Concurrent distributed storage protocols

Using secret sharing to create a byzantine regular register

Master Thesis

Marco Tobia Cacciatore

Faculty of Science
at the University of Bern

February 2023

Prof. Dr. Christian Cachin
David Lehnherr

Cryptology and Data Security Research Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

In a ($t + 1$, N) secret sharing scheme, a secret is divided into N shares and shared among N shareholders, such that any $t + 1$ or more shareholders can reconstruct the secret, but any less cannot gain any information about the secret. We show that a standard ($t + 1$, N) secret sharing scheme works only in a $N > 3f + 1$ model and needs to be adapted to work in $N > 3f$, because in a $N > 3f$ model, a process may safely wait for $2f + 1$ responses out of which f may be faulty and thus is missing one share to have the minimum of $f + 2$ shares to verify a reconstructed secret. In order to solve this, we introduce the approach of the distributed additional share, where we create $N + 1$ shares from the secret and secret share the $(N + 1)^{st}$ share. With this method, we preserve information-theoretical security and require only $N > 3f$. We also define the cloud-of-clouds model, where cloud storage services are combined, in a modular way with clear properties, formally prove them and implement a safe and regular register using our technique. Finally, we compare our algorithms to two existing systems.

Acknowledgements

I thank David Lehnherr for all the help, critical thinking, and support. I would also like to thank Prof. Dr. Christian Cachin for the insights, for pointing the thesis in the right direction, and for giving me the chance to work on this project. In addition, I thank the whole Cryptology and Data Security Research Group as well as the other students in the office for interesting coffee breaks and discussions.

Contents

1 Introduction

A $(t+1, N)$-secret sharing scheme is a method for distributing a secret among N participants such that any $t+1$ of these participants can combine their shares to reconstruct the secret. Any group of participants with a lower cardinality will not gain any information about the secret. Secret sharing is often used in applications where security and confidentiality are of the utmost importance, such as encryption or decryption keys, missile launch codes, or updates for DNS root servers. In a byzantine setting, where an adversary may control up to f participants, it is ensured, that these adversaries cannot gain any information about the secret, as long as $t \geq f$ (c.f. Shamir [Sha79]). To prevent adversaries from providing forged shares that would result in the reconstruction of an incorrect secret, the shares are often digitally signed before being distributed. However, according to Padilha and Pedone [PP11], this can be complex and introduce a significant overhead for large secrets and many participants. Harn and Lin [HL09] proposed to take more than $t+1$ shares, reconstruct all combinations, and if a majority of the reconstructed secrets are the same, it can be concluded with high probability, that the reconstructed secret is correct. Padilha and Pedone [PP11] used this method to obfuscate parameters of operations in a state-machine-replication system (SMR). Their system is a key-value storage system that allows byzantine faults and provides confidentiality through secret sharing.

Cloud-of-clouds. Companies like Amazon, Microsoft, Dropbox, or Google offer cloud storage services, i.e. Amazon S3, OneDrive, Dropbox, and Google Drive, that are very popular to store everyday pictures, personal files, or when used by companies, work files. Since the stored data is outside of the customer's premises, the customers have concerns about integrity, availability, and confidentiality (c.f. Rocha and Correia [RC11] and Alliance [Clo13]). In their Service Agreements or Terms of Service, those companies allow themselves to either directly scan your content or, only in case of a *suspicion* of illegal activities, access data without notifying the customer [Ama22] [Mic22] [Dro22] [Goo22]. This means that they have full access to all the stored data and thus can read, modify and delete stored data, all without an automatic notification to the customer. The Cloud-of-clouds approach combines several cloud storage systems to tolerate faults and improve certain properties. By storing data throughout different cloud storage services, integrity and availability are guaranteed, but confidentiality is usually achieved with encryption. Since a perfectly secret encryption scheme needs a key space as large as the plaintext contents, the key space would need to be as big as the files themselves, which would be unusable in practice (c.f. Katz and Lindell [KL14]).

There are a number of different systems and frameworks that have been proposed to implement byzantine fault-tolerant storage systems in a cloud-of-clouds model or similar architectures. Some notable examples include systems like AVID [CT05], Belisarius [PP11], Charon [MOC$^+$21], DepSky [BCQ$^+$13], HAIL [BJO08], ICStore [CHV10], NCCloud [CHLT14], and RACS [APW10]. While the models in NCCloud and RACS assume that some servers may crash, the models in AVID, ICStore, Charon, HAIL, DepSky, and Belisarius additionally assume the existence of an adversary and provide a form of fault tolerance against attacks. To provide redundancy, Belisarius uses secret sharing, while all other mentioned systems use some form of erasure coding.
In this thesis, we present secret-shared registers using the method of Harn and Lin [HL09]. According to Harn and Lin [HL09] in order verifiably reconstruct a secret-shared value, $f+2$ *correct* shares are required. This means that

during a read operation, a process must wait for $2f + 2$ responses, as up to f shares may be from an adversary and thus be faulty. In a wait-free implementation, a read operation may wait for up to $N - f$ responses, because up to f responses may be from *byzantine* servers, which might not respond. However, if the model requires $N > 3f$, this can present a problem, since when $N = 3f + 1$, then $N - f = 2f + 1$, but $2f + 2$ shares are needed. To address this issue, we have developed a method that allows for $2f + 2$ shares to be obtained from $2f + 1$ responses. Using this method, we have created a safe and a regular register that provide integrity, availability, and confidentiality in a cloud-of-clouds model. In addition to developing a secret-shared register, we have also formalized the API to cloud services and defined the properties of the connection and behavior of the servers. This general formalization of APIs makes it easy to implement further algorithms that utilize any type of client-server model and allows for properties of those algorithms to be easily proven because the underlying connection and server behavior are well-defined. Any specific aspects that need to be adapted can be done by adjusting existing properties or adding new ones. For example, if an algorithm only supports conditional security, the *unconditional encryption* property, which provides unconditional security for communication between clients and servers, can be adapted to use conditional security instead. Furthermore, we have evaluated our registers against two other storage systems that provide confidentiality in the cloud-of-clouds environment.

Thesis organization. We will explain the topics that are necessary to the rest of the paper in the related work chapter. In chapter 3, we define our model, introduce our secret sharing scheme, the modules we need for the modular construction, and provide pseudo-code of our solutions. In the fourth chapter, we prove that our algorithms meet properties and compare them to existing systems. Finally, we summarize our findings in the conclusion chapter.

2 Related work

In this chapter, we explain previous research that is necessary for our thesis. We outline how the secret sharing scheme of Shamir [Sha79] works in detail, how Harn and Lin [HL09] can detect and identify *faulty* shares with unconditional security, and how detection and identification of *faulty* shares can be done with conditional security. After which we show the definition of conditional and unconditional security as defined by Diffie and Hellmann [DH76]. Then we summarize what registers are and how they work as in Cachin et al. [CGR11], Lamport [Lam86] and Herley and Shavit [HS08]. Finally, we introduce two byzantine fault-tolerant storage systems developed by Padilha and Pedone [PP11] and Bessani et al. [BCQ+13].

2.1 Shamir's secret sharing

Shamir's secret sharing (c.f. Shamir [Sha79]) is based on polynomial interpolation. A polynomial of degree t is of the form $f(x) = a_0 + a_1x + ... + a_tx^t$. The parameter a_0 is chosen to be the secret and a_1 to a_t are chosen randomly from a finite field $\mathbb{F}_p$,which is defined by a prime number p, that is higher than the number of generated shares and every parameter a_i, thus the secret is $s = a_0 = f(0)$. Next, for each participant, x-values are chosen randomly from $\mathbb{F}_p$ or $x_i = i$ for each participant P_i and the coordinates $(x, f(x)\ mod\ p)$ are given out as shares. The modulo is used to enforce the y-value to be within $\mathbb{F}_p$ and thus any later reconstruction that would lead to a $f(0)$ value outside of $\mathbb{F}_p$ cannot be assumed as an invalid solution and thus it is inhibited that some information about the secret or the shares can be gained with t or less shares. Only the degree t and the field $\mathbb{F}_p$ need to be known for the polynomial $f(x)$ as well as the secret $f(0)$ to be reconstructed with any $t + 1$ shares. Since there are $t + 1$ unknown parameters $a_0, ..., a_t$, at least $t + 1$ shares (x_i, $f(x_i)$) are needed, such that there exists an equation $f(x_i) = a_0x_i + ... + a_tx_i$ for each unknown. If there were less than $t + 1$ shares, there would exist an underdetermined equation system with infinitely many solutions. To be exact, all values in $\mathbb{F}_p$ would be viable parameters for the secret a_0, for the equation system to hold. With exactly $t + 1$ shares, there exists a system of equations with the same amount of unknowns and equations and thus has a single solution for $f(x)$, with $f(0) = a_0 = s$. If more than $t + 1$ shares are taken, an overdetermined equation system exists, with also a single solution for $f(x)$, since every further share than the $(t + 1)^{st}$ stems from the same polynomial and hence is a linear combination of the other equations.
A further approach to reconstruct the secret is the Lagrange interpolation, where for each share (x_i, y_i) a basis polynomial b_i is created that, such that the basis polynomial b_i is one for its share x_i and zero for each other share: $b_i(x_i) = 1 \wedge \forall x_j \neq x_i : b_i(x_j) = 0$. The polynomial f is then the linear combination of the basis polynomials b_i and their corresponding y values y_i as in (2.1). Hence, $f(x_i) = y_i \cdot b_i$ holds for each share, since all $b_j = 0, j \neq i$. The Lagrange interpolation will return the polynomial $f'(x)$ that satisfies all used shares and assuming at least $t + 1$ shares have been used, the degree of the polynomial will be t and $f'(x) = f(x)$. Checking if a further share (x_j,y_j) belongs to the same set of shares follows immediately. First, $f(x)$ is reconstructed with at least $t + 1$ shares, no

matter the method, and then it can be checked whether $y_j = f(x_j)$ holds.

$$f'(x) = \sum_{i=0}^{t} y_i \cdot l_i(x) \mod p \text{, where } l_i(x) = \prod_{0 \le m \le k, m \ne i} \frac{x - x_m}{x_i - x_m} \tag{2.1}$$

Since the secret is at $x = 0$, x can be eliminated in the formula and the formula is simplified to:

$$f'(0) = \sum_{i=0}^{t} y_i \cdot \prod_{m=0, m \ne i}^{t} \frac{-x_m}{x_i - x_m} \mod p$$

2.1.1 Shamir's secret sharing in a byzantine model

In a byzantine model, participants can behave arbitrarily and send *faulty* shares that do not belong to the same polynomial as the *correct* ones. When using at least one *faulty* share for the reconstruction of the polynomial $f(x)$, the reconstructed polynomial $f'(x)$ will differ from $f(x)$ and thus $s' = f'(0) \ne f(0) = s$. According to Harn and Lin [HL09], when more than $t + 1$ shares are used for reconstruction and there is at least one *faulty* share, the resulting polynomial has a higher degree than t. However, Ghodosi [Gho11] has shown that this is not always the case and only holds with high probability. We will show his wise cheating attack in section 3.2. In short, if the *faulty* shares are the sum of the original polynomial $f(x)$ and another polynomial $g(x) \ne 0$ and $g(x_i) = 0$ for all x_i of the *correct* shares, then the reconstructed polynomial will be $h(x) = f(x) + g(x) \ne f(x)$, due to the homomorphic property of Shamir's secret sharing. Importantly, the inverse of Harn and Lin's [HL09] discovery is also true. Meaning that, when more than $t + 1$ shares are used and the degree of the reconstructed polynomial is t, then all used shares are *correct* with high probability [Har13]. Since a polynomial is defined with $t + 1$ shares and for every x value, there exists exactly one corresponding y value $y = f(x)$, the degree of a polynomial with more than $t + 1$ shares can only be t if all further shares than the $(t + 1)^{st}$ share are a linear combination the other ones, i.e. all shares were created with the same polynomial and are *correct*. Hence, the detection of *faulty* shares can be done by using more than $t + 1$ shares and checking the degree of the resulting polynomial as can be seen in figure 2.1.3 of example 2.1.

Example 2.1. Assuming the secret is $s = 0$ and there is one *faulty* participant P_3. First, a polynomial of degree t is created with random coefficient $a_1 = 1$, i.e. the polynomial is $f(x) = a_0 + a_1 x = x$. Next, the shares $(x_i, f(x_i))$ are transmitted to the participants P_i for x_i in the range $[1, N]$, with $N > 3$. Note that instead of x_i being picked randomly, the index of the participants is used for illustration purposes. Further, it is assumed that for reconstruction, the shares (1,1), (2,2), (3,1) are received as illustrated in figure 2.1.1. The presence of faulty shares can be checked by recreating and comparing the three functions $f'_0(x)$, $f'_1(x)$ and $f'_2(x)$ generated by two shares each and observing that they are different as in figure 2.1.2. Alternatively, a function $f_1(x)$ can be recreated with all three shares and it is seen that the degree is 2 instead of 1 as shown in figure 2.1.2. When a fourth share (4,4) is received, as in figure 2.1.3, and more than $t + 1$ correct shares are available, four functions $f_0(x)$, $f_1(x)$, $f_2(x)$ and $f_3(x)$ can be recreated and it is observed that only $f_0(x)$ has degree 1 and thus the secret, $f_0(0) = 0 = s$, is verified.

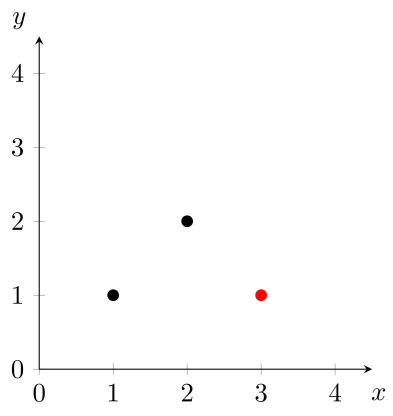

Figure 2.1.1: $t + 1$ *correct* shares

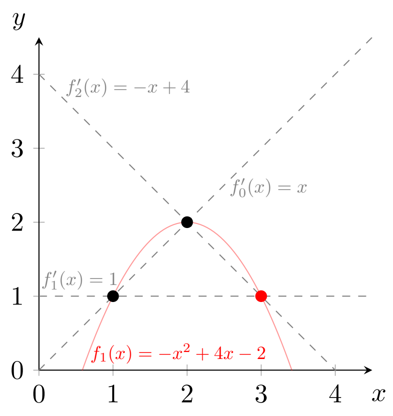

Figure 2.1.2: reconstructed functions

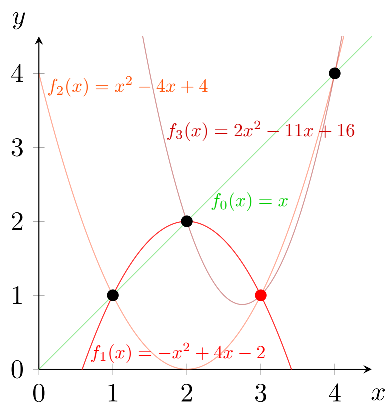

Figure 2.1.3: reconstructed functions with $t + 2$ *correct* shares

When allowing *byzantine* behavior in a permissioned setting, where all participants are known and need to be authenticated, identification of *byzantine* actors and the revocation of their permissions is important. The process of identifying *byzantine* participants is also called cheater identification. Harn and Lin [HL09] differentiate between cheater/*byzantine* detection and cheater/*byzantine* identification, where the detection shows the presence of at least one *byzantine* share in a set of shares and the identification reveals the *byzantine* share. In verifiable secret sharing protocols, where each share can be verified individually, the *byzantine* identification is done by verifying each received share, which we will explain in section 2.1.2. Harn and Lin [HL09] proposed an algorithm for *byzantine* detection where they reconstruct the polynomial f_{all} from all received shares and if the degree of f_{all} is not t, then there are *byzantine* shares. Their proposed algorithm for *byzantine* identification reconstructs all possible secrets from any subset of shares with size $t + 1$ and defines the secret that occurs the most often as the *correct* secret. Next, they take any t shares from any reconstruction of the *correct* secret and one share s_j, that was not used in any reconstruction of the *correct* secret. If the resulting secret does not match the *correct* secret, then s_j is *byzantine*.

