Designs and constructions of combinatorial threshold secret sharing schemes

Chen Rui Qi, Hnin Sat Phyu Sin

SS015

2 Introduction

In the seminal paper on secret sharing (Shamir, 1979), Shamir put forward the following question: Eleven scientists are working on a secret project. They wish to lock up the documents in a cabinet so that the cabinet can be opened if and only if six or more of the scientists are present. What is the smallest number of locks needed? What is the smallest number of keys to the locks each scientist must carry?

This mathematical problem is not only applicable to the above scientists in question but also the current governments which have vaults containing confidential information, banks and financial institutions with their money vaults. (Desmedt, Safavi-Naini, and Wang 2003). Furthermore, secret sharing schemes have been adapted to be implemented into the digital world, where the secret sharing technique is used to secure access to platforms through encryption algorithms and their respective keys. (Cuˇrík, Ploszek, and Zajac 2022). However, there have been recent heists or security breaches in which confidential information have been leaked or cash been stolen due to a lack of security (CyberMagazine 2022) or were done by insiders who had access to these information or cash (Gharat 2022)

3 Secret Sharing Schemes

To keep our online and offline information secure, the secret sharing schemes aim to distribute the access to the confidential information among n participants such that only some predefined 'authorised' sets of participants, t, can obtain access to the secret, while all other 'unauthorised' sets are unable to. (Long et al. 2006).

The obvious advantage of this secret sharing scheme is that as a single person or 'unauthorised' sets of people are unable to access it. A security breach that involves one person or 'unauthorised' sets of people will not give away the access to the secret as the perpetrators will be unable to obtain the confidential information with the shares that a single person or 'unauthorised' sets of people have.

Secondly, as t increases, the confidential information is more secure as more people are needed to open it while having more variations of (t, n) will provide decision-makers with more choice as to how they can implement their security systems depending on their desired t or n value.

On the other hand, as t and n increases, the number of shares and locks required increases more than proportionately as will be explained in the following section of the proposal which may make the scheme more complicated or make it inconvenient for the n participants.

3.1 Shamir Secret Sharing Schemes (SSS)

The idea of a secret sharing scheme was brought up by Adi Shamir in 1979 (Shamir 1979). Shamir stated that SSS distributes secrets into shares such that all n participants have one share each. A minimum of t number of participants (shares) are then needed to reconstruct the secret. This is done by constructing A polynomial of power $t + 1$ in which n points will be randomly taken and given to the n number of people involved. The reconstruction of the secret can be obtained using the Lagrange Interpolation Formula and the secret will be revealed when $f(0)$.

$$f(x) = \sum_{j=1}^{t} y_{i_j} \prod_{1 \le l \le t,\, l \ne j} \frac{x - x_{i_l}}{x_{i_j} - x_{i_l}}$$

However there are many security vulnerabilities to the SSS (Lemnouar 2022) hence the need for other secret sharing schemes like the General Secret Sharing. Since secret sharing schemes from cumulative arrays have been shown to be the appropriate building blocks in non-homomorphic thresh-hold cryptosystems where the conventional secret sharing methods are generally of no use, in this research we meant to implement the secret sharing schemes from Generalised Cumulative Arrays realised with perfect hash families and Cumulative Arrays (GCA) with the python programming language and determine whether GCAs provide more a more optimal secret sharing scheme.

4 Method

In this research, we will be comparing the sharing scheme using Cumulative Array and the sharing scheme using Generalised Cumulative Array.

4.1 Sharing Scheme from Cumulative Array (CA)

4.1.1 Explanation of CA

(Xing, Ling, and Wang 2019, p.g. 98-101) demonstrates how to construct a cumulative array and its resulting secret sharing scheme. Suppose P = $\{P_1, P_2, ..., P_n\}$ is a set of n participants and Γ is an access structure on P.

Let X = $\{x_1, x_2, \ldots, x_d\}$ and let τ: $P \rightarrow 2^x$ be a mapping from P to the family of subsets of X. (τ, X) is a cumulative array for Γ if the following condition is satisfied:

$$\bigcup_{P \in I} \tau(p) = X, \text{ if and only if } I \in \Gamma.$$

Assume that the secret set, K, is an abelian group. To share a secret $k \in K$, the dealer randomly chooses d elements $k_1, k_2 ..., k_d \in K$ such that $k = k_1 + k_2 + \cdots + k_d$ then distributes the share to each participant

according to (τ, X). That is, the share of participant P_i is $\{k_j |$ if $X_j \in \tau(P_i)\}$. The shares from participants in $l \in 2^p$ consist of all the d components $k_1, k_2, \ldots, k_d$ if and only if $l \in \Gamma$.

