UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXII° Cycle

SSD: ING-INF/05

Protecting Resources and Regulating Access in Centralized and Decentralized Cloud Systems

Advisor
Prof. Stefano Paraboschi

Doctoral Thesis
Enrico BACIS
Student ID 1006904

Academic year 2018/2019

Abstract

The low costs and high reliability guarantees associated with cloud storage led many IT organizations to offload their data to cloud service providers. Yet, this raises new challenges to manage access control and data confidentiality in these environments.

The first part of this doctoral thesis analyzes centralized cloud storage providers (CSP). In this setting, the CSP is considered honest-but-curious: it always complies with users' requests, but it might access unprotected data. A possible defense is to encrypt the data; however, standard encryption modes would introduce relevant overheads when performing access revocation. To address this problem, we present an approach that relies on a resource transformation that provides strong mutual inter-dependency in its encrypted representation. To revoke access on a resource, it is then sufficient to update a small portion of it.

The second part studies how these guarantees can be extended to the decentralized cloud storage environment. In this case, the data is sharded and offloaded in a peer-to-peer network, in which nodes might be dishonest and try to disobey users' deletion and access revocation requests to maximize their revenue. We propose a solution that addresses both availability and security guarantees and enables resource owners to tune these settings to their needs.

When dealing with decentralized networks, an important aspect is how to detect misbehaving nodes, to stop paying for their service and migrate the data to new peers. This process has to work even when the data owner is offline and without imposing trust or honesty assumption on any of the involved parties. To address this problem, in the third part of this thesis, we detail a novel way of deploying self-releasing time-locked secrets. This technique can be used to implement delegated challenge-response protocols that, in turn, can guarantee data confidentiality and retrievability properties in fully distributed systems. Our solution leverages smart contracts and economic incentives to regulate a game among the mutually distrusting users that compose a blockchain, thus removing the need of any trusted party.

The technologies detailed in this thesis push the state of the art as regards resource protection and access regulation in centralized and decentralized cloud storage systems. The implementations have been released under open-source licenses and can be readily integrated with real systems.

Contents

List of Figures

a Giusi e Gian

1

Introduction

1.1 Document structure

This thesis is organized in five chapters.

Chapter 1 illustrates the document structure and the publications that set the basis for this thesis.

Chapter 2 describes an encryption mode, named *Mix&Slice*, that enables efficient access revocation on resources stored at external cloud providers [14]. By leveraging a technique known as *All-Or-Nothing Transform*, *Mix&Slice* allows the data owner to perform access revocation by re-encrypting a small portion of the file.

The chapter is organized as follows.

- Section 2.1 presents the setting, discusses the trust assumptions, together with existing solutions and their limitations.
- Section 2.2 details the *Mix&Slice* encryption mode, an approach to get an encrypted representation with the desired protection guarantees.
- Section 2.3 presents the enforcement of access revocation.
- Section 2.4 discusses the effectiveness of our solution.

- Section 2.5 illustrates our implementation and the extensive experimental evaluation confirming its advantages and applicability.
- Section 2.6 discusses the related work.
- Section 2.7 concludes the chapter.

Chapter 3 presents an approach enabling resource owners to effectively protect and securely delete their resources while relying on decentralized cloud services for their storage [17, 18]. Our solution combines All-Or-Nothing Transform (such as the Mix&Slice [14] algorithm detailed in Chapter 2) and carefully designed allocation strategies for slicing resources and distributing them in the decentralized storage. We address both availability and security guarantees, jointly considering them in our model and enabling resource owners to tune these settings independently.

The chapter is organized as follows.

- Section 3.1 describes the decentralized cloud storage scenario, its challenges and opportunities.
- Section 3.2 introduces the basic concepts.
- Section 3.3 defines the properties of a decentralized allocation function with respect to replication and protection properties.
- Section 3.4 discusses slicing and allocation strategies.
- Section 3.5 illustrates availability and security guarantees and discusses the setting of parameters guiding slicing and allocation.
- Section 3.6 illustrates the implementation of our approach on a real decentralized cloud storage service and presents the experimental results.
- Section 3.7 discusses related work.
- Section 3.8 concludes the chapter.

- The proofs of theorems are provided in Appendix B.

Chapter 4 presents a novel way of implementing time-locked secrets. Time-Locks enable the release of secret data at a specific future point in time. Our vision is that this will be a fundamental piece to be able to fully exploit the power of decentralized cloud storage. The use of time-locked secrets enables the creation of delegated challenge-response protocols, which can be used to provide properties such as access revocation and resiliency in decentralized cloud storage networks. Current research efforts mostly bind the recovery of the secret with the solution of cryptographic puzzles. These solutions, however, are impractical: not only they require the interested parties to undergo a significant computational effort to solve the puzzle, but also they provide no precise timing guarantees. To address these problems, we propose *I* Told *You Tomorrow (ITYT)*, a novel way of implementing time-locked secrets based on smart contracts to remove the need of any trusted party.

The chapter is organized as follows.

- Section 4.1 presents the settings and discusses the alternatives.
- Section 4.2 introduces the basic concepts we build our proposal on.
- Section 4.3 describes the protocol from a high-level perspective.
- Section 4.4 illustrates how to impose constraints on the parameters to protect the resulting protocol from rational adversaries.
- Section 4.5 describes the details of our reference implementation based on the Ethereum blockchain and the FRESCO sMPC framework.
- Section 4.6 discusses the possible attacks the implementation can be subject to and the respective countermeasures.
- Section 4.7 illustrates the experimental evaluation.
- Section 4.8 discusses related work.
- Section 4.9 concludes the chapter.

Chapter 5 draws the conclusions of the thesis and discusses future work.

1.2 Publications

This section presents the list of publications authored during the Ph.D. course and that set the basis for this thesis.

Articles in journals

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, Pierangela Samarati. "**Securing Resources in Decentralized Cloud Storage**". *IEEE Transactions on Information Forensics and Security, vol. 15, n. 1.* 2019.

Papers in proceedings of international conferences

- Enrico Bacis, Sabrina de Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, Pierangela Samarati. "**Access Control Management for Secure Cloud Storage**". *12th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM).* EAI, 2016.

