes in Mathematics*, vol. 406, pp. 180 – 185, 1974.
6. M. Blaum, “A coding technique for recovery against double disk failures in disk arrays,” *IEEE International Conference on Communications*, SUPERCOMM/ICC '92, Discovering a New World of Communications, vol. 3, pp. 1366 – 1368, June 1992
7. M. Blaum, J. Brady, J. Bruck, and J. Menon, “EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures,” *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 192 – 202, 1995.
8. D. Bryant, B. Maenhaut, and I. Wanless, “A family of perfect factorisations of complete bipartite graphs,” *Journal of Combinatorial Theory*, Series A, vol. 98, pp. 328 – 342, 2002.

9. P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," *Proceedings of the USENIX FAST '*04 *Conference on File and Storage Technologies,* pp. 1 – 14, 2004.
10. J. Dawkins and J. Hale, "A systematic approach to multi-stage network attack analysis," *Proceedings of the Second IEEE International Information Assurance Workshop,* pp. 48 – 56, 2004.
11. N. Deo and S. Nanda, "A graph theoretic algorithm for placing data and parity to tolerate two disk failures in disk arrays," To appear in *International Journal of Applied Mathematics and Computer Science*.
12. N. Deo and S. Nanda, "A graph theoretic algorithm for placing data and parity to tolerate two disk failures in disk array systems," *Proceedings of the Ninth International Conference on Information Visualization*, London England, pp. 542 – 549, 2005.
13. C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," *Proceedings of the* $4^{th}$ *USENIX conference on File and Storage Technologies*, pp. 197 – 210, 2005.
14. R. Katz, D. Patterson and G. Gibson, "Disk system architectures for high performance computing," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1842 – 1858, 1989.
15. M. Kobayashi, "On perfect one-factorization of the complete graph $K_{2p}$," *Graphs and Combinatorics*, vol. 5, pp. 351 – 353, 1989.
16. A. Kotzig, "Hamilton graphs and Hamilton circuits," *Theory of Graphs and Its Applications,* (Proc. Sympos. Smolenice 1963), Nakl. CSAV, Praha, pp. 62 – 82, 1964.

17. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 190 – 201, 2000.
18. P. Laufer, "On strongly Hamiltonian complete bipartite graphs," *ARS Combinatoria*, vol. 9, pp. 43 – 46, 1980.
19. M. Müller and M. Jimbo, "Erasure-resilient codes from affine spaces," *Discrete Applied Mathematics*, vol. 143, issue 1-3, pp. 292 – 297, 2004.
20. S. Mylavarapu, J. Zachary, D. Ettlich, J. McEachen and D. Ford, "A model of conversation exchange dynamics for detection," *The* 47$^{th}$ *Midwest Symposium on Circuits and Systems*, vol. 3, pp. 231 – 234, 2004.
21. Y. Nam, D. Kim, T. Choe and C. Park, "Enhancing write I/O performance of disk array RM2 tolerating double disk failures," *Proceedings of the International Conference on Parallel Processing*, pp. 211 – 218, 2002.
22. S. Nanda, "Method and system for three disk fault tolerance in a disk array," U.S. Patent number 6792391.
23. S. Nanda and N. Deo, "A highly scalable model for network attack identification and path prediction" *Proceedings of the IEEE SoutheastCon* 2007, pp. 663 – 668, 2007.
24. S. Nanda and N. Deo, "Methods for placing data and parity to tolerate two disk failures in disk arrays using complete bipartite graphs," *Congressus Numerantium*, vol. 179, pp. 67 – 79, 2006.

25. S. Nanda and N. Deo, "One-factors and Hamiltonian paths in modeling data and parity placement in disk arrays," *Congressus Numerantium*, vol. 176, pp. 191 – 199, 2005.
26. S. Nanda and N. Deo, "An algorithm for a two-disk fault-tolerant array with (prime – 1) disks," *Congressus Numerantium*, vol. 171, pp. 13 – 23, 2004.
27. S. Noel, M. Jacobs, P. Kalapa and S. Jajodia, "Multiple coordinated views for network attack graphs," *Proceedings of the IEEE Workshop on Visualization for Computer Security*, pp. 99 – 106, 2005.
28. S. Noel and S. Jajodia, "Understanding complex network attack graphs through clustered adjacency matrices," *Proceedings of the* 21$^{st}$ *Annual Computer Security Applications Conference*, pp. 160 – 169, 2005.
29. S. Noel and S. Jajodia, "Managing attack graph complexity through visual hierarchical aggregation," *Proceedings of the* 2004 *ACM workshop on Visualization and data mining for computer security*, pp. 109 – 118, 2004.
30. S. Noel, E. Robertson and S. Jajodia, "Correlating intrusion events and building attack scenarios through attack graph distances," *Computer Security Applications Conference*, pp. 350 – 359, 2004.
31. C. Park, "Efficient placement of parity and data to tolerate two disk failures in disk array systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1177 – 1184, 1995.
32. D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks," *Proceedings of the ACM SIGMOD*, pp. 109 – 116, 1988.
33. D. Patterson, P. Chen, G. Gibson, and R. Katz, "Introduction to redundant arrays of inexpensive disks," *Proceedings of the IEEE COMP-CON*, pp. 112 – 117, 1989.

34. S. Patton, W. Yurcik and D. Doss, “An Achilles’ heel in signature based IDS: squealing false positives in Snort,” *Fourth International Symposium on Recent Advances in Intrusion Detection*, 2001.

35. C. Phillips and L. Swiler, “A graph-based system for network-vulnerability analysis,” *Proceedings of the* 1998 *Workshop on New Security Paradigms*, pp. 71 – 79, 1998.

36. J. Plank, “A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems,” *Software - Practice and Experience*, vol. 27, no. 9, pp. 995 – 1012, 1997.

37. R. Ritchey and P. Ammann, “Using model checking to analyze network vulnerabilities,” *Proceedings of the* 2000 *IEEE Symposium on Security and Privacy*, pp. 156 – 165, 2000.

38. K. Shah, S. Bohacek and A. Broido, “Feasibility of detecting TCP-SYN scanning at a backbone router,” *Proceedings of the* 2004 *American Control Conference*, vol. 2, pp. 988 – 995, 2004.

39. O. Sheyner, J. Haines, S. Jha, R. Lippmann and J. Wing, “Automated generation and analysis of attack graphs,” *Proceedings of the* 2002 *IEEE Symposium in Security and Privacy*, pp. 272 – 284, 2002.

40. F. Valeur, G. Vigna, C. Kruegel and R. Kemmerer, “A comprehensive approach to intrusion detection alert correlation,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 3, pp. 146 – 169, 2004.

41. D. Wagner, “One the perfect one-factorization conjecture,” *Discrete Mathematics*, vol. 104, pp. 211 – 215, 1992.

42. Y. Wang, Z. Liu, X. Cheng and K. Zhang , “An analysis approach for multi-stage network attacks,” *Proceedings of* 2005 *International Conference on Machine Learning and Cybernetics*, vol. 7, pp. 3949 – 3954, 2005.
43. L. Xu and J. Bruck, “X-Code: MDS array codes with optimal encoding,” *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272 – 276, 1999.
44. L. Xu, V. Bohossian, J. Bruck and D. Wagner, “Low density MDS codes and factors of complete graphs,” *IEEE Trans. on Information Theory*, vol. 45, no. 6, pp. 1817 – 1826, 1999.