To construct a minimal cumulative array or any given access structure (Wen-Ai and Martin 2005), let Γ be an access structure for P. We denote $\Gamma^+ = \{U_1, U_2, ..., U_d\}$ the set of maximal unauthorised sets with respect to Γ, and define the mapping τ by $\forall P \in P, \tau(P) = \{U_i \mid P \notin U_i, 1 \leq i \leq d\}$. Then (τ, Γ^+) is a cumulative array for Γ. For example, in a (t, n) threshold access structure Γ, the set of maximal unauthorised sets is $\Gamma^+ = \{U \subseteq P \mid |U| = t - 1\}$, so $|\Gamma^+| = \binom{n}{t-1}$ and the share of each participant consists of $\binom{n-1}{t-1}$ components of k_i's. For example, a (3, 5) threshold access structure will consist of $\binom{5}{3-1} = 10$ locks and $\binom{5-1}{3-1} = 6$ keys for each participant.

Maximum unauthorised access structure Γ^+ / Participants	P_1P_2	P_1P_3	P_1P_4	P_1P_5	P_2P_3	P_2P_4	P_2P_5	P_3P_4	P_3P_5	P_4P_5
P_1	Φ	Φ	Φ	Φ	|	|	|	|	|	|
P_2	Φ	|	|	|	Φ	Φ	Φ	|	|	|
P_3	|	Φ	|	|	Φ	|	|	Φ	Φ	|
P_4	|	|	Φ	|	|	Φ	|	Φ	|	Φ
P_5	|	|	|	Φ	|	|	Φ	|	Φ	Φ

Φ not assigned | assigned.

Fig 3.1.1.1: Table showcasing the assignment of elements(keys) in Γ^+ to participants when $t = 3, n = 5$.

4.1.2 Implementation of CA

We attempted to carry out an implementation of a secret sharing scheme using cumulative arrays with Python.

4.1.2.1 Constructing Maximal Unauthorised Subsets

Firstly, with (t, n) as input, we construct a maximal unauthorised subset. The number of locks will be reflected as the length of the maximal unauthorised subset. (Appendix 4.1.2.1)

4.1.2.2 Generating a Cumulative Array of Shares and Distribute Them to Participants

Next, generate a share for each element in the maximal unauthorised subset at the range of 1 to a prime number p which is larger than the length of the maximal unauthorised subset. However, for the last share, it will be calculated from the sum of all previous shares and secret s, mod p. (Appendix 4.1.2.2.1)
Then, assign the elements in the maximal unauthorised subset to participants. If the participant p_i is in the the element, the key will not be assigned to p_i. (Appendix 4.1.2.2.2)

4.1.2.3 Recovering the secret

To recover the secret, we require a list of participants who will contribute to unlock the locks. As each participant will have a certain number of shares, we will combine them. As we have constructed the cumulative array, the last element is the sum of the cumulative array and secret s. Hence, we reverse the process, if the result is the same as the secret s, the secret is recovered. (Appendix 4.1.2.3)

4.1.3 Limitation and Possible Application of CA

The size of each share of a secret sharing scheme constructed from a cumulative array is generally bigger than the one in an ideal scheme, in which the size of each share is the same as the size of the secret. (Beimel, Tessa, and Weinreb 2005) The sizes of shares increase exponentially to the size of secrets at worst. Thus cumulative arrays have not been researched that much in the field of secret sharing. For any t and n, there is a general solution (San Ling, 2013) using only AND locks as shown in the above example, which requires $\frac{n!}{(t-1)!(n-t-1)!}$ locks and $\frac{(n-1)!}{(t-1)!(n-t)!}$ keys. However, as the t and n, becomes larger, their resultant number of keys and locks also increase. In the seminal paper on secret sharing (Shamir, 1979) mentioned above, it stated that it is not hard to show that the minimal solution uses 462 locks and 252 keys per scientist in the case of $(6,\ 11)$. This is simply too much for each scientist to carry around.