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Daniele Guttadoro, Stefano Paraboschi, Marco Rosa, Pierangela Samarati, Alessandro Saullo. "**Managing Data Sharing in OpenStack Swift with Over-Encryption**". *3rd Workshop on Information Sharing and Collaborative Security (WISCS).* ACM, 2016.

- Enrico Bacis, Sabrina de Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, Pierangela Samarati. "**Mix&Slice: Efficient Access Revocation in the Cloud**". *23rd ACM Conference on Computer and Communications Security (CCS).* ACM, 2016.

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, Pierangela Samarati, "**Dynamic Allocation for Resource Protection in Decentralized Cloud Storage**". *Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM).* IEEE, 2019.

- Enrico Bacis, Dario Facchinetti, Marco Rosa, Marco Guarnieri, Stefano Paraboschi, "**I *Told* You Tomorrow: Practical Time-Locked Secrets using Smart Contracts**". *(under submission)*.

Chapters in books

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, Pierangela Samarati. "**Protecting Resources and Regulating Access in Cloud-Based Object Storage**". *From Database to Cyber Security*. 2018.

2

Mix&Slice: Efficient Access Revocation in the Cloud

This chapter presents an approach to enforce access revocation on resources stored at external cloud providers. The approach relies on a resource transformation that provides strong mutual inter-dependency in its encrypted representation. To revoke access on a resource, it is then sufficient to update a small portion of it, with the guarantee that the resource as a whole (and any portion of it) will become unintelligible to those from whom access is revoked. The extensive experimental evaluation on a variety of configurations confirmed the effectiveness and efficiency of our solution, which showed excellent performance and compatibility with several implementation strategies.

2.1 Introduction

With the considerable advancements in IT solutions, users and companies are finding increasingly appealing to rely on external services for storing resources and making them available to others. In such contexts, a promising approach to enforce access control to externally stored resources is via encryption: resources are encrypted for storage and only authorized users have the keys that enable their decryption. There are several advantages that justify the use of encryption for enforcing access control. First, robust encryption has become computationally inexpensive, enabling its introduction in domains that are traditionally extremely sensitive to performance (like cloud-based applications and management of large resources). Second, encryption provides protection against the service provider itself, which - while trustworthy for providing access - cannot typically be considered authorized to know the content of the resources it stores (*honest-but-curious* scenario) and hence also to enforce access control. Third, encryption solves the need of having a trusted party for policy enforcement: resources enforce self-protection, since only authorized users, holding the keys, will be able to decrypt them.

One of the complex aspects in using encryption to enforce access control policy concerns access revocation. If granting an authorization is easy (it is sufficient to give the newly authorized user access to the key), revoking an authorization is a completely different problem. There are essentially two approaches to enforce revocation: *i)* re-encrypt the resource with a new key or *ii)* revoke access to the key itself. Re-encryption of the resource entails, for the data owner, downloading the resource, decrypting it and re-encrypting it with a new key, re-uploading the resource, and re-distributing the key to the users who still hold authorizations. If decryption, re-encryption, and even key management (for this specific context) can be considered a trivial issue, the remaining challenge is represented by the need to download and re-upload the resource, with a considerable overhead for the data owner. This overhead will continue to grow as usage of cloud resources grows, in particular in the context of emerging big data applications. The alternative approach of enforcing revocation on the resource by preventing access to the key with which the resource is encrypted cannot be considered a solution. As a matter of fact, it protects the key, not the resource itself, and it is inevitably fragile against a user who - while having been revoked from an access - has maintained a local copy of the key.

Our approach. In this chapter, we present a novel approach to enforce access revocation that provides efficiency, as it does not require expensive upload/re-upload of (large) resources, and robustness, as it is resilient against the threat of users who might have maintained copies of the keys protecting resources on which they have been revoked access.

The basic idea of our approach is to provide an encrypted representation of the resources that guarantees complete interdependence (*mixing*) among the bits of the encrypted content. Such a guarantee is ensured by using different rounds of encryption, while carefully selecting their input to provide complete mixing, meaning that the value of each bit in the resulting encrypted content depends on every bit of the original plaintext content. In this way, unavailability of even a small portion of the encrypted version of a resource completely prevents the reconstruction of the resource or even of portions of it. Brute-force attacks guessing possible values of the missing bits are possible, but even for small missing portions of the encrypted resource, the required effort would be prohibitive. The *all-or-nothing transform* (AONT) [102] considers similar requirements, but the techniques proposed for it are not suited to our scenario, because they are based on the assumption that keys are not known to users, whereas in our scenario revoked users can know the encryption key and may plan ahead to locally store critical pieces of information.

Our approach trades off between the requirement to connect all bits of a resource (to provide the desired interdependency of the content), and the requirement to maintain fine grained access of the resource itself. This is a particular challenge due to the potentially huge size of the resources. To achieve this, we apply the idea of mixing content within portions of the resource, enforcing then revocation by overwriting encrypted bits in every such portion. Before mixing, our approach partitions the resource in different, equally sized, chunks, called *macro-blocks*. Then, as the name hints, it is based on the following concepts.

- *Mix:* the content of each macro-block is processed by an iterative application of different encryption rounds together with a carefully designed bit mixing, that ensures, at the end of the process, that every individual bit in the input has had impact on each of the bits in the encrypted output.

- *Slice:* the mixed macro-blocks are sliced into fragments so that fragments provide complete coverage of the resource content and each fragment represents a minimal (in terms of number of bits of protection, which we call *mini-block*) unit of revocation: lack of any single fragment of the resource completely prevents reconstruction of the resource or of portions of it.

To revoke access from a user, it is sufficient to re-encrypt one (any one) of the resource fragments with a new key not known to the user. The advantage is clear: re-encrypting a tiny chunk of the resource guarantees protection of the whole resource itself. Also, the cloud provider simply needs to provide storage functionality and is not required to play an active role for enforcing access control or providing user authentication. Our *Mix&Slice* proposal is complemented with a convenient approach for key management that, based on key regression, avoids any storage overhead for key distribution.