2.1.2 Verifiable secret sharing

Another way of solving the problem of byzantine shares is verifiable secret sharing (VSS) (c.f. Katz and Lindell [KL14]). We explain verifiable secret sharing using Feldman's method [Fel87]. When a client C starts a *write*, first $a_0, ..., a_t$, including a_0, are picked uniformly random from $\mathbb{F}_p$ to define the polynomial $f(x) = \sum_{j=0}^{t} a_j \cdot x^j$. Then, x_i are randomly chosen from $\mathbb{F}_p$ for each server P_i and securely each P_i is sent their share $(x_i, f(x_i))$. Next C broadcasts the masked secret $c := H(a_0) \oplus s$ and $A_0 = g^{a_0}, ..., A_t = g^{a_t}$, for a publicly known prime g, with H being a one-way hash function in $\mathbb{F}_p$. Finally, a share can easily be verified to be correct by the following formula:

$$\begin{aligned} g^{f(x_i)} &= \prod_{j=0}^{t} (A_j)^{x_i^j} \\ &= \prod_{j=0}^{t} (g^{a_j})^{x_i^j} \\ g^{f(x_i)} &= g^{\sum_{j=0}^{t}(a_j)\cdot x_i^j} \\ \implies f(x_i) &= \sum_{j=0}^{t} a_j \cdot x_i^j \end{aligned} \tag{2.2}$$

When a client D now reads, each share is checked with (2.2), all *faulty* shares are discarded and $f(0) = a_0$ is reconstructed as in the Shamir's Secret Sharing. Finally, a_0 is hashed and the secret is reconstructed with $s = c \oplus H(a_0)$. With VSS, anyone with access to a share and the publicly known information can verify a share and when all shares that are used for the reconstruction are verified, the reconstructed secret must be *correct* [KL14]. The security of VSS is twofold. First, it is assumed, that the discrete-logarithm problem is hard and secondly, that the hash function H provides the properties explained in definition 2.2.

Definition 2.1 (Discrete-Logarithm Problem)**.** In a cyclic group $G = \{g^i | i = 0, ..., p-1\}$, where p is a prime, the discrete logarithm problem is defined as computing i, such that $g^i = y$ for a randomly chosen y. It is assumed, that this is hard for groups with $|p| = 2048$, i.e. p is in the order of 2^{2048}.

Definition 2.2 (One-way hash function)**.** A one-way hash function $H : \{0,1\}^* \to \{0,1\}^k$ converts a variably sized bit-string into a fixed-sized output that provides *collision resistance*, *pseudo-randomness* and two types of *pre-image resistance*. *Collision resistance* guarantees computational infeasibility for finding distinct m and m', with $H(m) = H(m')$. *Pseudo-randomness* guarantees that the output of H appears to be statistically random, i.e. given a hash $h = H(m)$ and random bytes of the same length, the probability of categorizing them correctly is $\frac{1}{2}$. The *1st pre-image resistance* guarantees computational infeasibility for a given hash h to find a message m, that satisfies $H(m) = h$. The *2nd pre-image resistance*, guarantees computational infeasibility for a given message m to find m', where $H(m) = H(m')$.

2.2 Conditional vs. unconditional security

The most prevalent security notion is perfect secrecy, which was defined by Shannon [Sha49]. It states that a system is perfectly secret if the probability of a ciphertext representing various messages is identical before an adversary

receives the ciphertext and after. Katz and Lindell [KL14] make the example that an adversary $\mathcal{A}$ might know that the encrypted message represents either "don't attack" or "attack tomorrow" and knows their probability. If $\mathcal{A}$ intercepts the ciphertext and its probability representing "don't attack" or "attack tomorrow" does not change, then the system is called perfectly secret. Such a system might be implemented with a one-time pad (OTP), where a message is encrypted with a key only once. The OTP, sometimes called Vernam's cipher, uses the bit-wise exclusive or ($\oplus$) operator, which results in a 1, if the input bits are different and 0 if the input bits are equal, i.e. $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$. In this cipher, a key k is chosen uniformly random from a bit-space with cardinality equal to 2^n, when n is the length of the message, and for encryption, as well as the decryption, the exclusive or operator is applied to the message or ciphertext and the key: Encrypt($k \in \{0,1\}^n, m \in \{0,1\}^n$) $= (k \oplus m) \in \{0,1\}^n$; Decrypt($k \in \{0,1\}^n, c \in \{0,1\}^n$) $= (k \oplus c) \in \{0,1\}^n$, with m being the message and c being the ciphertext. The decryption recovers the message because the exclusive or operator is symmetric and associative, a bit exclusively or-ed with itself results in 0 and 0 exclusively or-ed with any bit b results in b. Hence, Decrypt(k, Encrypt(k, m))=$k \oplus (k \oplus m) = (k \oplus k) \oplus m = 0 \oplus m = m$ for any $k, m \in \{0,1\}^n$.
Finally, the OTP is perfectly secret, because if the key k is unknown to $\mathcal{A}$, for each of the probable messages, there exists one key ($k_i = m_i \oplus c$), which would result in an intercepted ciphertext. For example, the probable messages are 011 and 101, and $\mathcal{A}$ intercepts the ciphertext 111. Then the key 100 could have been used for the message 011 ($100 \oplus 011 = 111$) or the key 010 could have been used for the message 101 ($010 \oplus 101 = 111$) and both of those keys are equally likely since the key is chosen uniformly random over $\{0,1\}^3$. Also, assuming the message was 011 and the key is unknown, all eight possibilities for the ciphertext are equally probable with probability $1/8$.
Diffie and Hellman [DH76] introduced two categories of security in cryptographic systems: conditional and unconditional security. They defined a conditionally secure system as a system, that is secure due to the computational cost of reversing the cryptographic functions, but that would succumb to an adversary with unlimited computational power. They defined unconditional security as a system that can resist any adversary, no matter how much computational power is used. An instance of an unconditionally secure system is a system with perfect secrecy since if a reveal of a ciphertext does not change the probability of it representing a specific message, the ciphertext alone does not contain any information and thus no cryptoanalytic attack can reveal any information about the message even under an attack with unlimited computational power. In this thesis, we focus only on conditional and unconditional security to have two security notions that are distinguishable by design.

Definition 2.3 (Conditional security [DH76])**.** A system is called conditionally secure, if it is secure, due to the computational cost of cryptoanalysis, but would succumb to an attack with unlimited computational power.

Example 2.2 (Conditional Security)**.** A hash function provides conditional security for the pre-image resistance. Assuming H is a hash function as in definition 2.2 and the digest length, i.e. length of the output hash, is 512 bits. Then the pre-image resistance (find m such that $H(m) = h$, for a given hash h) is conditionally secure. There are 2^{512} different hashes possible and thus in the worst case $2^{512} \approx 10^{154}$ messages need to be tried out, until a message m' has been found, such that $H(m') = h$. There are some optimizations to bring it down to $2^{500} \approx 10^{150}$ messages, which is still infeasible to compute within a reasonable time. Assuming every person on earth (≈ 8 billion people) work together and each person has a device, that has 10 billion cores, that each can compute 10 billion hashes per second, then it would take $\approx \frac{10^{150}}{8 \cdot 10^9 \cdot 10^{10} \cdot 10^{10}}$ seconds $\approx 10^{120}$ seconds $\approx 10^{113}$ years to compute all 2^{500} hashes. Thus, at least in the near future, the pre-image resistance is guaranteed and conditionally secure. Since it's theoretically possible (or with unlimited computational power) to find such a m', a hash function is not unconditionally secure.

Definition 2.4 (Unconditional security [DH76])**.** A system is called unconditionally secure, if it can resist any cryptoanalytic attack, no matter how much computation is allowed.

An example for an unconditionally secure algorithm is algorithm 3.2 (Theorem 4.7), which we will explain and prove in chapter 3.

2.3 Concurrency in registers

In distributed systems, processes often need to store and share data using *read* and *write* operations. To facilitate this, Cachin et al. [CGR11] use the concept of a register, which is an abstraction that allows processes to store and consistently retrieve data. The register abstraction is inspired by the functionality of registers in multiprocessor machines at the hardware level, but it can also be applied to other objects with similar functionality, such as disk drives accessed over a storage area network or collaborative editing files. Essentially, a register is a data storage mechanism that allows processes to store and retrieve values using the *write* and *read* operations. Registers are characterized by the number of readers and writers, where a (m, n)-register is a register with m writers and n readers. Usually, the values are either N, as in the number of processes, or 1, such that a $(1, 1)$-register is a single-writer-single-reader register (SWSR), a $(1, N)$-register is a single-writer-multi-reader register (SWMR) and a (N, N)-register is a multi-writer-multi-reader register (MWMR). One issue that can arise when using registers is the concurrency problem, which can occur when an operation is concurrent with another operation. In a single-writer-multi-reader register (SWMR), this can only occur, if a *read* is concurrent to a *write* and can lead to confusion about the correct value to return, as demonstrated in the following example:

Example 2.3. Assume that there are four processes $\{p, q, r, s\}$ as in figure 2.3.1 in a byzantine setting with process r being *byzantine* and q being slow. These processes each store a value and a naive protocol ties the processes together to form a naive approach to a wait-free register. A system is called wait-free if no operation requires responses from more than $N - f$ processes (c.f. Cachin et al. [CGR11]). If a client R now reads the stored value, while another process W writes a new value, there might occur a concurrency problem. In the example in figure 2.3.1, R received the old value from s, the new value from p, and a faulty value from r. Since the protocol is wait-free, it should be able to return a value with responses from three out of four processes. Though R can not decide which value is correct, since all three options are equally likely and only occur f times.

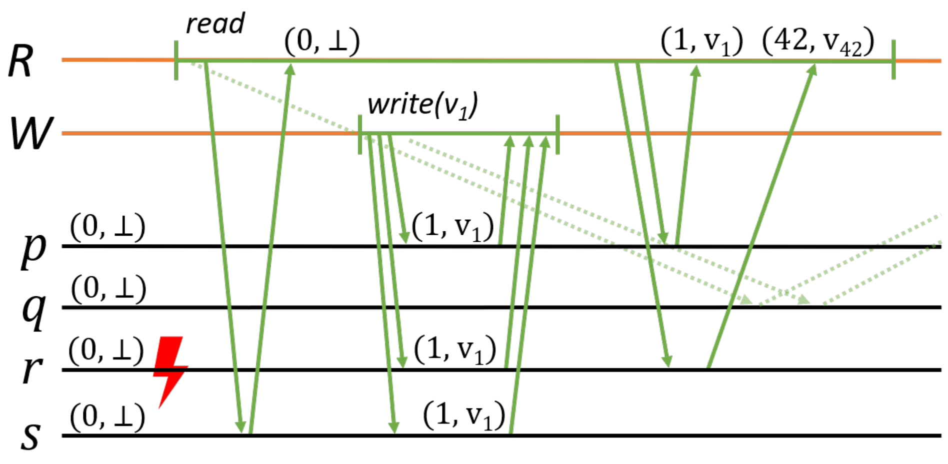

Figure 2.3.1: Concurrency problem for $f = 1$

To address the issue highlighted in example 2.3 and figure 2.3.1, there are several approaches that specify what a *read* of register should return, if the *read* is concurrent to one or more *writes*. In each case, a *read* operation that is not concurrent to a *write* operation returns the last value written (c.f. Lamport [Lam86] and Herlihy and Shavit [HS08]).

- **Safe Register.** A safe register guarantees for a read that is concurrent to a write, that the value will be in the domain of possible values. Thus the read value might not be a value that has ever been written.
- **Regular Register.** A regular register guarantees for a read that is concurrent to a write, that the value will either be the last written value or a value concurrently written. This means that a read value will be a value that has been written, but two reads that are concurrent with the same write but not concurrent with each other are allowed to read the value out of order. For example, the value v_0 is stored. Then a write starts with value v_1. Concurrent to this write a read starts which returns v_1. After this read, but still concurrent to the write, a second read occurs which may return v_0.
- **Atomic Register.** In an atomic register reads and writes behave as if they occur in some definite order. For example, if a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

2.4 Existing systems

As mentioned in the introduction, many systems have been developed, that provide some form of data storage in the cloud-of-clouds model and this section outlines Belisarius by Padilha and Pedone [PP11] and DepSky by Bessani et al. [BCQ+13]. Both of these systems are byzantine fault-tolerant (BFT) and while DepSky uses a model very similar to ours, which we explain in section 3.1, Belisarius uses the same approach (secret sharing) as we do in section 3.3 to provide confidentiality and redundancy.

2.4.1 Belisarius

Belisarius is a state-machine replication system (SMR) developed by Padilha and Pedone [PP11], which aims to provide confidentiality for the parameters of operations in a $N > 3f + 1$ client-server setting. The operations that Belisarius provides are *read(key)*, *write(key, value)*, *add(key, value)* and *cmp(key, value)*, where *read* returns the value stored at the provided key, *write* stores a new value, *add* adds a provided value to the current value at key and *cmp* compares a value to the stored value. The system consists of three main components as shown in figure 2.4.1: the client-side confidentiality handler, the communication protocol, and the server-side data handler. The client-side confidentiality handler is responsible for taking requests for operations, secret sharing the parameters of operations, and forwarding the data to the communication protocol to be sent to the servers. It uses the secret sharing scheme developed by Harn and Lin [HL09], which requires $f + 2$ correct shares and participants to reconstruct the secret, and thus Belisarius requires $N > 3f + 1$ and not $N > 3f$ from the model. The communication protocol implements a byzantine fault-tolerant total order broadcast using a BFT consensus, which ensures that the order of operations is maintained between clients and servers. The server-side data handler stores and retrieves the shares, and in the case of the *add* operation, it utilizes the additive homomorphicity of Shamir's secret sharing. This means that the addition of two or more shares from different secrets results in the addition of those secrets.