4.2 General Secret Sharing using Generalised Cumulative Array (GCA)

(Shoulun Long, 2006) gave another solution to show that it is possible to reduce the numbers of locks and keys by using both AND and OR combination as shown in the following figure. This method uses the theory of a Perfect Hash Family (PHF).

4.2.1 Explanation of GCA

Let Γ be an access structure on P. A generalised cumulative array (GCA) $(\tau_1, \tau_2,...,\tau_l|X_1, X_2,..., X_l)$ for Γ is a collection of finite sets $X_1, X_2,..., X_l$, where each X_i is associated with a mapping τ_i: $P \rightarrow 2^{X_i}$, such that for $I \subseteq$ P we have

$$\bigcup_{P \in I} \tau_i(P) = X_i \text{ for some } i\ (1 \leq i \leq l),\ \text{if and only if } I \in \Gamma.$$

Secret sharing schemes for Γ can be realised from generalised cumulative arrays in much the same way as from a cumulative array. Again, we assume that the secret set K, is an abelian group and $X_i = \{X_1^{(i)}, X_2^{(i)},.., X_{d_i}^{(i)}\}$. To share a secret $k \in$ K, the dealer randomly and independently chooses l vectors, each consisting of d_i elements $k_1^{(i)}, k_2^{(i)},.., k_{d_i}^{(i)} \in$ K such that $k = k_1^{(i)}, k_2^{(i)},.., k_{d_i}^{(i)}$, for each $1 \leq i \leq l$, and distributes shares to participants according to $(\tau_1, \tau_2,...,\tau_l|X_1, X_2,..., X_l)$. That is, the share of participant P_t is $\cup_{i=i}^{l}\{k_j^{(i)} | \ if\ x_j^{(i)} \in \tau_i(P_t)\}$. Then, obviously, the shares of participants from $I \in 2^P$ will contain one of the components $k_1^{(i)}, k_2^{(i)},.., k_{d_i}^{(i)}$ for some i if and only if $I \in \Gamma$.

4.2.1.1 Generalised cumulative arrays from perfect hash families

Perfect hash families can be used to construct GCAs, which was suggested in (Martin et al. 2005)

Let $A = \{1, 2,..., n\}$, $B = \{1, 2,..., m\}$ and H be a set of functions from A to B. We say that H is an (n, m, t)-perfect hash family if for any subset $I \subseteq A$ with $|I| = t$, there exists at least one element $h \in H$ such that H restricted to I is one-to-one. When $|H| = l$, we refer to the triple (A, B, H) as a $PHF(l; n, m, t)$. When $m = t$, we call it a minimal perfect hash family.

Given a $PHF(l; n, t, t)(A, B, H)$, we can construct a GCA as follows. Let $H = \{h_1, h_2,..., h_l\}$. For each , h_l we associate a t-set $X_i = \{x_1^{(i)}, x_2^{(i)},.., x_t^{(i)}\}$ and define a function τ from P to 2^{x_i} such that $\tau_i(P_j) = \{x_{h_i(j)}^{(i)}\}$. Since for any t subset of P there exists an i such that h_l restricted to it is one-to-one and less than t participants in P together are associated with at most $t - 1$ elements in X_i, for all $1 \leq i \leq l$ (note that each participant will get only one element from each X_i), it is easy to see that $(\tau_1, \tau_2,...,\tau_l|X_1, X_2,..., X_l)$ is indeed a GCA for the (t, n) threshold access structure.

We employ the following algorithm for perfect hash families due to (Atici et al. 1996)

Let $A = \{1, 2,..., n\}$ and $B = \{1, 2,..., t\}$. For a subset $I = \{x_1, x_2, \dots, x_t\} \subseteq A$, where $x_1 < x_2 < \cdots < x_t$, define a function h_I as follows:

$$h_I(x) = \begin{cases} 1 & \text{if } x < x_2 \\ 2i - 1 & \text{if } x_{2(i-1)} < x < x_{2i}, \text{ for some } i = 2, 3, \dots, \lfloor \frac{t}{2} \rfloor \\ 2i & \text{if } x = x_{2i} \text{ for some } i = 1, 2, \dots, \lfloor \frac{t}{2} \rfloor \\ t & \text{if } x \geq x_t. \end{cases}$$