2.2 Mix&Slice

2.2.1 Blocks, mini-blocks, and macro-blocks

The basic building block of our approach is the application of a symmetric block cipher. A symmetric cryptographic function operating on blocks guarantees complete dependency of the encrypted result from every bit of the input and the impossibility, when missing some bits of an encrypted version of a block, to retrieve the original plaintext block (even if parts of it are known). The only possibility to retrieve the original block would be to perform a brute-force attack attempting all the possible combinations of values for the missing bits. For instance, modern encryption functions like AES guarantee that the absence of i bits from the input (plaintext) and of o bits from the output (ciphertext) does not permit, even with knowledge of the encryption key k, to properly reconstruct the plaintext and/or ciphertext, apart from performing a brute-force attack generating and verifying all the $2^{\min(i,o)}$ possible configurations for the missing bits [7].

Clearly, the larger the number of bits that are missing in the encrypted version of a block, the harder the effort required to perform a brute-force attack, which requires attempting 2^x possible combinations of values when x bits are missing. Such *security parameter* is at the center of our approach and we explicitly identify a sequence of bits of its length as the atomic unit on which our approach operates, which we call *mini-block*. Applying block encryption with explicit consideration of such atomic unit of protection, and extending it to a coarser-grain with iterative rounds, our approach identifies the following basic concepts.

- *Block*: a sequence of bits input to a block cipher (it corresponds to the classical block concept).
- *Mini-block*: a sequence of bits, of a specified length, contained in a block. It represents our *atomic unit* of protection (i.e., when removing bits, we will operate at the level of mini-block removing all its bits).
- *Macro-block*: a sequence of blocks. It allows extending the application of block cipher on sequences of bits larger than individual blocks. In particular, our approach operates *mixing* bits at the macro-block level, extending protection to work against attacks beyond the individual block.

Our approach is completely parametric with respect to the size (in terms of the number of bits) that can be considered for blocks, mini-blocks, and macro-blocks. The only constraints are for the size of a mini-block to be a divisor of the size of the block (aspect on which we will elaborate later on) and for the size of a macro-block to be a product of the size of a mini-block and a power of the number of mini-blocks in a block (i.e., the ratio between the size of a block and the size of a mini-block). In the following, for concreteness and simplicity of the figures, we will illustrate our examples assuming the application of AES with blocks of 128 bits and mini-blocks of 32 bits, which corresponds to having 4 mini-blocks in every block and therefore operating on macro-blocks of size $32 \cdot 4^x$, with x arbitrarily set. In the following, we will use *msize*, *bsize*, *Msize* to denote the size (in bits) of mini-blocks, blocks,

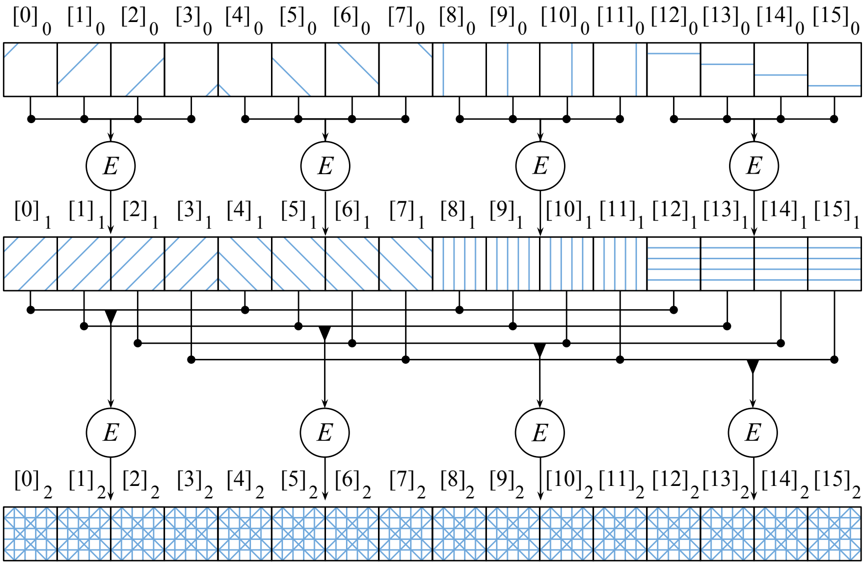


Figure 2.1: An example of mixing of 16 mini-blocks assuming $m = 4$

and macro-blocks, respectively. We will use $b_j[i]$ ($\mathrm{M_j}[i]$, resp.) to denote the i-th mini-block in a block b_j (macro-block $\mathrm{M_j}$, resp.). We will simply use notation $[i]$ to denote the i-th mini-block in a generic bit sequence (be it a block or macro-block), and $[[j]]$ to denote the j-th block. In the encryption process, a subscript associated with a mini-block/block denotes the round that produced it.

2.2.2 Mixing

The basic step of our approach (on which we will iteratively build to provide complete mixing within a macro-block) is the application of encryption at the block level. This application is visible at the top of Figure 2.1, where the first row reports a sequence of 16 mini-blocks ($[0], \ldots, [15]$) composing 4 blocks. The second row is the result of block encryption on the sequence of mini-blocks. As visible from the pattern-coding in the figure, encryption provides mixing within each block so that each mini-block in the result is dependent on every mini-block in the same input block. In other words, each $[i]_1$ is dependent on every $[j]_0$ with (i div 4) = (j div 4).

One round of block encryption provides mixing only at the level of block. With reference to our example, mixing is provided among mini-blocks $[0]_0 \ldots [3]_0$, $[4]_0 \ldots [7]_0$, $[8]_0 \ldots [11]_0$, and $[12]_0 \ldots [15]_0$, respectively. Absence of a mini-block from the result will prevent reconstruction only of the plaintext block including it, while not preventing the reconstruction of all the other blocks. For instance, with reference to our example, absence of $[2]_1$ will prevent reconstruction of the first block (mini-blocks $[0]_0, \ldots, [3]_0$) but will not prevent reconstruction of the other three blocks (mini-blocks $[4]_0, \ldots, [15]_0$). Protection at the block level is not sufficient in our context, where we expect to manage resources of arbitrarily large size and would like to provide the guarantee that the lack of any individual mini-block would imply the impossibility (apart from performing a brute-force attack) of reconstructing any other mini-block of the resource.