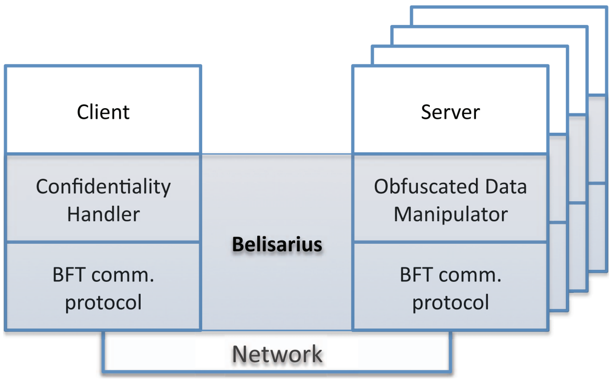

Figure 2.4.1: Overview of Belisarius (Figure by Padilha and Pedone [PP11])

2.4.2 DepSky

DepSky by Bessani et al. [BCQ+13] are two BFT storage systems that use several cloud-storage servers to provide availability (DepSky-A) and additionally confidentiality (DepSky-CA). They use the cloud-of-clouds model in a byzantine $N > 3f$ setting, where the servers are assumed to be passive storage replicas and do not execute custom code. DepSky-CA which supports confidentiality encrypts the data using symmetric encryption and stores it using erasure coding. The encryption key is then secret-shared so that each server stores a share of the key along with their erasure-coded block as shown in figure 2.4.2. In addition, DepSky-CA stores a signed metadata file per file it writes, containing the version number or timestamp as well as the digest of the file. When a write operation is performed, the system first requests the metadata file to determine the new timestamp, then creates and signs the new metadata file. It finally writes the file and then the metadata file, such that if the metadata file is read, the file itself is guaranteed to already exist on the server. For a read operation, the system first requests the metadata file from the servers to determine the correct timestamp and file digest and then requests the files. By using the digest, the system can discard any *faulty* responses and by using the timestamp, it can ignore *correct* but uninformed responses. A server is called informed if the client has received the corresponding acknowledgement from the server and a server is called uninformed if the client did not receive the corresponding acknowledgement from the server. The system continues to read until $f + 1$ *correct* and informed responses have been received, and then reconstructs the encrypted file from the received blocks. Finally, it decrypts the file using the received shares of the encryption key. Since the signature on the metadata file is assumed to be unforgeable, and the write operation waits for $N - f$ acknowledgements, the system supports regularity and forms a regular register as proven by Bessani et al. [BCQ+13].

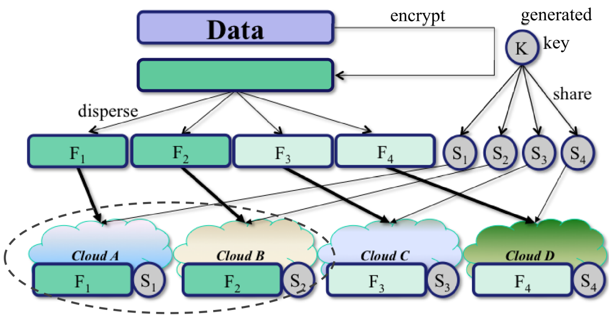

Figure 2.4.2: The combination of symmetric encryption, secret sharing and erasure codes in DepSky-CA (Figure by Bessani et al. [BCQ+13])

3 Byzantine register as a service

In this chapter, we outline the key components of our cloud-of-clouds model, including the function and abilities of clients and servers, as well as the assumptions of the model. We then present the problem we aim to solve and introduce our proposed solution. We explain our modular and event-based pseudo-code framework, for which we provide several modules and formulate their properties. Using these modules and the framework, we implement our solution and demonstrate how it addresses our problem. Lastly, we discuss the identification of *faulty* shares and the impossibility of marking the *byzantine* servers in $\mathrm{O}(1)$ without their knowledge.

3.1 Model

Our model consists of several cloud storage services (servers) that can be accessed by clients via APIs, which are supplied by the cloud storage services. Even though the cloud storage services themselves might be comprised of several servers, the APIs abstract them to one server. In our asynchronous model in which we have clients $\{C_1, C_2, ...C_k\}$ and servers $\{P_1, ..., P_N\}$, the clients are running our protocol and may *read*, while one of them is also authorized to *write*. The servers store values and do not communicate with each other. There is no known bound on processing times and message delays. We assume that we have $N > 3f$ servers, and up to f of these servers may crash or be controlled by an adversary, who can make them act arbitrarily. These servers are referred to as *faulty* or *byzantine*, while the remaining servers are called *correct* and are assumed to have their own register that clients can access remotely through read and, if permitted, write operations. The clients are connected to each server through a reliable, authenticated, asynchronous channel that provides message integrity but does not guarantee ordering. As usual in such models, it is assumed, that the adversary is not able to forge or decrypt messages from and to servers he does not control. We assume that access control is provided by the servers with an unconditionally secure system and that the servers are so-called passive storage replicas as in Bessani et al. [BCQ$^+$13], meaning that the servers are code-less and no custom code may be executed and only timestamp checks for write consistency are implemented, such that a slow connection does not overwrite newer data. We will provide the assumed code running on the servers to prove certain properties.

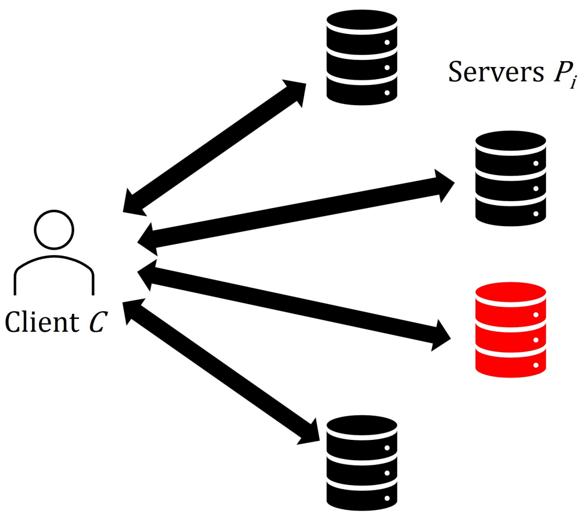

Figure 3.1.1: The model with $N = 4$ and $f = 1$ and one client

3.2 Motivation

In this thesis, we adapt Harn and Lin's [HL09] secret sharing scheme to a wait-free byzantine register. We use the same definition of a wait-free register as in Cachin et al. [CGR11], i.e. no operation should wait for responses from more than $N - f$ processes. To create this register, we first discuss the detailed working of the *read* and *write* operations and deduce the properties that follow from this. For a non-concurrent *read* operation to return a verified value, $f + 2$ *correct* shares are needed, such that there are more than f combinations of $f + 1$ shares and thus more than f equal secrets are reconstructed. In the worst case, $2f + 2$ shares from the same timestamp are needed, assuming f servers return *faulty* shares, if there is an index for *write* operations and old values from slow servers are filtered out. An operation can wait for up to $N - f$ responses to be wait-free, which, in a $N > 3f$ setting, results in $2f + 1$ responses.

Example 3.1. To show that Shamir's secret sharing as described in Harn and Lin [HL09] does not always work in a $N > 3f$ setting with byzantine servers, we set a scenario of two clients W and R, of which W has writing privileges, and p, q, r and s as servers out of which the server r crashes after the write operation. We specifically do not assume any ordering on the links between the client and the servers as per our model. The client W starts a *write* with shares of value v_1 and gets acknowledgments of servers p, q and r, after which r crashes. In this case, it does not matter for future *read* operations, whether s crashes or is byzantine and provides *faulty* shares. When client R now *reads*, it will receive the shares of v_1 from p and q and the initial share of $\perp$ from s. Since it now only has $f + 1$ shares of the same value, it only knows, that the three shares are not from the same secret. Specifically, that the shares from p, q, and r do not form a polynomial of degree f as in example 2.1. Thus it has to wait for s to give its share, which it never will. See Figure 3.2.1.

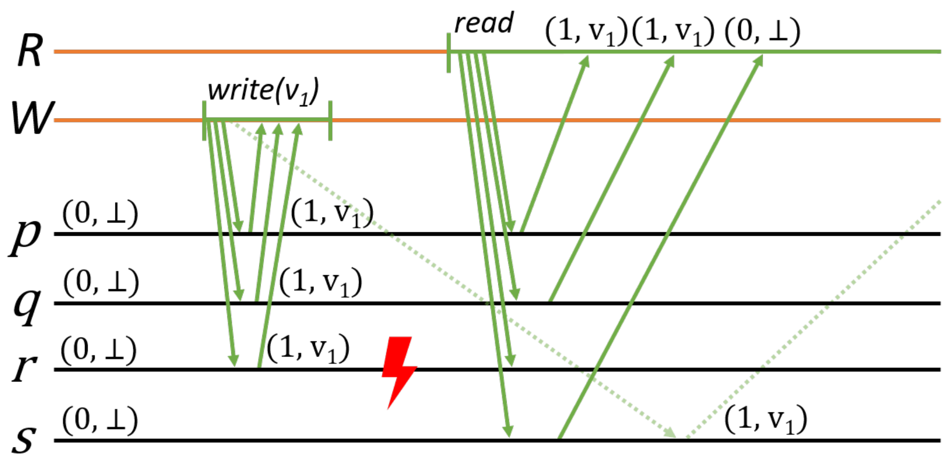

Figure 3.2.1: Problem illustration

A common solution is the Read-your-Write mechanism (c.f. Wada et al. [WFZ$^+$11]), where the writer writes a value, immediately reads it, and repeats this, until the read value matches the written value, which does not work in our case. In example 3.1 the Read-your-Write would succeed, because p,q, and r are able to send back their share and W can reconstruct v_1. So the Read-your-Write will succeed, W will indicate the *WriteReturn*, but R would not be able to reconstruct v_1 after the crash of r. Adding one more correct server would solve this problem since in a $N > 3f + 1$ system a wait-free process can wait for $2f + 2$ responses and thus a reader has $2f + 2$ shares out of which $f + 2$ are *correct* and hence a reconstructed secret can be verified. Thus a secret sharing register with Harn and Lin's [HL09] cheater detection and identification does not directly work in a $N > 3f$ protocol and needs some modification.

Wise cheating attack. Ghodosi [Gho11] proposed a *wise cheating attack* on Harn and Lin's [HL09] secret sharing scheme, where an adversary creates a second polynomial $g(x)$ with the same degree and where $g(x_i) = 0$ for all i of *correct* servers P_i. The byzantine servers then return the sum of their share from $f(x_i)$ and $g(x_i)$. Due to the homomorphicity of Shamir's secret sharing, not only the *byzantine* shares but also the *correct* shares are shares of a polynomial $h(x) = f(x) + g(x)$. When trying to reconstruct the polynomial from a set of shares that include such a *faulty* share, the resulting polynomial will be $h(x)$ and the reconstructed secret $h(0) = s' \neq s = f(0)$. In other words, a reader will not be able to detect *faulty* shares and *faulty* secrets. We address this problem by randomizing x_i and treating $(x_i, f(x_i))$ as a share, such that an adversary will have no knowledge about the x_i of *correct* servers. The probability of a successful wise cheating attack hence is low because the *byzantine* servers need to correctly guess the randomized x_i-value, which is unlikely given the large size of the field $\mathbb{F}_p$ and the x_i being randomly generated for each write. Additionally this is a blind attack for the servers, i.e. the servers are not informed, whether the attack was successful or not, and the readers are able to identify *faulty* shares, as we discuss in section 3.6. As a result, the wise cheating attack is infeasible with randomized and hidden x_i and we do not consider it in the proofs outlined in section 4.1. Even if a successful attack were to occur, it would only result in a *faulty* reconstructed secret being passed as verified, without compromising the unconditional security of the secret sharing scheme, as the *byzantine* servers do not gain any information.

To summarize, we want an unconditionally secure algorithm, that returns verified secrets, is wait-free, and works in a $N > 3f$ setting. Since $N > 3f$, the $N - f$ responses from the wait-freeness are at least $2f + 1$ and verified secrets require $2f + 2$ shares. Hence, we need to develop a new secret sharing scheme, that has $2f + 2$ shares within $2f + 1$ responses.

3.3 Distributed additional share

We want our register to work in a $N > 3f$ setting. To solve the problem of needing $2f + 2$ shares in $2f + 1$ responses, we propose to create an additional share s_{N+1} of the secret, secret-share it and give each server P_i their

share s_i as well as their share of the additional share s_{N+1_i}. In a reconstruction with $2f + 1$ responses, a client first reconstructs all possible additional shares, of which at least one is correct, since at least $f + 1$ shares of the additional share are *correct*. Hence there are at least $f + 1$ correct s_i and at least one correct s_{N+1_j}. Following from that there are at least $f + 2$ correct s_i and thus the secret can be reconstructed.
In our $(t + 1, N)$ secret sharing scheme, a client C first picks a polynomial $f(x)$ of degree t, such that $f(x) = a_0 + a_1x + ... + a_tx^t$, in which the secret $s = a_0$ and the coefficients $a_1, ..., a_t$ are picked randomly from a finite field $\mathbb{F}_p$. C then computes $s_1 = f(x_1), ..., s_{N+1} = f(x_{N+1})$, with x_i picked uniquely random from $\mathbb{F}_p$. Next, C picks a different polynomial $g(u)$ of the same degree t, such that $g(u) = b_0 + b_1u + ... + b_tu^t$, where $b_0 = s_{N+1}$ and the coefficients $b_1, ..., b_t$ are picked uniquely random from a finite field $\mathbb{G}_q$. Finally, C computes $r_1 = g(u_1), ..., r_N = g(u_t)$ with u_i picked uniquely random from $\mathbb{G}_q$ and sends $(x_i, s_i, x_{N+1}, u_i, r_i)$ to each server P_i secretly. For simplicity, we will assume that the value x_{N+1} is publicly known and will therefore omit its transmission in our description of the algorithm because each server receives the same x_{N+1} value. To reconstruct the secret s, a client B requests the shares (x_i, s_i, u_i, r_i) from each server P_i. B first reconstructs all possibilities for s_{N+1} with $t + 1$ r_i shares. Next for each combination of $t + 1$ s_i shares and one s_{N+1_j} share, it reconstructs a $f'(x)$ polynomial until it finds one with a degree of t. By Harn and Lin [HL09], the $f'(x)$, that has degree t is equal to $f(x)$ and hence B will calculate the secret $s = f'(0) = f(0)$.

3.4 Modules

To create modular algorithms, we work with modules as in Cachin et al. [CGR11]. The modules abstract algorithms as an event-based system, such that a module or algorithm can invoke a service of another module by *requesting* the corresponding event. A module can also deliver information to another module by *indicating* an event, which might trigger an execution of some code at a listener. Thus modules can collaborate through requests and indications. Each module is given a name, events, and properties. The properties guarantee the behavior of the module between requests and indications. The events may be in the form of $\langle$ *xx*, *event* | *arguments*,... $\rangle$, where *xx* uniquely defines the instance of a module, the *event* corresponds to a service, that the module exposes and optionally, the *event* might have *arguments* that are parameters of the service. We will explain how to implement modules in section 3.5. A full treatment of the modules is beyond the scope of this thesis and further details can be consulted in Cachin et al. [CGR11].

3.4.1 Secret sharing module

We first declare the secret sharing module that has three properties: Termination, Correctness, and Validity. Specifically, it is able to generate shares for a secret m, reconstruct a list of possible secrets from a set of shares, and verify a secret from a set of shares and a list of possible secrets. This module works as trivially expected, such as a secret is verified if at least $f + 2$ *correct* shares have been used in the reconstruction. If less than $f + 2$ shares have been used in the reconstruction, the verify operation will return the special value $\perp$. Also, all operations will eventually complete. Hence, these properties ensure the reliability and security of the secret sharing module.

Module 3.1 Interface and properties of a secret sharing module (SS)

Module

Name: SecretSharing, **instance** ss.

Events:

Request: $\langle$ *ss*, *GenerateShares* | t, m $\rangle$: Invokes a generation operation to create t shares from the message m
Request: $\langle$ *ss*, *Reconstruct* | t, $\{s_0, ..., s_i\}$ $\rangle$: Invokes an reconstruction operation with shares $\{s_0, ...s_i\}$ and threshold t.
Request: $\langle$ *ss*, *Verify* | t, $\{s_0, ..., s_i\}$, $\{m_0, ..., m_i\}$ $\rangle$: Invokes a verification operation.
Indication: $\langle$ *ss*, *DeliverShares* | $\{s_0, ..., s_i\}$ $\rangle$: Completes a generation operation and returns the generated shares $\{s_0, ..., s_i\}$.
Indication: $\langle$ *ss*, *Reconstructed* | $\{m_0, ..., m_i\}$ $\rangle$: Completes a reconstruction operation and returns all possible secrets $\{m_0, ..., m_i\}$.
Indication: $\langle$ *ss*, *Verified* | m $\rangle$: Completes a verification operation and returns the verified secret m.