x_l / Participant	$x_1 = h_1 = \{1^1, 2^1, \dots, t^1\}$	$x_2 = h_2 = \{1^2, 2^2, \dots, t^2\}$	...	$x_l = h_l = \{1^l, 2^l, \dots, t^l\}$
P_1	$3^1 x^{(1)}$	$2^2 x^{(2)}$	...	$3^l x^{(l)}$
P_2	$t^1 x^{(1)}$	$1^2 x^{(2)}$	...	$4^l x^{(l)}$
. .	. .	. .	. .	. .
P_n	$1^1 x^{(1)}$	$4^2 x^{(2)}$	...	$t^l x^{(l)}$

Fig 3.2.1.1: An example of how the GCA will work with PHF

It has been shown that $H = \{h_I : I \subseteq \{1, 2,..., n\}, |I| = t\}$ is a $PHF(l; n, t, t)$, where $l = \binom{n-\lceil \frac{t}{2} \rceil}{\lceil \frac{t}{2} \rceil}$. Taking $n = 11, t = 6$, we obtain a $PHF(56; 11, 6, 6)$. If we apply the above perfect hash family to the construction of GCA, we have a GCA $(\tau_1, \tau_2,..., \tau_l, X_1, X_2,..., X_l)$ with $l = 56$ and $|X_i| = 6$ for all $1 \leq i \leq l$,which then results in a solution (small modification to the way of locks, that is, the cumulative array solution uses only AND locks while the GCA solution takes both AND and OR locks), requiring only $\binom{11-\lceil \frac{6}{2} \rceil}{\lceil \frac{6}{2} \rceil} = 336\ locks$ and $l = 56\ keys$ for each participant.

4.2.2 Implementation of GCA using perfect hash families

We attempted to carry out an implementation of a secret sharing scheme from GCA with Python, making use of perfect hash families from algorithms proposed by (Atici et al. 1996)

4.2.2.1 Generating a set of hash functions

Firstly, H, a set of hash functions will be generated using the algorithm above, by generating all the possible combinations of sets consisting of t numbers which are all the elements of $A(1, 2,.., n)$. Looking at the

algorithm, the hash functions will only depend on the the even x value x_2, x_4, $x_6$and so on. Hence, those hash functions with exact same even x values will be considered repeated and can be removed.(Appendix 4.2.2.1)

4.2.2.2 Hashing the elements in X_1, X_2,..., X_t and distributing them to the participants

Next, we will assign a share mapped by each hash function to a participant. In total, each participant will receive *I* (number of hash functions) numbers of keys. (Appendix 4.2.2.2)

4.2.2.3 Recovering the secret

In order to recover the secret, we will need to combine the shares of all participants and if one of the blocks the value perfectly match that of set $B(1,\ 2,..,\ t)$, the secret can be recovered. (Appendix 4.2.2.3)

5. Results

As shown by the results (Appendix 5.1), generally, assuming the t and n numbers are $\leq\ 10$, using the GCA method will guarantee a lower number of locks and a lower number of keys than just the CA method.
However, if the $t\ =\ 2$, the number of locks using the GCA method is higher than when using the CA method while the number of keys remains the same. The following table shows the complete data for both methods.

6. Conclusion

The Generalised Cumulative Arrays (GCA) can be used for cases in which the t is greater than 2 as it performs better (in terms of lower locks and keys) as compared to just the cumulative arrays method. On the other hand, if $t\ =\ 2$, we recommend using the Cumulative Arrays (CA). Due to limited open source information on the internet on the GCA and CA, these computational implementations of the GCA and CA methods can be a source of reference for usage in real world applications such as threshold cryptography, block ciphers, etc.