The concept of macro-block, and its extension of block ciphers to operate across blocks, allows us to provide mixing on an arbitrarily long sequence of bits. The idea is to extend mixing to the whole macro-block by the iterative application of block encryption on, at each round, blocks composed of mini-blocks that are representative (i.e., belong to the result) of different encryptions in the previous round. Before giving the general definition of our approach, let us discuss the simple example of two rounds illustrated in Figure 2.1, where $[0]_1, \ldots, [15]_1$ are the mini-blocks resulting from the first round. The second round would apply again block encryption, considering different blocks each composed of a representative of a different computation in the first round. To guarantee such a composition, we define the blocks input to the four encryption operations as composed of mini-blocks that are at distance 4 (=*m*) in the sequence, which corresponds to say that they resulted from different encryption operations in the previous round. The blocks considered for encryption would then be $\langle[0]_1[4]_1[8]_1[12]_1\rangle$, $\langle[1]_1[5]_1[9]_1[13]_1\rangle$,$\langle[2]_1[6]_1[10]_1[14]_1\rangle$,$\langle[3]_1[7]_1[11]_1[15]_1\rangle$. The result would be a sequence of 16 mini-blocks, each of which is dependent on each of the 16 original mini-blocks, that is, the result provides mixing among all 16 mini-blocks, as visible from the pattern-coding in the figure. With 16 mini-blocks, two rounds of encryption suffice for guaranteeing mixing among all of them. Providing mixing for larger sequences clearly requires more rounds. This

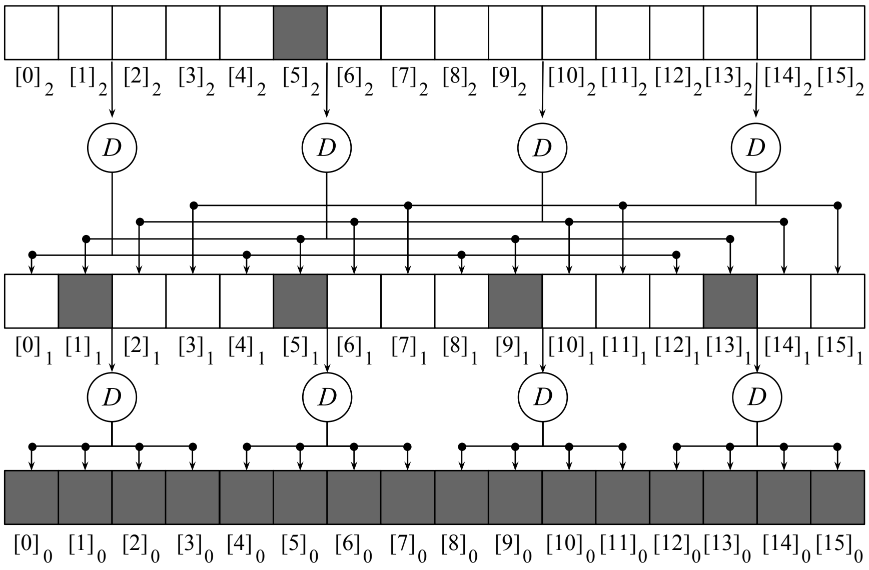


Figure 2.2: An example of un-mixing of 16 mini-blocks assuming $m = 4$

brings us to the general formulation of our approach operating at the level of macro-block of arbitrarily large size (the example just illustrated being a macro-block of 16 mini-blocks).

Absence of a mini-block from the result will prevent reconstruction of the whole plaintext. With reference to our example in Figure 2.2, absence of $[5]_2$ will prevent reconstruction of the block $\langle[1]_1[5]_1[9]_1[13]_1\rangle$, which in turn will prevent reconstruction of the macro-blocks $[0]_0, \ldots, [15]_0$.

To ensure the possibility of mixing, at each round, blocks composed of mini-blocks resulting from different encryption operations of the previous round, we assume a macro-block composed of a number of mini-blocks, which is the power of the number (*m*) of mini-blocks in a block. For instance, with reference to our running example where blocks are composed of 4 mini-blocks (i.e., *m*=4), macro-blocks can be composed of 4^x mini-blocks, with an arbitrary x (x=2 in the example of Figure 2.1). The assumption can be equivalently stated in terms of blocks, where the number of blocks *b* will be 4^{x-1}. Any classical padding solution can be employed to guarantee such a requirement, if not already satisfied by the original bit sequence in input.

Providing mixing of a macro-block composed of *b* blocks with *b*=*m*$^{x-1}$ requires x rounds of encryption each composed of *b* encryptions. Each round

```
**Mix**($\mathrm{M}$)

for $i := 1, \ldots, x$ do                                  /* at each round $i$ */
  *span* := $m^i$                                  /* number of mini-blocks in a mixing */
  *distance* := $m^{i-1}$                    /* leg of mini-blocks input to an encryption */
  for $j := 0, \ldots, b-1$ do                            /* each $j$ is an encryption */
                                /* identify the input to the $j$-th encryption picking, */
                                 /* within each span, mini-blocks at leg *distance* */
    let *block* be the concatenation of all mini-blocks $[l]$
      s.t. $(l \mod distance) = j$ and
        $(j \cdot m)$ div *span* = $l$ div *span*
    $[[j]]_i := E(k, block)$             /* write the result as the $j$-th block in output */
```

Figure 2.3: Mixing within a macro-block M

allows mixing among a number *span* of mini-blocks that multiplies by *m* at every round. At round i, each encryption j takes as input *m* mini-blocks that are within the same *span* (i.e., the same group of m^i mini-blocks to be mixed) and at a *distance* (m^{i-1}). Figure 2.3 illustrates the mixing procedure. To illustrate, consider the example in Figure 2.1, where blocks are composed of 4 mini-blocks (*m*=4) and we have a macro-block of 16 mini-blocks, that is, 4 blocks (*b*=4). Mixing requires $x = 2$ rounds of encryption ($16 = 4^2$), each composed of 4 (*b*) encryptions operating on 4 (*m*) mini-blocks. At round 1, the *span* is 4 (i.e., mixing operates on chunks of 4 mini-blocks) and mini-blocks input to an encryption are taken at distance 1 within each span. At round 2, the *span* is 16 (all mini-blocks are mixed) and mini-blocks input to an encryption are taken at *distance* 4 within each *span*. Let us consider, as an another example, a macro-block composed of 64 mini-blocks (i.e., 16 blocks). Mixing requires 3 rounds. The first two rounds would work as before, with the second round producing mixing within chunks of 16 mini-blocks. The third round would then consider a *span* of all the 64 mini-blocks and mini-blocks input to an encryption would be the ones at *distance* 16.