Properties:

SS1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

SS2: *Correctness:* If a correct process invokes a GenerateShares operation of message m and threshold t and the operation completes with shares s. Then the Reconstruct operation with any set s' containing at least $t + 2$ shares of s will complete and returns secrets $m' = \{m_1, ..., m_i\}$. Then a Verify operation with t, s' and m' will complete with $m_j = m$. If the Reconstruct operation is requested with a set s^* containing $t + 1$ shares of s, then it will complete with $m^* = \{m_1^*, ..., m_j^*\}$ and $m \in m^*$.

SS3: *Validity:* If a correct process invokes a Reconstruct operation and the operation returns secrets m_1, ..., m_i and a Verify operation with the same parameters and those secrets completes, then it returns exactly one secret m_j, that has been used in a GenerateShares operation. If a correct process invokes a Verify operation and the operation returns $\perp$, then no secret could be verified.

3.4.2 Communication between clients and servers

We want our system to work in a cloud-of-clouds model, where a client can request a read or write operation on a server. To fully define a communication channel between a client and a server, we need a few modules, which enable us to do so. We expect the cloud service providers to expose APIs (application programming interfaces) that clients can use to request a server to do an operation. In a real system, where an application calls an API, the underlying library of the programming language calls system libraries, which in turn talk to the hardware of the machine. Next, this hardware connects over the internet to the corresponding device, where the system libraries forward the message to the corresponding server program, which handles the request. Usually, the server program then sends back a response in the same way. To model this already simplified process, we create an abstract module, called API, that handles the calls to the system library, the connection to the server, and replies with the server message to simulate the behavior of the above-mentioned library of the programming language. We will also need to model the hardware links to justify the choice of properties of the API.
As a communication base, we need a Point-to-Point communication, for which we will be using the AuthPerfect-PointToPointLinks from Cachin et al. [CGR11] which provides us a reliable and authenticated link in module 3.2. Next, we develop an unconditionally secure message transfer with module 3.3. Then we split the processes into Clients and Servers and model the client and server behavior in module 3.4, which allows us to later model the round-trip feature of the API as in figure 3.4.1, in which a message from a client gets delivered to a server, which in turn sends back a message to the client. Finally, we can model the API behavior, which combines the properties of the ClientServerLink module, contains the round-trip feature, and provides the client an interface as in figure 3.4.1.

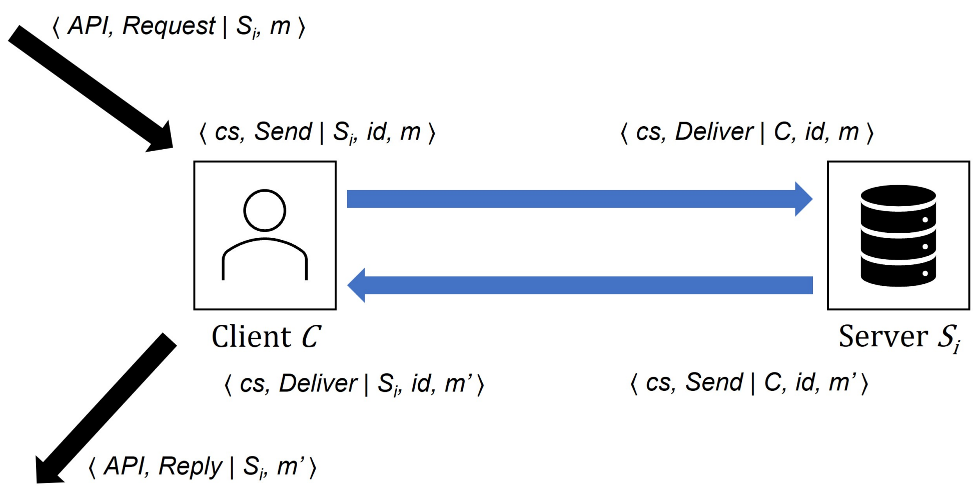
⟨ API, Request | S_i, m ⟩
⟨ cs, Send | S_i, id, m ⟩
⟨ cs, Deliver | C, id, m ⟩
Client C
Server S_i
⟨ cs, Deliver | S_i, id, m' ⟩
⟨ cs, Send | C, id, m' ⟩
⟨ API, Reply | S_i, m' ⟩

Figure 3.4.1: The schema of the API module

Module 3.2 Interface and properties of authenticated perfect point-to-point links (c.f. Cachin et al. [CGR11]) (AL)

Module

Name: AuthPerfectPointToPointLinks, **instance** *al*.

Events:

Request: ⟨ *al*, *Send* | q, m ⟩: Requests to send a message m to process q.
Indication: ⟨ *al*, *Deliver* | p, m ⟩: Delivers message m sent by process p

Properties:

AL1: *Reliable Delivery:* If a correct process sends a message m to a correct process q, then q eventually delivers m.

AL2: *No Duplication:* No message is delivered by a correct process more than once.

AL3: *Authenticity:* If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

Since our register should provide unconditional security, we need the communication over the links to provide a form of secrecy that resists any adversary, even with unlimited computational power. Hence, we add the *unconditional encryption* property, which prohibits an adversary with unlimited computational power to obtain any information about the sent message in transit.

Module 3.3 Interface and properties of encrypted perfect point-to-point links (EL)

Module

Name: EncryptedPerfectPointToPointLinks, **instance** *el*.

Events:

Request: $\langle$ *el*, *Send* | q, m $\rangle$: Requests to send a message m to process q.
Indication: $\langle$ *el*, *Deliver* | p, m $\rangle$: Delivers message m sent by process p

Properties:

EL1-EL3: Same as AL1-AL3.

EL4: *Unconditional Encryption:* If some correct process p send a message m to a correct process q, then no adversary even with unlimited computational power can derive any information about m.

Even though no further properties can be developed from the encrypted point-to-point links from module 3.3, we split the message into an identification id, a command *command*, and a message m. With this splitting of the message, we can now contextually bind messages together with id and we use the COMMAND to model HTTP methods or endpoints at the server. For example, a client might want to send a GET request to google.com/search?q=Link with the search term ”Link”, which can be modeled as $\langle$ *cs*, *Send* | *google*, id, GETSEARCH, ”Link” $\rangle$. The identification is a general mechanism, that allows several use cases, such as session cookies for web requests or timestamps for ordering purposes.

Module 3.4 Interface and properties of Client-Server (CS)

Module

Name: ClientServerLink, **instance** *cs*, with
Clients $C = \{C_1, C_2, ...\}$, Servers $S = \{S_1, S_2, ..., S_N\}$ and Processes $P = C \cup S$

Events:

Request: $\langle$ *cs*, *Send* | q, id, *command*, m $\rangle$: Requests to send a message m with identification id and *command* to process q.
Indication: $\langle$ *cs*, *Deliver* | p, *id′*, *command′*, m' $\rangle$: Internally delivers a message m with identification id and command *command′* sent by process p.

Properties:

CS1-CS4 Same as EL1-EL4

Finally, we can build our API module, which provides an interface for the client to call an API on the server and receive a reply with the response from the server. This module is no longer just a link, but will internally use a link for the communication between clients and servers. The properties **API1-API4** are the same properties as in the client-server module 3.4, but slightly adapted, since the API uses *Request* and *Reply*, instead of *Send* and *Deliver* to emphasize that a server cannot *Request* or *Reply*. This module should be implemented by both, clients and servers.

Module 3.5 Interface and properties of Application Programming Interface (API)

Module

Name: Application Programming Interface, **instance** *API*, with
Clients $C = \{C_1, C_2, ...\}$, Servers $S = \{S_1, S_2, ..., S_N\}$ and Processes $P = C \cup S$

Events:

Request: $\langle$ *API*, *Request* | q, id, *command*, m $\rangle$: Requests destination process q to do *command* with id message m. //only Clients C

Indication: $\langle$ *API*, *Reply* | p, id', *command*$'$ m' $\rangle$: Source process p replies *command*$'$, id' and message m'. //only Clients C

Properties:

API1: *Reliable Requests:* If some process p requests with destination process q and q is correct, then p eventually replies with source process q.

API2: *No Duplication:* No request with correct destination process q is replied more than once.

API3: *Authenticity:* If some process p replies *command* with source process q and q is correct, then q sent *command* to p.

API4: *Unconditional Encryption:* If some process p replies with a message m' and source process q to a request with message m and destination process q and q is correct, then no adversary even with unlimited computational power can derive any information about m or m'.

API5: *Client-Server Separation:* If some process p requests with destination process q or p replies with source process q, then p is a client and q is a server.

With module 3.5, we have modeled the usual API behavior as in figure 3.4.1, where an application can invoke a request to the API on the client side and eventually gets indicated a reply. Module 3.5 does not enforce the usage of the client-server module 3.4 as a link between the client and server, but for illustration purposes, we used the client-server as the link.

3.4.3 Registers

We base our register on the modules from Cachin et al. [CGR11] and use the module 4.1 (1, N) Byzantine Regular Register in the book from Cachin et al. [CGR11]. But we adapt their validity property to allow multiple concurrent writes to a read as in Herlihy and Shavit. [HS08]. A regular register typically has two properties: *termination*, which ensures that every operation eventually completes, and *validity*, which ensures that a read will only return the last written value or one of the values concurrently written. Our byzantine version of a regular register maintains the validity property, but has a weaker version of the termination property. Specifically, the byzantine regular register has the finite write termination property, meaning that termination is only guaranteed for a finite number of writes. We explain the reasoning in section 3.5.2.

Module 3.6 Interface and properties of a byzantine safe register (BSR)

Module

Name: (1, N)-ByzantineSafeRegister, **instance** bsr, with writer w.

Events:

Request: ⟨ *bsr*, *Read* ⟩: Invokes a read operation on the register.
Request: ⟨ *bsr*, *Write* | v ⟩: Invokes a write operation with value v on the register.
Executed only by process w.
Indication: ⟨ *bsr*, *ReadReturn* | v ⟩: Completes a read operation on the register with return value v.
Indication: ⟨ *bsr*, *WriteReturn* ⟩: Completes a write operation on the register.
Occurs only at process w.

Properties:

BSR1: *Termination:* Every operation eventually completes.

BSR2: *Validity:* A read that is not concurrent with a write returns the last value written. A read that is concurrent with one or more writes returns a value in the domain.

Module 3.7 Interface and properties of a byzantine regular register (BRR)

Module

Name: (1, N)-ByzantineRegularRegister, **instance** brr, with writer w.

Events:

Request: ⟨ *brr*, *Read* ⟩: Invokes a read operation on the register.
Request: ⟨ *brr*, *Write* | v ⟩: Invokes a write operation with value v on the register.
Executed only by process w.
Indication: ⟨ *brr*, *ReadReturn* | v ⟩: Completes a read operation on the register with return value v.
Indication: ⟨ *brr*, *WriteReturn* ⟩: Completes a write operation on the register.
Occurs only at process w.

Properties:

BRR1: *Finite Write Termination:* Every write operation eventually completes and either every read operation eventually completes or the writer invokes infinitely many write operations

BRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with one or more writes returns the last value written or one of the values concurrently written.

The registers in modules 3.6 and 3.7 trigger requests to the API in module 3.5, which uses the client-server link in module 3.4 to facilitate communication between clients and servers as in figure 3.4.2. In this implementation of a register, the clients are responsible for the execution flow and providing the functionality, while the servers simply offer remote access to their local storage to the clients. Figure 3.4.3 illustrates which modules are implemented by the servers and which by the clients, and it is noted that both the client and server code for the API module are required to fulfill its properties and cannot be provided by one alone.

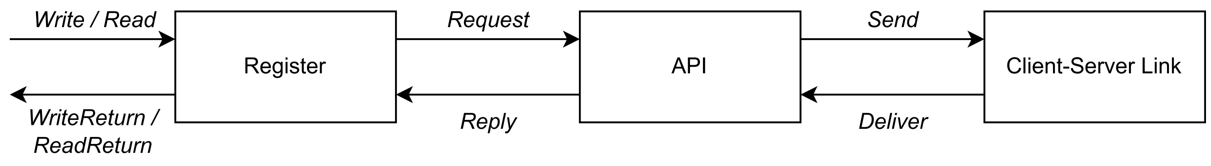

Figure 3.4.2: Event flow across modules

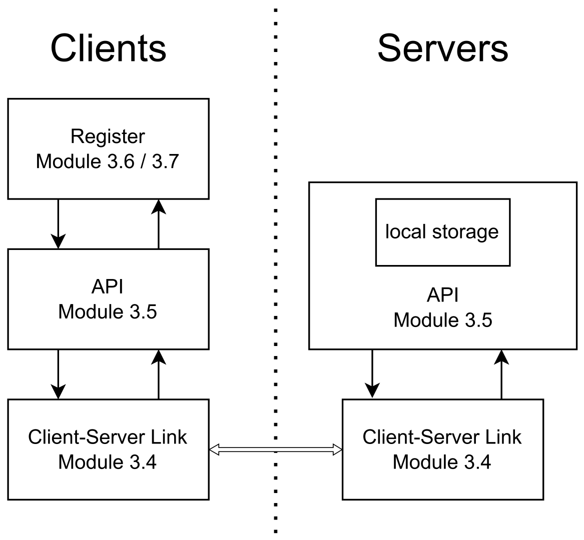
Clients
Servers
Register Module 3.6 / 3.7
API Module 3.5
Client-Server Link Module 3.4
local storage
API Module 3.5
Client-Server Link Module 3.4

Figure 3.4.3: Roles of clients and servers

3.5 Algorithm

In this section of the thesis, we present the pseudo code for different algorithms in an event-based form that allows the algorithms to implement the corresponding modules from section 3.4. There are two types of events: Request and Indication, which are the same as in the modules in section 3.4. As explained that section, any module can invoke requests to an other module, which might invoke indications back to the invoker module. The structure of an event is ⟨ *xx*, *event* | *arguments*, ... ⟩, where *xx* is the instance of a module, *event* is the name of the event and optionally *arguments* may be provided as parameters of the event. The events request and indication can be invoked with the statement **trigger** *event*. If any indication has been invoked at an instance of a module, a corresponding flag is set behind the scenes and with the statement **event** *indication*, the status of the flag corresponding to *indication* can be checked. If this flag is set, **event** *indication* resets the flag and returns TRUE, otherwise, it returns FALSE. The statement **upon** *condition* **do** forms an event listener, where *condition* is repeatedly executed and if *condition* resolves to TRUE, the code inside the event listener is executed. The *condition* is automatically executed if no other code is executed. The statement **wait for** *condition* repeatedly executes *condition* until it resolves to TRUE as with **upon**. Conversely, if *condition* resolves to FALSE, *condition* will immediately be executed again and thus **wait for** hinders **upon** from executing its own *condition* as well as any other code from being executed, except the setting of the flags of indications. Hence, with **trigger** ⟨ *xx*, *requestX* | *args* ⟩ immediately followed by **wait for event** ⟨ *xx*, *indicationX* | *args'* ⟩, events can be used as if they were functions, such that properties of these functions can be formulated in modules and local variables do not need to be stored globally. However, if a request does not guarantee termination, the invoker as well can not guarantee termination, since ⟨ *xx*, *indicationX* | *args'* ⟩ might never get invoked. The special event ⟨ *xx*, *Init* ⟩ is assumed to be invoked without arguments at the instantiation of the algorithm. To simplify pattern matching in event listeners, we allow arguments within an **upon event** statement to be in SMALLCAPS to indicate that the parameter has to match the value of the variable in small capitals while arguments in *cursive* are passed as a variable, i.e. **upon event** ⟨ *xx*, *name* | *variable*, VAL ⟩ **do** is equivalent to **upon event** ⟨ *xx*, *name* | *variable*, *value* ⟩, **such that** *value*=*val* **do**.