7. Appendix

5.1 results

Number of locks and keys for each (t, n) case using the CA and the GCA method for general secret sharing

t	n	Number of locks using CA	Number of keys each participants have to hold using CA	Number of locks using GCA	Number of keys each participation have to hold using GCA
2	3	3	2	4	2
2	4	4	3	6	3
2	5	5	4	8	4
2	6	6	5	10	5
2	7	7	6	12	6
2	8	8	7	14	7
2	9	9	8	16	8
2	10	10	9	18	9
3	4	6	3	3	1
3	5	10	6	9	3
3	6	15	10	18	6
3	7	21	15	30	10
3	8	28	21	45	15
3	9	36	28	63	21
3	10	45	36	84	28
4	5	10	4	12	3
4	6	20	10	24	6
4	7	35	20	40	10
4	8	56	35	60	15

4	9	84	56	84	21
4	10	120	84	112	28
5	6	15	5	5	1
5	7	35	15	20	4
5	8	70	35	50	10
5	9	126	70	100	20
5	10	210	126	175	35
6	7	21	6	24	4
6	8	56	21	60	10
6	9	126	56	120	20
6	10	252	126	210	35
7	8	28	7	7	1
7	9	84	28	35	5
7	10	210	84	105	15
8	9	36	8	40	5
8	10	120	36	120	15
9	10	45	9	9	1

4.1.2.1

```
import itertools
def generate_max_unauthorised_subset(t, n):
    all_numbers = [] #to store all the numbers from 1 to n
    for j in range(1, n + 1):
        all_numbers.append(j)
    MUS=[]
    #generate a maximal unauthorised set by generating all the possible
    #combinations of sets consisting of t - 1 numbers ranging from 1 to n.
    for i in itertools.combinations(all_numbers, t-1):
        MUS.append(i)
    return MUS
```

4.1.2.2.1

```
import random
def generate_shares(p, s, width):
#p is a prime number larger than width which is the length of the
#unauthorised subset and s is the secret in the form of integer
    CArray = []
    for i in range(width - 1):
        k = random.randint(1, (p-1))#generate shares as random number 1-p
        while k in CArray or k == s:
            k = random.randint(1, (p-1))
        CArray.append(k)
    #the last share is the sum of previous shares and s, mod p
    total = 0
    for n in CArray:
        total += n
    total += s
    total = total % p
    CArray.append(total)
    return CArray
```

4.1.2.2.2

```
def generate_map(MUS, n, shares):
    maps = []
    for P in range(1, (n+1)):
        PArray = ""
        for element in MUS:
            if P in element:
                PArray += "0" #do not assign
            else:
                PArray += "1" #assign
                keyCount += 1
        maps.append(PArray)
    P = []
    for map in maps:
        p = []
        for i in range(len(shares)):
            k = shares[i] * int(map[i])#distribute the shares
            p.append(k)
        P.append(p)
    return P #this is a list of {p_1, p_2, … p_n} with assigned shares
```

4.1.2.3

```
def recover(secretMap, width, p, s, participants):
    recoveredShares = [0] * width
    for i in participants:
        share = secretMap[i-1]
        for j, k in enumerate(share):
            if(k != 0):
                recoveredShares[j] = k #to replace already unlocked locks
    sumK = recoveredShares[-1]
    for i in range(len(recoveredShares) - 1):
        sumK -= recoveredShares[i]
        if(sumK < 0):
            sumK += p
    if(sumK != s):
        print("Recovered secret incorrect.")
    else:
        print("Correct! The secret is {}.".format(s))
```

4.2.2.1

```
import itertools
def get_H(A, t):
    H=[]
    for i in itertools.combinations(A, t):
        H.append(i)
    #since the hash only depends on x2, x4, x6, we will not
    #consider those subsets that have exact values for even x
    i = 0
    while(i < len(H)):

        k = i + 1
        while(k < len(H)):
            same = True
            j = 1
            while(j < t):
                if(H[i][j] != H[k][j]): same = False
                j+=2
            if same == True: H.pop(k)
            else: k+=1
        i+=1
    return H
```

4.2.2.2

```
def generate_shares(H, t, n):
    #generate a list of hash funcions
    maps = []
    for h in H:
        maps.append(Hash(h, t))
    P = []
    for i in range(1, n+1):
        shares = []
        for map in maps:#assign a key from every hash to a participant
            shares.append(map.hash(i))
        P.append(shares)
    return P
class Hash:
    def __init__(self, I, t):
        self.I = I
        self.t = t
    def hash(I, x, t):
        #implementation of the algorithm by Aciti et al
        if(x < self.I[1]): return 1
        elif(x >= self.I[self.t-1]): return self.t
        else:
            for i in range(1, int(self.t/2) + 1):
                if(i >= 2 and x > self.I[2*(i-1) - 1] and x < self.I[2*i -1]):
                      return 2 * i  - 1
                elif(x == self.I[2*i - 1]): return 2*i
```