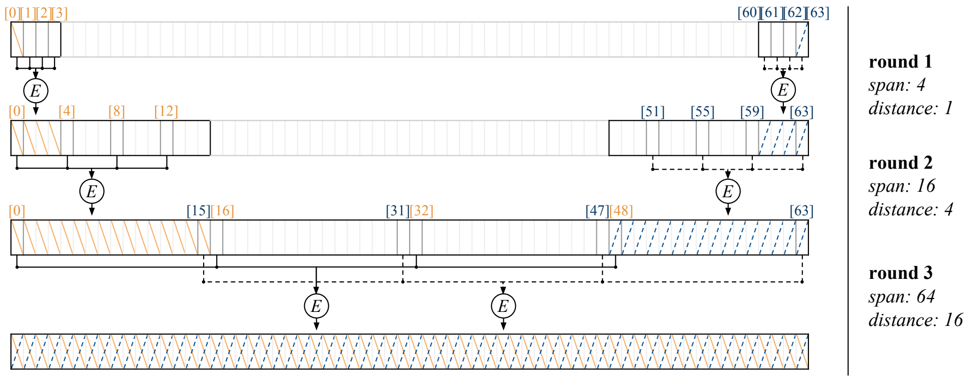


Figure 2.4: Propagation of the content of mini-blocks [0] and [63] in the mix process

At each round i, mini-blocks are mixed among chunks of m^i mini-blocks, hence ensuring at round x, mixing of the whole macro-block composed of m^x mini-blocks.

Figure 2.4 captures this concept by showing the mixing of the content of the first ([0]) and last ([63]) mini-blocks of the macro-block at the different rounds, given by the encryption to which they (and those mini-blocks mixed with them in previous rounds) are input, showing also how the two meet at the step that completes the mixing. While for simplicity the figure pictures only propagation of the content of two mini-blocks, note that at any step they (just like other mini-blocks) actually carry along the content of all the mini-blocks with which they mixed in previous rounds. Given a macro-block $\mathbb{M}$ with m^x mini-blocks (corresponding to b blocks), the following two properties hold: *1)* a generic pair of mini-blocks $[i]$ and $[j]$ mix at round r with i div m^r = j div m^r; and *2)* x rounds bring complete mixing. In other words, the number of encryption rounds needed to mix a macro-block with $m \cdot b$ mini-blocks is $\log_m(m \cdot b)$.

An important feature of the mixing is that the number of bits that are passed from each block in a round to each block in the next round is equal to the size of the mini-block. This guarantees that the uncertainty introduced by the absence of a mini-block at the first round (2^{msize}) maps to the same level of uncertainty for each of the blocks involved in the second round, and iteratively to the next rounds, thanks to the use of AES at each iteration. This implies that a complete mixing of the macro-block requires at least $\log_m(m{\cdot}b)$ rounds, that is, the rounds requested by our technique.

Another crucial aspect is that the representation after each round has to be of the same size as the original macro-block. In fact, if the transformation produced a more compact representation, there would be a possibility for a user to store this compact representation and maintain access to the resource even after revocation (this is a weakness of other solutions discussed in Section 2.6). Since, in our approach, each round produces a representation that has the same macro-block size, the user has no benefit in aiming to attack one round compared to another (see Section 2.4).

We note that an interpretation of the proposed mixing is that it extends the ability of protecting the correspondence between input and output of a block cipher to blocks of arbitrary size. An alternative approach that we considered to obtain this result was based on the use of a Feistel architecture [79], which is known to be an effective technique for the construction of block ciphers. The approach uses, as the *round* function of the Feistel architecture, a block cipher. The approach can be applied iteratively, doubling the block size at every iteration. The analysis we performed showed that this approach would lead to less efficiency compared to the solution proposed in this chapter, with a number of invocations of the basic block cipher equal to $2 \cdot \log_m(m \cdot b)$. The Feistel-based approach can be adopted when the mini-block size desired for security goes beyond the block size of the available block cipher. Similarly, symmetric cryptosystems operating on large blocks can support larger mini-blocks and also reduce the number of rounds of our approach. For instance, AESQ [25,26] shuffles 4 AES blocks and could be used as a 512-block cipher in our structure.

2.2.3 OAEP mixing

Another approach that provides strong inter-dependency in the representation of a resource is *Optimal Asymmetric Encryption Padding* (OAEP) [21, 22] proposed by Bellare and Rogaway. OAEP is a Feistel network [50] that uses a pair of cryptographically one-way trapdoor functions, G and H, to process the plaintext before encryption. The OAEP process is represented in Figure 2.5.

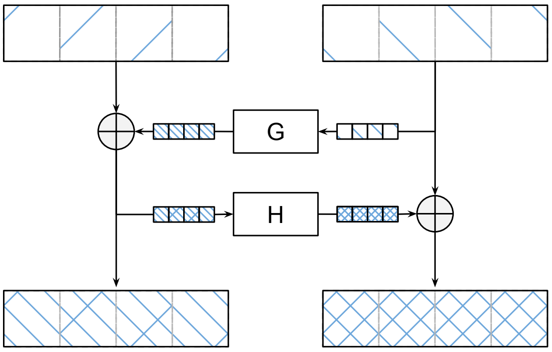


Figure 2.5: Classical OAEP does not evenly mix the plaintext

Boyko proved that OAEP can be used to build an All-or-Nothing Transform that is resistant to chosen-plaintext attacks [30]. However, the classical OAEP schema does not imply an *Avalanche Effect* [114]: the change of one bit in the left half of the plaintext, impacts the whole right half, but only affects the corresponding bit in the left half, as shown in Figure 2.5. Luby and Rackoff showed in [79] that an adaptation of OAEP composed of three rounds instead of two, guarantees that the change of any bit in the plaintext has the chance of affecting any bit in the ciphertext.

Another limitation of the use of OAEP as an AONT is that by construction the output size is $|G_{out}| + |H_{out}|$. A trivial technique that can be adopted to extend the output of the functions to fit plaintext of any size is to use G_{out} and H_{out} to seed stream ciphers (e.g., RC4) that then produce the required number of bits needed for the exclusive-or operation. Even if efficient, this schema would not be as strong as the original *Mix&Slice*. In fact, the two seeds G_{out} and H_{out}, which are compact in size, can be stored by the adversary to trivially invert the schema even when part of the ciphertext is missing, thus breaking the All-or-Nothing property.