We implement two versions of our additional share approach, the first algorithm (algorithm 3.2) highlights the scheme described in section 3.3 alongside implementing a safe register in a $N > 4f$ setting, by waiting for $N - f$ responses for the *read* and *write*. The second algorithm (algorithm 3.4) combines the first algorithm with the double-write approach to make a regular register in a $N > 3f$ model. The double write is used to guarantee the validity property of the byzantine regular register in module 3.7. We will be discussing the double-write algorithm in more detail in section 3.5.2. Before we can present the algorithms that use our approach with the additional share, we present implementations of the API used that describe which commands are available and how they work.

3.5.1 Safe Register

The API algorithm 3.1 implements two commands, READ and WRITE, where the READ command returns the stored value as a READRETURN command and the WRITE command with message m stores m in the local storage. We implemented a simple timestamp check to ensure that only newer data gets stored. Any other command will be returned with UNKNOWNCOMMAND to ensure that all commands are replied to. The algorithm we explained in section 3.3 works in a $N > 4f$ model and implements the byzantine safe register as in module 3.6 and can be split into a read- and a write part. $N > 4f$ is required, since the overlap of two $N - f$ responses from a write and a read may only contain 1 *correct* and informed response if $N > 3f$. In $N > 3f$, the write waits for $N - f = 2f + 1$ acknowledgments and the reader waits for $N - f = 2f + 1$ responses. Following from that, the $2f + 1$ responses for the read may contain f *byzantine* responses and f responses from *correct*, but uninformed servers and thus only one response from a *correct* and informed server and no reconstruction of the secret is possible even if the read is not concurrent to a write. Hence $N > 4f$ is required In the write part, the writer first generates $N + 1$ shares s_i for a message m. Next, it splits the additional $(N + 1)^{st}$ share into N shares s_{N+1_i} and writes the tuple (s_i, s_{N+1_i}) to each server P_i. When a client invokes a read operation, it first reads the stored values from $N - f$ servers. Then, to deter *byzantine* servers from providing old shares, the responses consisting of the timestamp, share, and additional share, are filtered for the maximal timestamp with lengths more than f and split into shares and additional shares. Next, the possible secrets are reconstructed and verified. When a secret is verified, the process completes the read operation and returns the secret. If no secret could be verified it returns a default value, indicating that there exists a concurrent write.

Algorithm 3.1 Safe Register API

```
Implements:
    ApplicationProgrammingInterface, instance rapi, with writer $w$.

Uses:
    ClientServerLink, instance cs

//Server only
upon event ⟨ rapi, Init ⟩ do
    storage := $\perp$
    ts := 0

//Client only
upon event ⟨ rapi, Request | $q$, id, command, $m$ ⟩ do
    trigger ⟨ cs, Send | $q$, id, command, $m$ ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, WRITE, $m$ ⟩ do
    wts := id //identification is used as the write timestamp
    if wts $>$ ts then
        ts := wts
        storage := $m$
        trigger ⟨ cs, Send | $p$, wts, ACK ⟩
    else
        trigger ⟨ cs, Send | $p$, wts, NACK ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, READ, $m$ ⟩ do
    trigger ⟨ cs, Send | $p$, id, READRETURN, (ts, storage) ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, command, $m$ ⟩ such that command $\notin$ {WRITE, READ} do
    trigger ⟨ cs, Send | $p$, id, UNKNOWNCOMMAND ⟩

//Client only
upon event ⟨ cs, Deliver | $q$, id, command, $m$ ⟩ do
    trigger ⟨ rapi, Reply | $q$, id, command, $m$ ⟩
```

Algorithm 3.2 Secret-Shared Safe Register (Part 1: Write)

```
Implements:
    (1, $N$)-ByzantineSafeRegister, instance sssr, writer $w$.

Uses:
    SecretSharing, instance ss.
    ApplicationProgrammingInterface, instance rapi.

upon event ⟨ sssr, Init ⟩ do
    $\Omega := \{S_1, ..., S_N\}$
    $N := |\Omega|$
    $f := \frac{|\Omega|-1}{3}$
    $additionalShareFlag_w$ := FALSE
    $additionalShareFlag_r$ := TRUE
    newShares := ∅
    newAdditionalShares := ∅
    shares := ∅
    additionalShares := ∅
    responses := ∅
    acklist := $[\ \perp\ ]^N$
    readlist := $[\ \perp\ ]^N$
    wts := 0
    rts := 0

//Writer $w$ only
upon event ⟨ sssr, Write | $m$ ⟩ do
    $additionalShareFlag_w$ := FALSE
    trigger ⟨ ss, GenerateShares | $N+1$, $m$ ⟩

//Writer $w$ only
upon event ⟨ ss, DeliverShares | generatedShares ⟩ do
    if not $additionalShareFlag_w$ then
        $additionalShareFlag_w$ := TRUE
        newShares := generatedShares
        trigger ⟨ ss, GenerateShares | $N$, newShares$[N+1]_y$ ⟩
    else
        $additionalShareFlag_w$ := FALSE
        newAdditionalShares := generatedShares
        wts := wts+1
        forall $q \in \Omega$ do
            trigger ⟨ rapi, Request | $q$, wts, WRITE, [newShares[$q$], newAdditionalShares[$q$]] ⟩

//Writer $w$ only
upon event ⟨ rapi, Reply | $q$, WTS, ACK ⟩ do
    acklist[$q$] := WRITEACK
    if $|acklist| = N - f$ then
        acklist := $[\ \perp\ ]^N$
        trigger⟨ sssr, WriteReturn ⟩
```

Algorithm 3.2 Secret-Shared Safe Register (Part 2: Read)

```
upon event⟨ sssr, Read ⟩ do
    rts := rts +1
    shares := ∅
    additionalShares := ∅
    additionalShareFlag_r := TRUE
    forall $q \in \Omega$ do
        trigger⟨ rapi, Request | q, rts, READ ⟩

upon event⟨ rapi, Reply | q, RTS, READRETURN, (ts, [share, additionalShare]) ⟩ do
    responses[q] := (ts, share, additionalShare)
    if $|responses| = N - f$ then
        $R$ := split responses into sets with common timestamps
        $Q$ := set with maximal timestamp of sets in $R$ with length $> f$
        shares := $Q_{\text{shares}}$
        additionalShares := $Q_{\text{additionalShares}}$
        additionalShareFlag_r := TRUE
        trigger ⟨ ss, Reconstruct | f, additionalShares ⟩

upon event ⟨ ss, Reconstructed | secrets ⟩ do
    if additionalShareFlag_r then
        additionalShareFlag_r := FALSE
        shares[$N + 1$] := secrets // secrets is the list of possible additional shares
        trigger ⟨ ss, Reconstruct | f, shares ⟩
    else
        additionalShareFlag_r := TRUE
        trigger ⟨ ss, Verify | f, shares, secrets) ⟩

upon event ⟨ ss, Verified | secret ⟩ do
    if secret $=\perp$ then
        trigger⟨ sssr, ReadReturn | default ⟩ // default is in the domain of possible secrets
    else
        trigger⟨ sssr, ReadReturn | secret ⟩
```

3.5.2 Regular Register

Since algorithm 3.2 only guarantees to return some value within the domain for a read operation that is concurrent with a write operation, we need to adapt it to support regularity. The following algorithm 3.4 implements a regular register, as an atomic register can be constructed from a regular register (c.f. [CGR11]). Abraham et al. [ACKM06] proposed the double-write byzantine quorum to implement a regular register in a $N > 3f$ setting without the need for digital signatures, which is suitable for our model. However, as we are using secret sharing to enable confidentiality and each server receives different data, we must adapt the double-write byzantine quorum to work with our algorithm.

To perform a write using the double-write algorithm, the data is first written to a storage called PREWRITE on the servers. The algorithm then waits for acknowledgments from $N - f$ servers before writing the same data to a storage called WRITE. This ensures that the data in WRITE is only overwritten when at least $N - f$ servers have the new data in their PREWRITE storage. Our algorithm as well as the double-write byzantine quorum work exactly the same for writing. However, we need to adapt the read, because we need to intersect in a set greater than f *correct* and informed responses, while Abraham et al. only require one.

Since at least $f + 2$ shares are required to verifiably return a secret and we achieve it with $f + 1$ responses, returning a value, that has never been written is not possible and we only need to make sure that values that have been written before the last non-concurrently written value are not returned. For example if three values (a_0, a_1, a_2) have been written and concurrently to the write of a_2 a read occurs, the read should not return a_0 or older values. Since some slow but *correct* servers might still have shares of a_0 or older, the *byzantine* servers might provide their shares of a_0 and thus the reader has shares from more than f servers and can reconstruct and return a_0, which opposes regularity. Thus algorithm 3.4 reads until two sets Q and R can be created, such that $|Q| > f$ and $|R| > 2f$. Q contains at least $f + 1$ responses to verifiably reconstruct the secret and has a common timestamp. R contains Q and shares with a lower timestamp than the ones in Q. R contains at least one *correct* share of a write, that is either concurrent to the read or the last written value because the write waited for $N - f$ responses and thus up to f might have old values and f might be faulty and hence the size of R is chosen to be $|R| > 2f$. Since R must only have timestamps that are equal or lower than the ones in Q and the set Q is used for a verified reconstruction and only verified secrets are returned, the algorithm will not return values that oppose regularity. Note here, that R is a subset of the *readlist* and thus contains PREWRITE and WRITE data, while Q contains pairs of timestamps, shares, and additional shares. Hence Q contains data, that is directly used in the reconstruction and R ensures regularity. We will prove regularity in section 4.1.2. Since we now have two types of storages PREWRITE and WRITE, we need to slightly adapt the API in algorithm 3.3 to allow the command PREWRITE. The READ command returns both storages together.

Algorithm 3.3 Regular Register API

```
Implements:
    ApplicationProgrammingInterface, instance dwapi, with writer $w$.

Uses:
    ClientServerLink, instance cs

//Server only
upon event ⟨ dwapi, Init ⟩ do
    PreWriteStorage := ⊥
    WriteStorage := ⊥
    pts := 0
    ts := 0

//Client only
upon event ⟨ dwapi, Request | $q$, id, Command, $m$ ⟩ do
    trigger ⟨ cs, Send | $q$, id, Command, $m$ ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, PREWRITE, $m$ ⟩ do
    wts := id //identification is used as the write timestamp
    if wts = pts then
        pts := wts
        PreWriteStorage := $m$
        trigger ⟨ cs, Send | $p$, wts, PREACK ⟩
    else
        trigger ⟨ cs, Send | $p$, wts, PRENACK ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, WRITE, $m$ ⟩ do
    wts := id //identification is used as the write timestamp
    if wts = ts then
        ts := wts
        WriteStorage := $m$
        trigger ⟨ cs, Send | $p$, wts, ACK ⟩
    else
        trigger ⟨ cs, Send | $p$, wts, NACK ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, READ, $m$ ⟩ do
    trigger ⟨ cs, Send | $p$, id, READRETURN, [{pts, PreWriteStorage}, {ts, WriteStorage}] ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, Command, $m$ ⟩ such that Command∉{PREWRITE, WRITE. READ} do
    trigger ⟨ cs, Send | $p$, id, UNKNOWNCOMMAND ⟩

//Client only
upon event ⟨ cs, Deliver | $q$, id, RESPONSECOMMAND, $m$ ⟩ do
    trigger ⟨ dwapi, Reply | $q$, id, RESPONSECOMMAND, $m$ ⟩
```

Algorithm 3.4 Secret-Shared Regular Register (Part 1: Write)

```
Implements:
    (1,$N$)-ByzantineRegularRegister, instance ssrr, writer $w$.

Uses:
    SecretSharing, instance ss.
    ApplicationProgrammingInterface, instance dwapi.

upon event ⟨ ssrr, Init ⟩ do
    $\Omega := \{S_1, ..., S_N\}$
    $N := |\Omega|$
    $f := \frac{|\Omega|-1}{3}$
    (pts, pval, pAddVal) := (0, ∅, ∅)
    (ts, val, AddVal) := (0, ∅, ∅)
    (wts, wval, wAddVal) := (0, ∅, ∅)
    preacklist := $[\perp]^N$
    acklist := $[\perp]^N$
    rid := 0
    readlist := ∅
    $additionalShareFlag_w$ := FALSE

//Writer $w$ only
upon event ⟨ ssrr, Write | $m$ ⟩ do
    $additionalShareFlag_w$ := FALSE
    trigger ⟨ ss, GenerateShares | $N+1$, $m$ ⟩

//Writer $w$ only
upon event ⟨ ss, DeliverShares | generatedShares ⟩ do
    if not $additionalShareFlag_w$ then
        $additionalShareFlag_w$ := TRUE
        wval := generatedShares
        trigger ⟨ ss, GenerateShares | $N$, $wval[N+1]_y$ ⟩
    else
        $additionalShareFlag_w$ := FALSE
        wAddVal := generatedShares
        wts := wts+1
        preacklist := $[\perp]^N$
        acklist := $[\perp]^N$
        forall $q \in \Omega$ do
            trigger ⟨ dwapi, Request | $q$, wts, PREWRITE, [wval[$q$], wAddVal[$q$]] ⟩

//Writer $w$ only
upon event ⟨ dwapi, Reply | $q$, WTS, PREACK ⟩ do
    preacklist[$q$] := PREACK
    if $|preacklist| = N - f$ then
        preacklist := $[\perp]^N$
        forall $q \in \Omega$ do
            trigger ⟨ dwapi, Request | $q$, wts, WRITE, [wval[$q$], wAddVal[$q$]] ⟩

//Writer $w$ only
upon event ⟨ dwapi, Reply | $q$, WTS, ACK ⟩ do
    acklist[$q$] := ACK
    if $|acklist| = N - f$ then
        acklist := $[\perp]^N$
        trigger ⟨ ssrr, WriteReturn ⟩
```