4.2.2.3

```
def recover(participants, P, l):
    #combine the shares of all participants
    recoveredShare = []
    for j in range(l):
        x = []
        for i in participants:
            x.append(P[i-1][j])
        recoveredShare.append(x)
    return recoveredShare
def unlock(recovered, B):
    for x in recovered:
        unlock = True
        if len(x) < len(B): return False
        else:
            for i in range(len(B)):
```

```
                if x[i] != B[i]: unlock = False
            if unlock:
                print("Secret recoverd!")
                return True
    return False
```

8. Bibliography

Works Cited

Atici, M., S. S. Magliveras, D. R. Stinson, and WD Wei. 1996. “Some recursive constructions for perfect hash families.” Wiley Online Library. https://doi.org/10.1002/(SICI)1520-6610(1996)4:5<353::AID-JCD4>3.0.CO;2-E.

Beimel, Amos, Tamir Tessa, and Enav Weinreb. 2005. “Characterizing Ideal Weighted Threshold Secret Sharing.” Springer Link. https://doi.org/10.1007/978-3-540-30576-7_32.

Blakley, G. R., and Gregory Kabatiansky. 2011. “Secret Sharing Scheme.” Springer.

Brickell, Ernie, Giovanni D. Crescenzo, and Yair Frankel. 2000. “Sharing Block Ciphers.” Springers Link. https://link.springer.com/chapter/10.1007/10718964_37.

Curik, Peter, Roderik Ploszek, and Pavol Zajac. 2022. “Practical Use of Secret Sharing for Enhancing Privacy in Clouds.” MDPI. https://www.mdpi.com/2079-9292/11/17/2758/pdf.

CyberMagazine. 2022. “A heist to remember: Three scandals that remain unsolved.” Cyber Magazine. https://cybermagazine.com/articles/a-heist-to-remember-three-scandals-that-remain-unsolved.

Desmedt, Yvo, Rei Safavi-Naini, and Huaxiong Wang. 2002. “Redistribution of Mechanical Secret Shares.” Springer. https://link.springer.com/chapter/10.1007/3-540-36504-4_17.

Gharat, Anamika. 2022. “12-cr Mumbai bank heist was an insider's job; accused arrested: Cops.” Hindustan Times. https://www.hindustantimes.com/cities/mumbai-news/12cr-mumbai-bank-heist-was-an-insider-s-job-accused-arrestedcops-101664887455899.html.

Ito, M., A. Saito, and T. Nishizeki. 87. “Secret Sharing Scheme realising general access structure.” *GLOBECOMM* 87:99-102. https://archiv.infsec.ethz.ch/education/as09/secsem/papers/ItSaNi87.pdf.

Lemnouar, Noui. 2022. “Security limitations of Shamir’s secret sharing.” tandf. https://doi.org/10.1080/09720529.2021.1961902.

Long, Shoulun, Josef Pieprzyk, and Huaxiong Wang. 2006. “Generalised Cumulative Arrays in Secret Sharing.” Springer Link. https://link.springer.com/article/10.1007/s10623-006-0007-5.

Martin, Keith M., Rei Safavi-Naini, Huaxiong Wang, and Peter R. Wild. 2005. “Distributing the Encryption and Decryption of a Block Cipher.” Springer Link. https://link.springer.com/article/10.1007/s10623-003-1719-4.

Shamir, Adi. 1979. “How to Share a Secret.” MIT. https://web.mit.edu/6.857/OldStuff/Fall03/ref/Shamir-HowToShareASecret.pdf.

Wang, Huaxiong, and Josef Pieprzyk. 2003. “Shared generation of pseudo-random functions with cumulative maps.” Macquaire University. https://researchers.mq.edu.au/en/publications/shared-generation-of-pseudo-random-functions-with-cumulative-maps.

Wen-Ai, Jackson, and Keith M. Martin. 2005. “Cumulative arrays and geometric secret sharing schemes. In Advances in Cryptology.” Springer Link. https://doi.org/10.1007/3-540-57220-1_51.

Xing, Chaoping, San Ling, and Huaxiong Wang. 2019. *Algebraic Curves in Cryptography*. N.p.: CRC Press.