To solve this problem, we can leverage the *Mix* algorithm detailed in Section 2.2.2 and replace the encryption function with a 3-rounds OAEP. We can use $G = H = SHA2$ and impose $|G_{in}| = |G_{out}|$ and $|H_{in}| = |H_{out}|$, so that by using $SHA2_{256}$ we can achieve $msize = 256$ and *bsize* $= 512$ (each round mixes two mini-blocks). If we need even more protection we can use $SHA2_{512}$ and obtain $msize = 512$ and *bsize* $= 1024$. OAEP does not provide message secrecy, so it is required to encrypt the output of the process. In our implementation, we used *AES* in *CTR mode* to encrypt the output of the OAEP process.

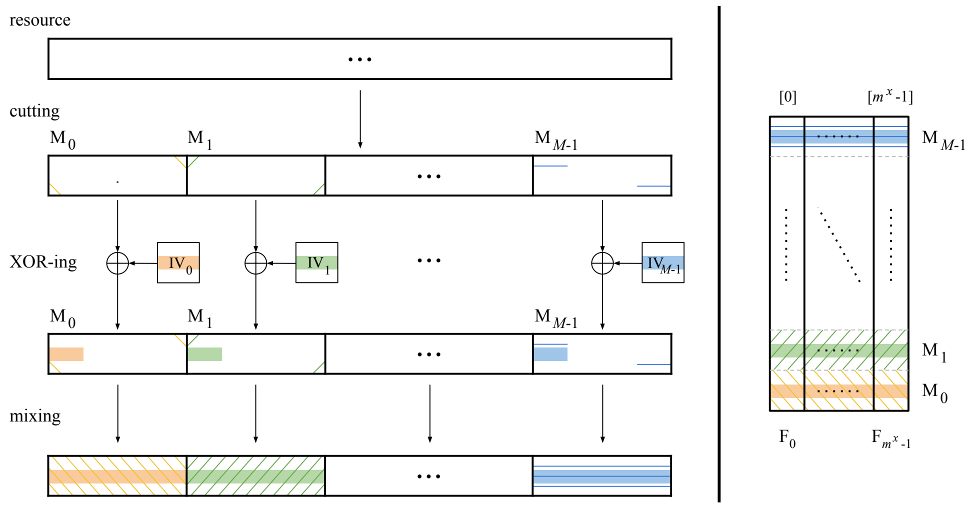
As it will be described in details in Section 2.5.1, the use of AES is computationally efficient due to its hardware implementation available in most of the modern CPUs, however it caps the mini-block size to 64 bits. On the other hand, OAEP with SHA2 does not benefit from hardware implementation, thus it has a lower throughput, yet it enables mini-blocks to be as big as 512 bits, thus enabling the use of *Mix&Slice* in scenarios that require higher protection from brute-force attacks.

2.2.4 Shortcomings of large macro-blocks

When resources are extremely large (or when access to a resource involves only a portion of it) considering a whole resource as a single macro-block may be not desirable. Even if only with a logarithmic dependence, the larger the macro-block the more the encryption (and therefore decryption to retrieve the plaintext) rounds required. Also, encrypting the whole resource as a single macro-block implies its complete download at every access, when this might actually not be needed for service.

Accounting for this, we do not assume a resource to correspond to an individual macro-block, but assume instead that any resource can be partitioned into M macro-blocks, which can then be mixed independently. The choice of the size of macro-blocks should take into consideration the performance requirements of both the data owner (for encryption) and of clients (for decryption), and the possible need to serve fine-grained retrieval of content. This requirement can be then efficiently accommodated independently encrypting (i.e., mixing) different portions of the resource, which can be downloaded and processed independently (we will discuss this in Section 2.5.2).

Encryption of a resource would then entail a preliminary step cutting the resource in different, equally sized, macro-blocks on which mixing operates. To ensure the mixed versions of macro-blocks be all different, even if with the same original content, the first block of every macro-block is XORed with an *initialization vector* (*IV*) before starting the mixing process. Since mixing



(a) (b)

Figure 2.6: From resource to fragments

guarantees that every block in a macro-block influences every other block, the adoption of a different initialization vector for each macro-block guarantees indistinguishability among their encrypted content. The different initialization vectors for the different blocks can be obtained by randomly generating a vector for the first macro-block and then incrementing it by 1 for each of the subsequent macro-blocks in the resource, in a way similar to the CTR mode [49]. Figure 2.6(a) illustrates such process.

2.2.5 Slicing

The starting point for introducing mixing is to ensure that each single bit in the encrypted version of a macro-block depends on every other bit of its plaintext representation, and therefore that removing any one of the bits of the encrypted macro-block would make it impossible (apart from brute-force attacks) to reconstruct any portion of the plaintext macro-block. Such a property operates at the level of macro-block. Hence, if a resource (because of size or need of efficient fine-grained access) has been partitioned into different macro-blocks, removal of a mini-block would only guarantee protection of the macro-block to which it belongs, while not preventing reconstruction of the other macro-blocks (and therefore partial reconstructions of the resource).

Resource protection can be achieved if, for each macro-block of which the resource is composed, a mini-block is removed. This observation brings to the second concept giving the name to our approach, which is *slicing*. Slicing the encrypted resource consists in defining different *fragments* such that a fragment contains a mini-block for each macro-block of the resource, no two fragments contain the same mini-block, and for every mini-block there is a fragment that contains it. To ensure all this, as well as to simplify management, we slice the resource simply putting in the same fragment the mini-blocks that occur at the same position in the different macro-blocks. Slicing and fragments are defined as follows.

Definition 2.2.1 (Slicing and fragments) *Let* R *be a resource and* $\mathrm{M}_0, \ldots, \mathrm{M}_{M-1}$ *be its (individually mixed) macro-blocks, each composed of* $(m \cdot b)$ *mini-blocks. Slicing produces* $(m \cdot b)$ *fragments for* R *where* $\mathrm{F}_i = \langle \mathrm{M}_0[i], \ldots, \mathrm{M}_{M-1}[i] \rangle$*, with* $i = 1, \ldots, (m \cdot b)$*.*

Figure 2.6(b) illustrates the slicing process and Figure 2.7 illustrates the procedure for encrypting a resource R. R is first cut into *M* macro-blocks and an initialization vector is randomly chosen. The first block of each macro-block is then XOR-ed with the initialization vector, which is incremented by 1 for each macro-block. The macro-block is then encrypted with a mixing process (Figure 2.3). Encrypted macro-blocks are finally sliced into fragments.