Algorithm 3.4 Secret-Shared Regular Register (Part 2: Read)

```
upon event⟨ ssrr, Read ⟩ do
    rid := rid + 1
    readlist := [ ⊥ ]$^N$
    forall $q$ in $\Omega$ do
        trigger⟨ dwapi, Request | $q$, rid, READ ⟩

upon event⟨ dwapi, Reply | $q$, RID, READRETURN, [{pts', pval',pAddVal'},{ts', val', AddVal'} ] ⟩ do
    if pts' = ts' + 1 $\vee$ (pts', pval', pAddVal') = (ts', val', AddVal') then
        readlist[$q$] := (pts', pval', pAddVal', ts', val', AddVal')
    if exists $Q$ := [(ts', val', AddVal')]$^i$, with $|Q| = i > f \wedge$ exists $R \subseteq$ readlist with $|R| > 2f$ such that
            authenticAndMax($Q$, $R$, readlist) := TRUE $\wedge$ $m$ =Verify(Reconstruct($Q$)) $\neq\perp$ then
        readlist := [ ⊥ ]$^N$
        trigger⟨ ssrr, ReadReturn | $m$ ⟩
    else
        trigger⟨ dwapi, Request | $q$, rid READ ⟩

function authenticAndMax($Q$, $R$, readlist)
    for $k \in Q$ do
        if $k$ not in PREWRITE or WRITE of $R$ then
            return FALSE
    if there are two or more different ts in $Q$ then
        return FALSE
    ts := the common timestamp in $Q$
    for (pts', pval', paddVal', ts', val', addVal') $\in R$ do
        if pts' $>$ ts $\vee$ ts' $>$ ts then
            return FALSE
    return TRUE

function Reconstruct($Q$)
    (rts, rshares, raddval) := $Q$
    trigger ⟨ ss, Reconstruct | $f$, raddval ⟩
    $\{m_0, ..., m_i\}$ := wait for event ⟨ ss, Reconstructed | $\{m_0, ..., m_i\}$ ⟩
    rshares[$N + 1$] := $\{m_0, ..., m_i\}$
    trigger ⟨ ss, Reconstruct | $f$, rshares ⟩
    $\{m_0, ..., m_i\}$ := wait for event ⟨ ss, Reconstructed | $\{m_0, ..., m_i\}$ ⟩
    return (rshares, $\{m_0, ..., m_i\}$)

function Verify(shares, $\{m_0, ..., m_i\}$)
    trigger ⟨ ss, Verify | $f$, shares, $\{m_0, ..., m_i\}$ ⟩
    $m$ := wait for event ⟨ ss, Verified | $m$ ⟩
    return $m$
```

3.6 Byzantine identification

Assuming algorithm 3.4 returned a verified secret, we have the list of shares $s_v = \{s_{v_1}, ..., s_{v_{t+2}}\}$, used in the reconstruction of the verified secret and the list of all shares $s = \{s_1, ..., s_{N+1}\}$ we received. Then we reconstruct f_k with $(s_{v_1}, ..., s_{v_{t+1}}, s_k)$ for all s_k that are in s but not in s_v and if the degree of f_k is t, then s_k is *correct*, otherwise s_k is *byzantine*. Finally, the reader revokes the permissions of all servers, that provided a *faulty* share. Since shares with wrong timestamps or x_{N+1} are ignored, slow servers will be ignored and will never be deemed byzantine. Note here, that the byzantine identification has a computational complexity of $O(N)$, compared to $O(N!)$ in Harn and Lin's [HL09] cheater identification, because we have already verified the secret. This only works in a setting, where the writer trusts the readers, otherwise the readers notify the writer about a byzantine P_i and the writer revokes P_i's permission after receiving some quorum of notifications.

3.6.1 Marking cheaters without their knowledge

Assuming we have a setting where the writer can only communicate to the readers through storing values in the servers and the writer has identified some of the byzantine servers, it might want to mark them to the readers, while the byzantine servers do not know that some of them have been marked. The intuitive approach is that the writer provides the byzantine servers forged shares, but then the readers will see the mark only after reconstruction and running the byzantine identification. We will show, that marking servers, such that readers can check the mark in $O(1)$, i.e. in a way that the readers will not include marked shares in the reconstruction, is impossible. Since algorithm 3.4 provides unconditional security, utilizing security through obscurity (e.g. $s_{i_y} \bmod 10 = 0 \xrightarrow{\text{implies}}$ mark) is unreasonable. Since the writer has only identified some byzantine servers and there are byzantine servers, that have not been identified, i.e. servers that are controlled by an adversary $\mathcal{A}$, but always provided the correct share, writing an old index or x_{N+1} is not feasible, since $\mathcal{A}$ sees different indices or x_{N+1}. Thus a mark is needed, that can only be reconstructed with more than f shares and a check can only be done minimally in $\mathrm{O}(N)$. Following from that, marking a server, while $\mathcal{A}$ cannot check the mark, but readers can check it in $\mathrm{O}(1)$ is impossible.

4 Analysis

In the following chapter, we present our results. First, we provide formal proofs that our algorithms, listed below, satisfy the properties outlined in the modules. We then compare our algorithms to two existing systems in the literature, in order to discuss the advantages and limitations of our proposed solutions.

- Algorithm 3.1: Safe register API
- Algorithm 3.2: Safe register
- Algorithm 3.3: Regular register API
- Algorithm 3.4: Regular register

4.1 Proofs

In this section, we prove that the algorithms in section 3.5 implement the modules from section 3.4 and the corresponding properties hold, as well as analyze the computational complexities of the operations of our registers. First, we prove that our APIs implement the API from the modules. Next, we analyze the computational complexities of the *read* and *write* operations of the safe register algorithm. We omit to analyze the computational complexity of the regular register algorithm because it only provides finite write termination and thus can have an infinite complexity for infinitely many writes. The computational complexity of a non-concurrent *read* of the regular register is equal to the complexity of a *read* in the safe register since both algorithms use the method of the additional share. We then prove our registers fulfill the properties of a safe and a regular register respectively. Finally, we prove that the registers provide unconditional security.

4.1.1 APIs

Theorem 4.1. *Algorithm 3.1 implements module 3.5 Application Programming Interface*

Proof. We prove that algorithm 3.1 implements the API from the module by proving each property:
Reliable requests: Requests with the commands WRITE and READ both eventually reply, due to the reliable delivery property of ClientServerLink. Any other command will be caught on the server side and will be answered with UNKNOWNCOMMAND. Thus, all requests to *correct* servers will eventually be replied to.
No duplication: The client will forward all received *ResponseCommands* and a correct server will only send one response per request and hence no request will be replied to more than once.
Authenticity: A client only replies if a command has been delivered.
Unconditional encryption: The unconditional encryption is inherited from CS4: *Unconditional Encryption*.
Client-server separation: Since servers and clients running algorithm 3.1 listen to different events, correct clients listen to requests and send replies and connect via ClientServerLink to servers to forward messages. □

Theorem 4.2. *Algorithm 3.3 implements module 3.5 Application Programming Interface*

Proof. The proof follows immediately from the proof of theorem 4.1. □

4.1.2 Registers

To prove that algorithm 3.4 implements a regular register, we first show that algorithm 3.2 implements a safe register, since both algorithms share their approach. Next, to help us prove the regular register, we first prove that algorithm 3.4 does not support termination, which helps us to finally prove that it implements a regular register.

Theorem 4.3. *Let $N > 4f$, then algorithm 3.2 implements a $(1, N)$-ByzantineSafeRegister as in module 3.6*

Proof. We prove that algorithm 3.2 implements a $(1, N)$-ByzantineSafeRegister by proving each property:
Termination: Algorithm 3.2 has two operations, read and write. It relies on the *termination* property of the secret sharing module and the API module provides termination with the *reliable delivery* property. In the *ss.DeliverShares* and *ss.Reconstructed* events, the events *cs.Send* and *ss.GenerateShares* are alternately triggered because of the *additionalShareFlag*s and thus *ss.DeliverShares* will be invoked exactly twice. Furthermore, the *cs.Deliver* events that occur in the write and read operations will eventually be invoked $N - f$ times and thus eventually trigger the *ssrr.WriteReturn* and *ss.Reconstruct* events respectively, since all correct servers will respond to *cs.Send*. Hence all operations in algorithm 3.2 eventually terminate.
Validity: Assuming a write operation with timestamp *ts* terminated and thus $N - f$ servers are informed, while f are assumed to be *correct* but uninformed. When a following read operation is requested, it waits for $N - f$ responses. Out of these responses, up to f responses are from uninformed servers and f are *faulty*. Thus if $N > 4f$, out of $N - f > 3f$ responses, f are *faulty*, f are *correct* but uninformed and more than f are *correct* and informed. Since *correct* servers only provide timestamps that are equal to or lower than *ts*, the set R cannot contain sets with cardinality greater than f and common timestamps larger than *ts*. Hence, the set Q contains more than f *correct* and informed shares. Then, the algorithm reconstructs and verifies the secret from the shares and additional shares and returns the correct secret, according to the correctness property of the secret sharing module. If the set Q is empty or contains shares from a timestamp lower than *ts*, less than $f + 1$ *correct* and informed responses have been received and it is concluded that there is a write concurrent to the read. This is because the previously assumed $f + 1$ *correct* and informed servers did not provide the same timestamp and hence a write must be happening concurrently. Since in a safe register, a read that is concurrent to a write may return anything in the domain, it is allowed to return preceding values. The same applies if the reconstruction failed, i.e. the *secret* in the *ss.Verified* event is $\perp$, and the algorithm returns the *default* value from the domain. Thus, algorithm 3.2 implements a byzantine safe register. □

Lemma 4.1. *Let $N > 3f$, then algorithm 3.4 does not support termination.*

Proof. We assume that the writer w starts a new write operation as soon as the previous write operation has been completed. In *cs.Deliver* of the read, first, the timestamps are checked for valid values, then if sets Q and R exist with the requirements, the read operation completes. Otherwise, the server is requested for the stored values again with *cs.Send*. The requirements of a set Q existing are that Q is created from exclusively the PREWRITE or WRITE values of R, has a common timestamp higher or equal to all timestamps in R and a secret can be reconstructed and verified from the values in Q. Assuming that because of the continuous writes, the reader did not receive more than f shares from the same timestamp, because every server might have shares from different writes, such a set Q is never found and the read operation continuously requests shares from the servers. Thus for infinitely many writes, algorithm 3.4 does not guarantee termination. □

Theorem 4.4. *Let $N > 3f$, then algorithm 3.4 implements a $(1, N)$-ByzantineRegularRegister as in module 3.7.*

Proof. We prove that algorithm 3.4 implements a $(1, N)$-ByzantineRegularRegister by proving each property:
Finite write termination: Similar to the termination proof of theorem 4.3 we inherit the termination properties of the modules, use the *additionalShareFlag*s, and can rely on $N - f$ responses. As we have shown in the proof of lemma 4.1, algorithm 3.4 does not support termination for infinitely many writes and we need to prove that the read operation concurrent to finitely many write operations terminates. Since algorithm 3.4 only guarantees finite write termination, we assume that a last write with timestamp *ts* exists after which no more write operations will be requested. When this write operation has completed, $N - f$ servers will have the values from *ts* stored. We need to show that the sets Q and R exist and thus a valid secret will be returned. Eventually, *readlist* will contain at least $2f + 1$ responses from *correct* servers, out of which at least $f + 1$ are informed since the write waited for

$N - f \geq 2f + 1 = (f+1)$ *correct* $+f$ *faulty* responses. Thus, a set Q with $f+1$ responses from *correct* and informed servers exists. The set R can be created with Q and the rest of the *correct* responses from *readlist* or from responses from *faulty* servers with a timestamp *ts* or lower. Since Q has at least $f+1$ *correct* and informed responses and such an R exists, a verified value can be returned, and algorithm 3.4 will terminate.
Validity: To return a value, a secret must be reconstructable and verifiable from a set Q, which can only be a value that has been written by the writer, since its length has to be strictly greater than f. Next, the reconstructed and verified secret cannot be from a write that preceded the last written value (ts, v) not concurrent to the read. This is because up to f shares might have a lower timestamp $ts' < ts$, but then no set R with length $> 2f$ can be created with timestamps ts' or lower, since R includes at least 1 *correct* and informed response. If Q contains at least $f+1$ *correct* and informed shares from the PREWRITE or WRITE storage of *readlist*, then a set R with length $2f+1$ exists with Q and f *correct* and uninformed shares, such that *authenticAndMax* with this Q, R and *readlist* will return TRUE and a secret can be verified. Otherwise, the values are read again and according to the finite write termination, the set Q will eventually contain $f+1$ *correct* and informed responses. Thus algorithm 3.4 can only return the last written value or a value concurrently written. □

4.1.3 Complexity

Theorem 4.5. *The computational complexity of a write in algorithm 3.2 is $O(N^2)$.*

Proof. The creation of $f(x)$ and $g(u)$ are both dependent on the degree $t = f$ since $f-1$ random coefficients need to be picked [2$\cdot$ $O(f-1)$]. Then $N+1$ x_i-values and N u_i-values need to be picked randomly and evaluated for $f(x)$ and $g(u)$, which is dependent on the degree f [$O((N+1)\cdot f) + O(N \cdot f)$]. Finally, the writer sends the data to each of the N servers [$O(N)$]. Since the size of $\mathbb{F}_p$ is dependent on p, i.e. the size of stored data, and not of the number of servers, picking a random number from $\mathbb{F}_p$ is done independently of N, i.e. in constant time $O(1)$, when analyzing the computational complexity in terms of servers N. Thus, the computational complexity in terms of the number of servers of a write in algorithm 3.2 is:

$$\begin{aligned}
&2 \cdot O(f-1) + O\left[(N+1)\cdot f\right] + O(N \cdot f) + O(N) \\
&\equiv O(2\cdot(f-1) + (N+1)\cdot f + N \cdot f + N) \\
&\equiv O(2\cdot(\frac{N}{3} - 1) + (N+1)\cdot\frac{N}{3} + N \cdot \frac{N}{3} + N) \\
&\equiv O(2\cdot N + N \cdot N + N \cdot N + N) \\
&\equiv O(2N + 2N^2 + N) \\
&\equiv O(N^2)
\end{aligned}$$

□

Theorem 4.6. *The computational complexity of a read in algorithm 3.2 is $O(N!^2)$.*

Proof. In the worst case, there are only $f+1$ *correct* shares and $f+1$ *correct* additional shares. Thus we have $\binom{2f+1}{f+1}$ possibilities of $g_i(u)$, of which only one is *correct* and $\binom{2f+1}{f+1}$ possibilities of finding the *correct* shares, assuming we have found the *correct* additional share. This gives us $\binom{2f+1}{f+1}\cdot\binom{2f+1}{f+1} = \binom{2f+1}{f+1}^2 = \frac{(2f+1)!^2}{(f+1)!^2 f!^2}$ total combinations to go through. Thus the complexity of the *read* is $O(f!^2) \equiv O(N!^2)$. □

To improve the complexity of the *read*, we use the fact, that $s_{N+1} = f(x_{N+1}) < p$, for the prime p that spans $\mathbb{F}_p$. We can modify the prime q, that spans $\mathbb{G}_q$, such that $q > cp$, for any $c \geq 1$. We define $g_{byz}(u)$ to be a random variable that represents the reconstructed polynomial of the additional share with at least one *faulty* share ($g_{byz}(0) \neq g(0) = f(x_{N+1})$). We know that $g_{byz}(0)$ has a uniform distribution over $\mathbb{G}_q$ since even having knowledge of up to f shares makes all possible values, i.e. $\mathbb{G}_q$, equally likely (c.f. Shamir [Sha79]). Thus $\mathbb{P}(g_{byz}(0) < p) = \frac{1}{c}$. The probability that there are j or less $g_{byz_i}(0)$ that are within $\mathbb{F}_p$, out of all reconstructed additional shares $g_{byz_i}(0)$ with at least one *faulty* share, is the sum of the Bernoulli trials up to j:

$$\mathbb{P}(|\{g_{byz_i}(0) < p\}| \leq j) = \sum_{k=0}^{j} \binom{m}{k} \left(\frac{1}{c}\right)^k \cdot \left(1 - \frac{1}{c}\right)^{m-k}$$

with m being the total amount of $g_{byz}(0)$. Since there are up to $\binom{2f+1}{f+1} - 1$ possible values for g_{byz}, we choose c to be $c \geq \binom{2f+1}{f+1}$, such that we have a low probability, that there are more one *faulty* reconstructed additional share within $\mathbb{F}_p$. Now the complexity changes. We still have $\binom{2f+1}{f+1}$ combinations for the additional share, but we can validate them individually and, with a high probability, be left with only two possibilities. One *correct* and one *faulty* additional share. Next, we need to combine each of them with the other shares of which there are $\binom{2f+1}{f+1}$ combinations. Thus the probabilistic worst-case complexity is $N!$:

$$\binom{2f+1}{f+1} + 2 \cdot \binom{2f+1}{f+1} = 3 \cdot \frac{(2f+1)!}{(f+1)!f!} \approx f! \approx N!$$

4.1.4 Unconditional security

To prove that this protocol is unconditionally secure, we assume the existence of an adversary $\mathcal{A}$ with unlimited computational power, but without access to *correct* servers. First, we formulate the proof of Shamir Secret Sharing being unconditionally secure in our notation and then we will prove that algorithm 3.4 is also unconditionally secure. We annotate a share s as (s_x, s_y), where s_x and s_y are the coordinates and an indexed share s_i to be constructed of the coordinates (s_{i_x}, s_{i_y}), where a subscripted $_x$ or $_y$ will always be the coordinate of the share and not an index.

Lemma 4.2. *Shamir's Secret Sharing is unconditionally secure*

Proof. We formulate Shamir's proof in our notation (c.f. Shamir [Sha79]).
We assume to have a secret sharing protocol according to Shamir [Sha79] with a polynomial $f(x)$ of degree t within a field $\mathbb{F}_p$ for a prime p and an adversary $\mathcal{A}$ has access to t shares (s_{1_x},s_{1_y}), ..., (s_{t_x}, s_{t_y}). Since $\mathcal{A}$ cannot directly recreate a polynomial of degree t, but only of $t-1$, it needs to brute-force a share s_{t+1} by trying all values within $\mathbb{F}_p$ to get (s_{t+1_1}, ..., s_{t+1_p}). Because $\mathcal{A}$ cannot verify the correctness of the reconstructed secret with only $t+1$ shares, as discussed in section 2.1.1, $\mathcal{A}$ needs a further share s_{t+2} to get $t+2$ shares. But for each reconstructed polynomial $f_i(x)$ with s_1, ..., s_t and s_{t+1_i}, for $i = 1, ..., p$, a further polynomial f_{i_j} can be constructed with the previous shares and any share ($s_{t+2_{j_x}}$, $s_{t+2_{j_y}}$), with $s_{t+2_{j_y}} = f_i(s_{t+2_{j_x}})$, such that $f_{i_j} = f_i$ will have a degree of t. Since the share s_{t+1_i} can be chosen randomly over $\mathbb{F}_p$, the reconstructed secret $s_i = f_i(0)$ is distributed uniformly over $\mathbb{F}_p$ and thus $\mathcal{A}$ cannot gain any information about the real secret from t or fewer shares. □

Theorem 4.7. *Algorithm 3.2 is unconditionally secure*

Proof. We assume that there is a protocol running according to Section 3.3 with a polynomial $f(x)$ of degree t within a field $\mathbb{F}_p$ for the main shares (s_{i_x}, s_{i_y}) and there is a polynomial $g(u)$ of degree t within a field $\mathbb{G}_q$ for a prime $q \geq p$ for the additional shares ($s_{N+1_{i_x}}$, $s_{N+1_{i_y}}$). An adversary $\mathcal{A}$ has control over t shares (s_{1_x},s_{1_y}), ..., (s_{t_x}, s_{t_y}) and t additional shares ($s_{N+1_{1_x}}$, $s_{N+1_{1_y}}$), ...,($s_{N+1_{t_x}}$, $s_{N+1_{t_y}}$). For the additional share, $\mathcal{A}$ only knows, that $g(0) < p$ and has no further information, as we proved theorem 4.2. Thus $\mathcal{A}$ can either brute-force a further additional share ($s_{N+1_{t+1_x}}$, $s_{N+1_{t+1_y}}$) to reconstruct the polynomial g to get the additional share or $\mathcal{A}$ can brute-force a further main share ($s_{N+1_{i+1_x}}$, $s_{N+1_{i+1_y}}$) to reconstruct the polynomial f, but for both brute-forces $\mathcal{A}$ would need to brute force an even further share, as discussed in the proof of theorem 4.2. Thus algorithm 3.2 is unconditionally secure.

□

Theorem 4.8. *Algorithm 3.4 is unconditionally secure.*

Proof. The proof follows immediately from the proof of theorem 4.7. □

4.2 Comparison to other systems

In this section, we compare the performance and functionality of our proposed safe (algorithm 3.2) and regular (algorithm 3.4) registers to two existing byzantine fault-tolerant storage systems, Belisarius and DepSky. The comparison between our registers and Belisarius is kept minimal due to the significant differences in their functionality and model. Belisarius operates as an SMR system and requires communication between servers, whereas our algorithms function as registers and operate in a cloud-of-clouds model without any need for server-server communication. On the other hand, the comparison with DepSky is more in-depth, as both our registers and DepSky have a similar cloud-of-clouds architecture.

We first analyze the similarities and differences between the models and assumptions of each system. Next, we provide an implementation of the confidentiality version (DepSky-CA) of DepSky in our event-based form for a more detailed comparison. We are not able to do the same with Belisarius, because Padilha and Pedone [PP11] do not provide pseudo-code. Additionally, we analyze the trade-offs and advantages of each system in terms of storage space, read and write complexity, message complexity, and security guarantees to evaluate the advantages and limitations of our proposed algorithms in comparison to these established systems.

4.2.1 Belisarius

Comparing Belisarius and algorithm 3.4 is difficult, as they differ significantly in terms of functionality and model. Belisarius is an SMR system that requires server-to-server communication due to a consensus protocol, while algorithm 3.4 is a regular register that operates without communication between servers. Despite these differences, a comparison is still valuable due to the shared client-server architecture and similar data storage method.
When analyzing the models, both of them work in a server-client environment and assume an adversary with unlimited computational power. While Belisarius works in a near-optimal $N > 3f + 1$ byzantine setting, we achieve the optimal $N > 3f$ for algorithm 3.4. Since Belisarius implements an SMR it expects the servers to be able to execute code including communication with other servers for the total order broadcast, which makes it impossible for Belisarius to work in a cloud-of-clouds model without adaptation. On the other hand, Belisarius can easily be reduced to a regular register by only using *read* and *write* for a key. The operations *compare* and *add* are not needed to implement a regular register. Thus, Belisarius provides more functionality than algorithm 3.4.
Algorithm 3.4 achieves the stricter $N > 3f$ requirement by storing an additional share, which increases the amount of storage space required. In the worst case scenario for algorithm 3.2, where our optimization is used, it requires disk space in the order of $N!$, since the additional share is in $\mathbb{G}_q$ and q is in the order of $N!$ bigger than p. Even when not using the optimization, we require twice as much disk space because of the additional share. Algorithm 3.4 requires double the space compared to algorithm 3.2 since it stores a PREWRITE as well as a WRITE. Hence our algorithms require significantly more disk space than Belisarius. Both Belisarius and our algorithms provide unconditional security through secret sharing and assuming our optimization is used, both have a read complexity of $\mathrm{O}(N!)$. However, Belisarius implements an atomic register with the consensus protocol of the communication protocol, while our algorithm implements a safe register for algorithm 3.2 and a regular register for 3.4. This comes at the cost of increased message complexity, as Belisarius requires $\mathrm{O}(N^2)$ messages to be sent (c.f. Castro and Loskov [CL02]), whereas algorithm 3.2 has a message complexity of $2 \cdot (N - f) = \mathrm{O}(N)$ for a write and non-concurrent read, and the message complexity of algorithm 3.4 is $3 \cdot (N - f) = \mathrm{O}(N)$.
In conclusion, Belisarius requires less disk space, has a lower computational complexity, and offers more functionality. But the model of algorithm 3.4 is less restrictive with $N > 3f$, not requiring server code execution and no inter-server communication. Thus algorithm 3.4 works in a cloud-of-clouds setting, which removes the need for custom servers, reduces operational cost, and only requires the development of a client program. Also, the message complexity of algorithm 3.4 is lower, because it does not need a consensus protocol.

4.2.2 DepSky

The model used in DepSky is similar to ours by representing the cloud-of-clouds approach and requiring $N > 3f$. Their cloud-of-clouds approach is more restrictive and does not allow a timestamp check for writes. Since Bessani et al. [BCQ$^+$13] provide pseudo-code, we will transform their pseudo-code to our event-based form, after implementing corresponding the API with the metadata and the file storage similar to algorithm 3.3. The DepSky-CA API in algorithm 4.1 implements four commands, WRITEMETA, READMETA, WRITE and READ, with which the metadata file, consisting of a timestamp, digest, and signature, as well as the stored file can be written and read. The implementations of the API commands in algorithm 4.1 are kept as simple as possible by exclusively storing and retrieving the corresponding values.
DepSky allows the usage of several data units, which we could implement with several instances of algorithm 3.4, that do not share the same storage on the servers. Hence, our implementation of DepSky in algorithm 4.2 will not use data units or can be assumed to use a fixed data unit. Additionally, DepSky assumes that pending requests can be canceled, which our model of APIs can not support, because of the reliable requests property. Our algorithms solve the canceling of requests by only listening to events with the corresponding id field. For example with every operation, our client creates a locally unique identification (*wts* or *rts*) that is sent with every request and the client only listens to events with the corresponding identification, thus a cancellation is not necessary, since old events are ignored. DepSky-CA uses the methods *queryMetadata* and *cloud$_i$.get* for communication to the servers, which we

do not implement as functions because we desire an event-based form and thus implement them directly into the *write* and *read*.

Algorithm 4.1 DepSky-CA API

```
Implements:
    ApplicationProgrammingInterface, instance dsapi, with writer $w$.

Uses:
    ClientServerLink, instance cs

//Server only
upon event ⟨ dwapi, Init ⟩ do
    (ts, digest, signature) := (0, $\perp$, $\perp$)
    storage := $\emptyset$ // contains block of file and share of encryption key

//Client only
upon event ⟨ dsapi, Request | $q$, id, command, $m$ ⟩ do
    trigger ⟨ cs, Send | $q$, id, command, $m$ ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, ts, WRITEMETA, (ts, digest', signature' ⟩ do
    (ts, digest, signature) := (ts', digest', signature')
    trigger ⟨ cs, Send | $p$, ts, METAACK ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, ver, WRITE, ($e$, $s$) ⟩ do
    storage[ver] := ($e$, $s$)
    trigger ⟨ cs, Send | $p$, ver, ACK ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, READMETA ⟩ do
    trigger ⟨ cs, Send | $p$, METARETURN, (ts, digest, signature) ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, ver, READ ⟩ do
    trigger ⟨ cs, Send | $p$, ver, READRETURN, (storage[ver].$e$, storage[ver].$s$) ⟩

//Server only
upon event ⟨ cs, Deliver | $p$, id, command, $m$ ⟩ such that Command $\notin$ {WRITEMETA, WRITE, READMETA, READ} do
    trigger ⟨ cs, Send | $p$, id, UNKNOWNCOMMAND ⟩

//Client only
upon event ⟨ cs, Deliver | $q$, id, command, $m$ ⟩ do
    trigger ⟨ dwapi, Reply | $q$, id, command, $m$ ⟩
```

Algorithm 4.2 DepSky-CA Algorithm (Part 1: Write)

```
Implements:
    (1,$N$)-ByzantineRegularRegister, instance dsrr, writer $w$.

Uses:
    ApplicationProgrammingInterface, instance dsapi.

upon event ⟨ dsrr, Init ⟩ do
    $\Omega := \{S_1, ..., S_N\}$
    $N := |\Omega|$
    $f := \frac{|\Omega|-1}{3}$
    $(max\_ver, new\_ver, max\_id) := (0, 0, 0)$
    $K^{du}_{rw}$ := create signing key
    $(h, acklist, m, d) := (\emptyset, \emptyset, \emptyset, \emptyset)$

//Writer $w$ only
upon event ⟨ dsrr, Write | value ⟩ do
    if $max\_ver$ = 0 then
        $m := \emptyset$
        forall $q \in \Omega$ do
            trigger ⟨ dsapi, Request | $q$, READMETA ⟩
    else
        write(value)

//Writer $w$ only
upon event ⟨ dsapi, Reply | $q$, METARETURN, (ts, digest, signature) ⟩ do
    if verify(ts, digest, signature) = TRUE then
        $m[q]$ := (ts, digest, signature)
    if $|m| = N - f$ then
        $max\_ver$ := max($m.ts$)
        write(value)

//Writer $w$ only
function write(value)
    $new\_ver := max\_ver + 1$
    $k$ := generateSecretKey()
    $e$ := encrypt(value, $k$)
    $s$ := generateShares($k$, $N$, $f + 1$)
    $v$ := encode($e$, $N$, $f + 1$)
    forall $q \in \Omega$ do
        $d[q].e := v[q]$)
        $d[q].s := s[q]$
        $h[q]$ := hash($d[q]$)
        trigger ⟨ dsapi, Request | $q$, new_ver, WRITE, ($d[q].e$, $d[q].s$) ⟩

//Writer $w$ only
upon event ⟨ dsapi, Reply | $q$, NEW_VER, ACK ⟩ do
    acklist[$q$] := ACK
    if |acklist|$= N - f$ then
        acklist := $[\ \perp\ ]^N$
        new_meta := (new_ver, $h$)
        new_meta_signed := sign(new_meta, $K^{du}_{rw}$)
        forall $q \in \Omega$ do
            trigger ⟨ dsapi, Request | $q$, new_ver, WRITEMETA, new_meta_signed ⟩

//Writer $w$ only
upon event ⟨ dsapi, Reply | $q$, NEW_VER, METAACK ⟩ do
    acklist[$q$] := ACK
    if |acklist|$= N - f$ then
        acklist := $[\ \perp\ ]^N$
        max_ver := new_meta
        trigger ⟨ dsrr, WriteReturn ⟩
```

Algorithm 4.2 DepSky-CA Algorithm (Part 2: Read)

```
upon event ⟨ dsrr, Read ⟩ do
    $m := \emptyset$
    forall $q \in \Omega$ do
        trigger ⟨ dsapi, Request | $q$, READMETA ⟩

upon event ⟨ dsapi, Reply | $q$, METARETURN, (ts, digest, signature) ⟩ do
    if verifySignature(ts, digest, signature) = TRUE then
        $m[q]$ := (ts, digest, signature)
    if $|m| = N - f$ then
        max_id := max($m$.ts)
        $d := \emptyset$
        forall $q \in \Omega$ do
            trigger ⟨ dsapi, Request | $q$, max_id, READ ⟩

upon event ⟨ dsapi, Reply | $q$, MAX_ID, READRETURN, ($tmp_q.e$, $tmp_q.s$) ⟩ do
    $h_{tmp_q}$ := hash(($tmp_q.e$, $tmp_q.s$))
    if $h_{tmp_q}$ := $m$[max_id].digest then
        $d[q]$ := ($tmp_q.e$, $tmp_q.s$)
    else
        $d[q]$ := ERROR
    if $(|\{i : d[i] \neq \perp \wedge d[i] \neq \text{ERROR}\}| > f) \vee (|\{i : d[i] \neq \perp\}| > N - f)$ then
        if $(|\{i : d[i] \neq \perp \wedge d[i] \neq \text{ERROR}\}| > f)$ then
            $e$ := decode($d.e$, $N$, $f + 1$)
            $k$ := combineShares($d.s$, $N$, $f + 1$)
            value := decrypt($e$, $k$)
            trigger ⟨ dsapi, ReadReturn | value ⟩
        else
            $d := \emptyset$
            forall $q \in \Omega$ do
                trigger ⟨ dsapi, Request | $q$, max_id, READ ⟩
```

The DepSky API (Algorithm 4.1) and our APIs (Algorithms 3.1 and 3.3) differ in one detail. DepSky assumes that servers only store and retrieve data, while our APIs assume that servers will not overwrite stored data with older data from a slow connection with a timestamp check. Both DepSky and Algorithm 3.4 implement a regular register and wait for $f + 1$ *correct* and informed responses, and both only support finite write termination. However, DepSky-CA verifies each response individually with a digest, while Algorithm 3.4 requires $f + 1$ responses to start the verification of the responses, due to the properties of our secret sharing scheme.