2.3 Access management

Accessing a resource (or a macro-block in the resource, resp.) requires availability of all its fragments (its mini-blocks in all the fragments, resp.), and of the key used for encryption. Policy changes corresponding to granting access to new users can be simply enforced, as usual, by giving them the encryption key. In principle, policy changes corresponding to revocation of access would instead normally entail downloading the resource, re-encrypting it with a new key, re-uploading the resource, and distributing the new encryption key to all the users who still hold authorizations. Our approach enables the enforcement of access revocation to a resource by simply making any of its fragments un-

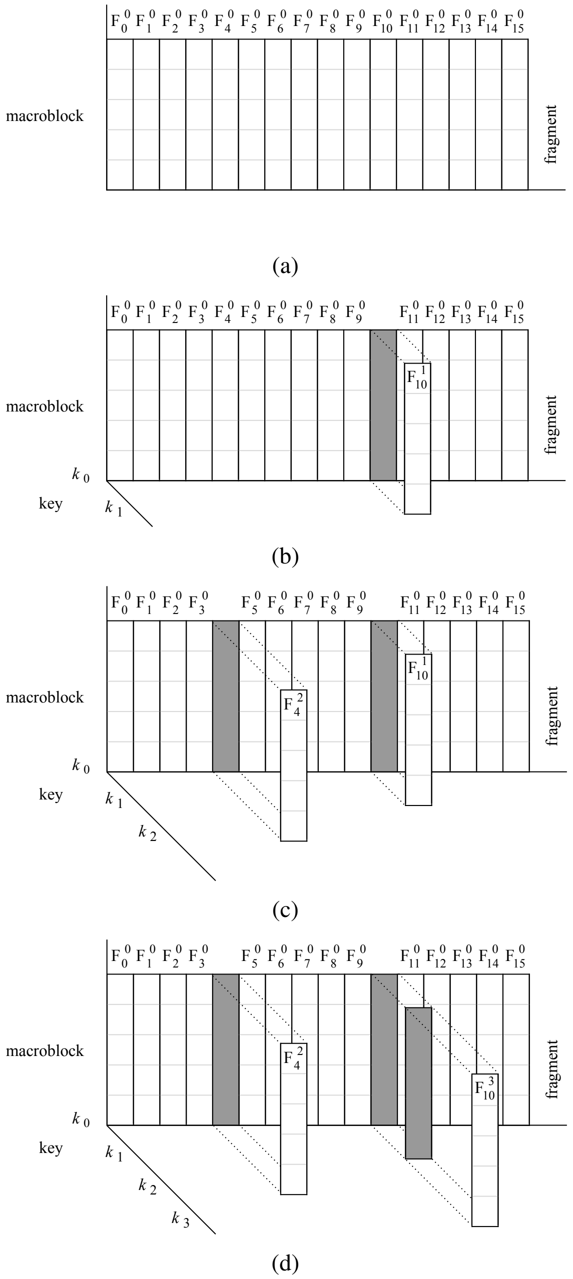
Encrypt

```
cut R in $M$ macro-blocks $\mathrm{M}_0,\ldots,\mathrm{M}_{M-1}$
apply padding to the last macro-block $\mathrm{M}_{M-1}$
$IV$ := randomly choose an initialization vector
for $i = 0,\ldots,M-1$ do                          /* encrypt macro-blocks */
    $\mathrm{M}_i[[1]] := \mathrm{M}_i[[1]] \oplus IV$           /* XOR the first block with the IV */
    Mix($\mathrm{M}_i$)                                 /* encrypt the macro-block */
    $IV := IV + 1$              /* initialization vector for the next macro-block */
    for $j = 0,\ldots,m^x-1$ do                            /* slicing */
        $\mathrm{F}_j[i] := \mathrm{M}_i[j]$
```

Figure 2.7: Algorithm for encrypting a resource R

available to the users from whom the access is revoked. Since lack of a fragment implies lack of a mini-block for each macro-block of a resource, and lack of a mini-block prevents reconstruction of the whole macro-block, lack of a fragment equates to complete inability, for the revoked users, to reconstruct the plaintext resource or any portion of it. In other words, it equates to revocation.

Access revocations are then enforced by the data owner by randomly picking a fragment, which is then downloaded, re-encrypted with a new key (which will be made known only to users still authorized for the access), and re-uploaded at the server overwriting its previous version. While still requesting some download/re-upload, operating on a fragment clearly brings large advantages (in terms of throughput) with respect to operating on the whole resource (see Section 2.5). Revocation can be enforced on any randomly picked fragment (even if already re-written in a previous revocation) and a fresh new key is employed at every revoke operation. Figure 2.8 illustrates an example of fragments evolution due to the enforcement of a sequence of revoke operations. Figure 2.8(a) is the starting situation with the original fragments computed as illustrated in Section 2.2. Figure 2.8(b-d) is the sequence of rewriting to enforce revocations, which involve, respectively, fragment F_{10},



F_0^0 F_1^0 F_2^0 F_3^0 F_4^0 F_5^0 F_6^0 F_7^0 F_8^0 F_9^0 F_{10}^0 F_{11}^0 F_{12}^0 F_{13}^0 F_{14}^0 F_{15}^0
macroblock
fragment
F_{10}^1
F_4^2
F_{10}^3
key
k_0
k_1
k_2
k_3

(a)

(b)

(c)

(d)

Figure 2.8: An example of fragments evolution

re-encrypted with key k_1, fragment F_4, re-encrypted with key k_2, and fragment F_{10} again, now re-encrypted with key k_3. In the following, we use notation F_i^j to denote a version of fragment F_i encrypted with key k_j, being F_i^0 the version of the fragment obtained through the mixing process. In the figure, the resource is represented in a three-dimensional space, with axes corresponding to fragments, macro-blocks, and keys. The re-writing of a fragment is represented by placing it in correspondence to the new key used for its encryption. The shadowing in correspondence to the previous versions of the fragments denote the fact that they are not available anymore as they are overwritten by the new versions.