Algorithm 3.4 uses a two-step process to ensure the validity of the read operation. It first creates two sets, one called Q which is used for the reconstruction of the secret and one called R which is used to determine if the servers in Q are informed or not. If the algorithm is able to verify the secret using the shares in Q, it will return the secret. However, if the algorithm is unable to verify the secret using Q, it will query the server, for which all responses are checked, again for the currently stored value. In contrast, DepSky uses a different approach. It waits for more than f correct responses or responses from $N - f$ servers in the second **if** condition in the READRETURN event. If it receives more than f correct and informed responses, it returns the value. If this is not the case but it receives $N - f$ responses, it will query all servers again for their currently stored value. This means that algorithm 3.4 will repeatedly query fast servers until it receives enough responses, while DepSky will wait for $N - f$ responses before querying the servers again. Algorithm 3.4 could be optimized to not unnecessarily request fast servers repeatedly by adapting the second **if** condition in its READRETURN by splitting the condition into two steps. First, the existence of the set R can be checked and if it does not exist, nothing will be done. If it exists, the remaining conditions are checked and if those are not met, all servers are queried again.

It should be noted that Bessani et al. [BCQ$^+$13] do not mention whether DepSky-CA is conditionally or unconditionally secure. The security of the algorithm lies in four places. The file is encrypted and hashed, the metadata including the hash and timestamp is signed and the encryption key is secret-shared. The encryption, the signature as well as the secret sharing can be made unconditionally secure, but the hash cannot be made unconditionally secure. Thus, an adversary with unlimited computational power could provide a *faulty* block with the same hash as the *correct* one and thus prevent the reader from reconstructing the file. Hence, DepSky-CA is only conditionally secure.

The computational complexity of a *write* operation in DepSky-CA is $\mathrm{O}(N^2)$, which includes $N - f$ verifications of the metadata signatures [$\mathrm{O}(N)$], the generation of N shares [$\mathrm{O}(N^2)$] and the erasure-coding for N blocks [$\mathrm{O}(N^2)$]. All other operations, such as encrypting one file or signing one metadata file, do not depend on the number of servers N. In contrast, the *read* operation in DepSky-CA has a computational complexity of $\mathrm{O}(N^2)$, which includes $N - f$ verifications of signatures [$\mathrm{O}(N)$], taking the maximum of $N - f$ responses [$\mathrm{O}(N)$], $N - f$ hashes [$\mathrm{O}(N)$], the erasure-decoding of N blocks [$\mathrm{O}(N^2)$], and the reconstruction of the encryption key from $N - f$ shares [$\mathrm{O}(N^2)$]. Furthermore, the space complexity of DepSky-CA grows with the number of writes, since each version is stored separately, which could be formulated as a versioning feature from a user's perspective, but is not a feature usually required in distributed storage. While algorithm 3.4 is restricted to one file but could be easily adapted to support multiple files.

In conclusion, DepSky is designed for a real-world application with the data units and versioning of files, while algorithm 3.4 focuses on the theoretical aspect of a regular register. DepSky-CA has a lower computational complexity but sacrifices security as it only provides conditional security. Given that the computational complexity of both algorithm 3.4 and DepSky-CA is analyzed depending on the number of servers, N, it would be noteworthy to quantitatively compare their complexity differences. This is of particular interest since DepSky was designed and tested with four servers ($N = 4$) in mind and therefore, there might only be a small increase in computational cost in exchange for added security through an unconditionally secure system. However, an experimental analysis falls outside of the scope of this thesis.

5 Conclusion

This thesis presents the development of an approach for creating a verifiable secret sharing scheme with unconditional security in a cloud-of-clouds model with $N > 3f$, which is optimal for the byzantine setting. Previous schemes either do not support unconditional security or require $N > 3f + 1$. Our approach secret-shares an additional share to have $2f + 2$ shares for the verified reconstruction with only $2f + 1$ participants, thus relaxing the $N > 3f + 1$ requirement down to $N > 3f$. Hence, the problem statement from section 3.2 and example 3.1 can now be solved in a $N > 3f$ model. Client R receives $3 = 2f + 1$ responses, out of which $f + 1$ are *correct* and reconstructs the additional share, resulting in $f + 2$ *correct* shares and R can return v_1. Figure 5.1.1 adapts figure 3.2.1 from example 3.1, such that the additional share is stored as well, and demonstrates that the *correct* value v_1 can now be returned.

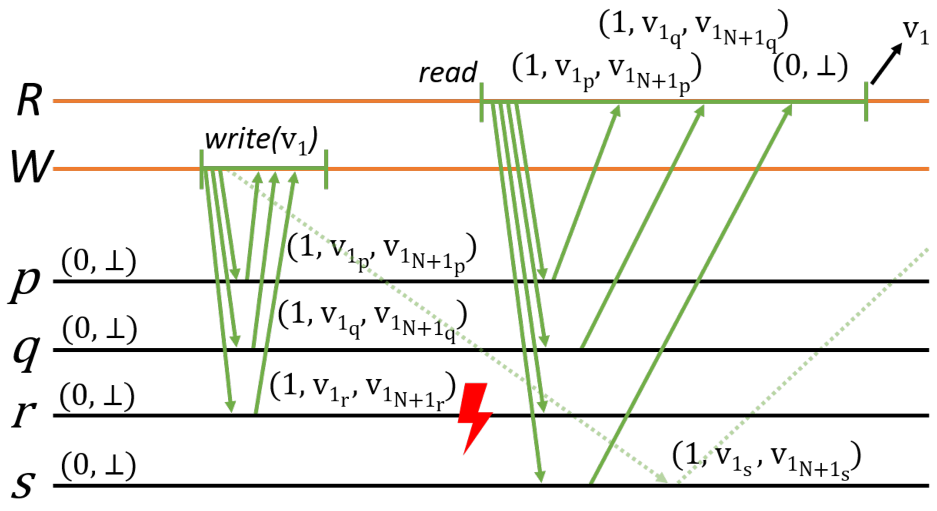

Figure 5.1.1: The additional share scheme

In order to utilize the cloud-of-clouds model, we modeled the behavior of a cloud service API in a modular way with powerful properties. This modular structure serves as a foundation for future research in the cloud-of-clouds model and can easily be relaxed to conditional security, if future research does not have such high security requirements. We also successfully created a safe and regular register from our approach as well as their corresponding APIs, that can be used in further algorithms. We provided thorough proofs of the code complexity, safety and regularity of the registers and unconditional security of our algorithms. To improve the functionality of our SWMR regular register, it could be transformed into MWMR regular register, such that all clients are allowed to write. Advancing

the functionality even further, it could be developed into an MWMR atomic register, as Cachin et al. [CGR11] have shown how a $(1, N)$-regular register can be first transformed into a $(1, 1)$-atomic register, then into a $(1, N)$-atomic register and finally into a (N, N)-atomic register. We also discussed the identification of byzantine servers and the impossibility of marking them without their knowledge with our requirements in a $(1, N)$ register in the cloud-of-clouds model.
An algorithm, that is unconditionally secure, is future-proof since it is mathematically shown, that no matter how fast computers may become, the algorithm will never be compromise, as long as the assumptions hold. It does not matter, what will get invented or developed, it is theoretically safe to be used forever. Our algorithms can easily detect *byzantine* servers with a low computational overhead, and depending on the permission system used, their permissions can be revoked. As we have shown, marking the *byzantine* servers, such that they can not notice it is infeasible.

Bibliography

[ACKM06] ABRAHAM, Ittai ; CHOCKLER, Gregory V. ; KEIDAR, Idit ; MALKHI, Dahlia: Byzantine disk paxos: optimal resilience with byzantine shared memory. In: *Distributed Comput.* 18 (2006), Nr. 5, 387–408. `http://dx.doi.org/10.1007/s00446-005-0151-6`. – DOI 10.1007/s00446–005–0151–6

[Ama22] AMAZON: *AWS Service Terms*. 2022. – `https://aws.amazon.com/service-terms/` [Accessed: Oct. 2022]

[APW10] ABU-LIBDEH, Hussam ; PRINCEHOUSE, Lonnie ; WEATHERSPOON, Hakim: RACS: a case for cloud storage diversity. In: HELLERSTEIN, Joseph M. (Hrsg.) ; CHAUDHURI, Surajit (Hrsg.) ; ROSENBLUM, Mendel (Hrsg.): *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, ACM, 2010, 229–240

[BCQ+13] BESSANI, Alysson N. ; CORREIA, Miguel ; QUARESMA, Bruno ; ANDRÉ, Fernando ; SOUSA, Paulo: DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In: *ACM Trans. Storage* 9 (2013), Nr. 4, 12. `http://dx.doi.org/10.1145/2535929`. – DOI 10.1145/2535929

[BJO08] BOWERS, Kevin D. ; JUELS, Ari ; OPREA, Alina: HAIL: A High-Availability and Integrity Layer for Cloud Storage. In: *IACR Cryptol. ePrint Arch.* (2008), 489. `http://eprint.iacr.org/2008/489`

[CGR11] CACHIN, Christian ; GUERRAOUI, Rachid ; RODRIGUES, Luís E. T.: *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011 `https://doi.org/10.1007/978-3-642-15260-3`. – ISBN 978–3–642–15259–7

[CHLT14] CHEN, Henry C. H. ; HU, Yuchong ; LEE, Patrick P. C. ; TANG, Yang: NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds. In: *IEEE Trans. Computers* 63 (2014), Nr. 1, 31–44. `http://dx.doi.org/10.1109/TC.2013.167`. – DOI 10.1109/TC.2013.167

[CHV10] CACHIN, Christian ; HAAS, Robert ; VUKOLIC, Marko: Dependable storage in the intercloud / IBM research. 2010 (3783). – Research Report

[CL02] CASTRO, Miguel ; LISKOV, Barbara: Practical byzantine fault tolerance and proactive recovery. In: *ACM Trans. Comput. Syst.* 20 (2002), Nr. 4, 398–461. `http://dx.doi.org/10.1145/571637.571640`. – DOI 10.1145/571637.571640

[Clo13] CLOUD SECURITY ALLIANCE: *The Notorious Nine Cloud Computing Top Threats in 2013*. 2013

[CT05] CACHIN, Christian ; TESSARO, Stefano: Asynchronous Verifiable Information Dispersal. In: FRAIGNIAUD, Pierre (Hrsg.): *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings* Bd. 3724, Springer, 2005 (Lecture Notes in Computer Science), 503–504

[DH76] DIFFIE, Whitfield ; HELLMAN, Martin E.: New directions in cryptography. In: *IEEE Trans. Inf. Theory* 22 (1976), Nr. 6, 644–654. `http://dx.doi.org/10.1109/TIT.1976.1055638`. – DOI 10.1109/TIT.1976.1055638

[Dro22] DROPBOX: *Privacy Policy - Dropbox*. 2022. – `https://www.dropbox.com/privacy` [Accessed: Oct. 2022]

[Fel87] FELDMAN, Paul: A Practical Scheme for Non-interactive Verifiable Secret Sharing. In: *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, IEEE Computer Society, 1987, 427–437

[Gho11] GHODOSI, Hossein: Comments on Harn-Lin's cheating detection scheme. In: *Des. Codes Cryptogr.* 60 (2011), Nr. 1, 63–66. `http://dx.doi.org/10.1007/s10623-010-9416-6`. – DOI 10.1007/s10623–010–9416–6

[Goo22] GOOGLE: *Google Terms of Service - Privacy & Terms - Google*. 2022. – `https://policies.google.com/terms?hl=en#toc-permission/` [Accessed: Oct. 2022]

[Har13] HARN, Lein: *The Generalization of Harn-Lin's Scheme on Cheater Detection and Identification*. 2013

[HL09] HARN, Lein ; LIN, Changlu: Detection and identification of cheaters in (t , n) secret sharing scheme. In: *Des. Codes Cryptogr.* 52 (2009), Nr. 1, 15–24. `http://dx.doi.org/10.1007/s10623-008-9265-8`. – DOI 10.1007/s10623–008–9265–8

[HS08] HERLIHY, Maurice ; SHAVIT, Nir: *The art of multiprocessor programming*. Morgan Kaufmann, 2008. – ISBN 978–0–12–370591–4

[KL14] KATZ, Jonathan ; LINDELL, Yehuda: *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. – ISBN 9781466570269

[Lam86] LAMPORT, Leslie: On Interprocess Communication. Part I: Basic Formalism. In: *Distributed Comput.* 1 (1986), Nr. 2, 77–85. `http://dx.doi.org/10.1007/BF01786227`. – DOI 10.1007/BF01786227

[Mic22] MICROSOFT: *Microsoft Services Agreement*. 2022. – `https://www.microsoft.com/en/servicesagreement/` [Accessed: Oct. 2022]

[MOC$^+$21] MENDES, Ricardo ; OLIVEIRA, Tiago ; COGO, Vinicius V. ; NEVES, Nuno ; BESSANI, Alysson: Charon: A Secure Cloud-of-Clouds System for Storing and Sharing Big Data. In: *IEEE Trans. Cloud Comput.* 9 (2021), Nr. 4, 1349–1361. `http://dx.doi.org/10.1109/TCC.2019.2916856`. – DOI 10.1109/TCC.2019.2916856

[PP11] PADILHA, Ricardo ; PEDONE, Fernando: Belisarius: BFT Storage with Confidentiality. In: *Proceedings of The Tenth IEEE International Symposium on Networking Computing and Applications, NCA 2011, August 25-27, 2011, Cambridge, Massachusetts, USA*, IEEE Computer Society, 2011, 9–16

[RC11] ROCHA, Francisco ; CORREIA, Miguel: Lucy in the sky without diamonds: Stealing confidential data in the cloud. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, S. 129–134

[Sha49] SHANNON, Claude E.: Communication theory of secrecy systems. In: *Bell Syst. Tech. J.* 28 (1949), Nr. 4, 656–715. `http://dx.doi.org/10.1002/j.1538-7305.1949.tb00928.x`. – DOI 10.1002/j.1538–7305.1949.tb00928.x

[Sha79] SHAMIR, Adi: How to Share a Secret. In: *Commun. ACM* 22 (1979), Nr. 11, 612–613. `http://dx.doi.org/10.1145/359168.359176`. – DOI 10.1145/359168.359176

[WFZ$^+$11] WADA, Hiroshi ; FEKETE, Alan D. ; ZHAO, Liang ; LEE, Kevin ; LIU, Anna: Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In: *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, www.cidrdb.org, 2011, 134–143

List of Figures

List of Modules

List of Algorithms