Each revoke operation requires the use of a fresh new key and, due to policy changes, fragments of a resource might be encrypted with different keys. Such a situation does not cause any complication for key management, which can be conveniently and efficiently handled with a *key regression* technique [52]. Key regression is an RSA-based cryptographically strong technique (the generated keys appear as pseudorandom) allowing a data owner to generate, starting from a seed s_0, an unlimited sequence of symmetric keys $k_0, \ldots, k_u$, so that simple knowledge of a key k_i (or the compact secret seed s_i of constant size related to it) permits to efficiently derive all keys k_j with $j \leq i$. Only the data owner (who knows the private key used for generation) can perform forward derivation, that is, from k_i, derive keys following it in the sequence (i.e., k_z with $z \geq i$). Note instead that, not knowing the private key, users cannot perform forward derivation. The cost that users must pay for key derivation is small. On a single core, the computer we used for the experiments is able to process several hundred thousand key derivations per second.

With key regression, every user authorized to access a resource just needs to know the seed corresponding to the most recent key used for it (s_0 if the policy has not changed, s_3 in the example of Figure 2.8(d)). To this end, there is no need for key distribution, rather, such a seed can be stored in the resource descriptor and protected (encrypted) with a key corresponding to the resource's *acl* (i.e., known or derivable by all authorized users) [9, 43]. Enforcing revocation entails then, besides re-encrypting a randomly picked fragment with a fresh new key k_i, rewriting its corresponding seed s_i, encrypted with a key associated with the new *acl* of the resource. Figure 2.9 illustrates the revocation process.

To access a resource, a user then first downloads the resource descriptor, to retrieve the most recent seed s_l, and all the fragments. With the seed, she computes the keys necessary to decrypt fragments that have been overwritten, to retrieve their version encrypted with k_0. Then, she combines the mini-blocks in fragments to reconstruct the macro-blocks in the resource. She then applies mixing in decrypt mode to macro-blocks to retrieve the plaintext resource. Figure 2.10 illustrates the process to access a resource.

Note that the size of macro-blocks influences the performance of both revoke and access operations. Larger macro-blocks naturally provide greater policy update performance as they decrease policy update cost linearly, with limited impact on the efficiency of decryption, since its cost increases logarithmically (Section 2.5).

2.4 Effectiveness of the approach

In this section, we elaborate on the effectiveness of our approach for enforcing revocation. For the discussion, we recall that *msize* is the size of individual mini-blocks, *m* is the number of mini-blocks in a block, *b* is the number of blocks in a macro-block, and *M* is the number of macro-blocks. Also, we denote with *f* the number of fragments, that is, $f = m \cdot b$.

We consider the threat coming from a user whose access to the resource has been revoked, and who downloads the resource from the server. With access policy enforced by encryption, not being authorized for an access should not prevent downloading the resource but rather it should prevent reconstruction of its plaintext representation. We then evaluate the protection against the user's attempts to reconstruct the plaintext resource. In doing so, we consider the worst case scenario, with respect to key management, where the user has maintained memory of the last key (or the corresponding seed) used for the resource up to the point in which she was authorized for the access. In other words, we assume the user to be able to decrypt the fragments that have been overwritten before she has been revoked access, and hence to know the original version encrypted with k_0 of the fragments that have not been overwritten since she has been revoked access. Since seeds are compact, such a threat

Revoke

```
randomly select a fragment $\mathrm{F_i}$ of R        /* fragment to be rewritten */
download $\mathrm{F}_i^c$ from the server             /* version of the fragment stored */
**if** $c > 0$ **then**                               /* $\mathrm{F}_i^0$ has been overwritten in a revocation */
    derive key $k_c$                                  /* derive $k_c$ using key regression */
    $\mathrm{F}_i^0 := \mathrm{D}(k_c, \mathrm{F}_i^c)$   /* retrieve the original version of the fragment */
determine the last key $k_{l-1}$ used                 /* it is stored in R's descriptor */
generate new key $k_l$
$\mathrm{F}_i^l := \mathrm{E}(k_l, \mathrm{F}_i^0)$
upload $\mathrm{F}_i^l$ overwriting $\mathrm{F}_i^c$  /* overwrite previous version */
encrypt $s_l$ with the key of $acl$(R)                /* limits it to authorized users */
update R's descriptor                                 /* including the new $s_l$ */
```

Figure 2.9: Revoke on resource R

is indeed realistic. To reconstruct the resource when missing a fragment, the user would have to perform a brute force attack attempting all possible combinations of values of the missing bits, that is, 2^{msize} attempts for each of the *M* macro-blocks. If more fragments, let's say f_{miss}, are missing, the user would have to perform $2^{msize \cdot f_{miss}}$ attempts for each of the *M* macro-blocks.

The inability of the user to reconstruct a resource if some fragments have been overwritten is because, without such fragments, the user cannot retrieve the corresponding original version (the one encrypted with k_0) needed to correctly reconstruct the resource plaintext. A potential threat can then come if the user maintains a local storage with the original version of part of the resource. We distinguish two cases, depending on whether the user stores complete fragments or portions of them across the whole resource.

2.4.1 Local storage of fragments

Suppose a user locally stores (when authorized) some fragments of the resource. Even if such fragments are later overwritten for revoking access to the user, and then their most recent version stored at the server is unintelligible to her, she has them available for reconstructing the resource. However, the fragment to be overwritten in a policy revocation is chosen randomly by the owner. Therefore, the user can still reconstruct the resource after one fragment has been overwritten if the fragment that the owner has overwritten is the same fragment that the user has also stored locally, which has probability $1/f$ to occur. Generalizing the reasoning to the consideration of the user locally storing more than one fragment and the policy naturally changing even after the specific user revocation, we determine the probability P_A of the user's ability to access the resource assuming local storage of f_{loc} fragments to be $P_A = (f_{loc}/f)^{f_{miss}}$. The probability clearly increases with the number of fragments stored locally, but quickly reaches extremely low values after a few updates of the policy, approximating zero even for high percentage of fragments locally stored. The low probability (and the high storage effort requested to the user) essentially makes such attack not suitable: if the user has to pay a storage cost that approaches the maintenance of the whole resource, then the user would have stored the plaintext resource when authorized in the first place. We note also that a possible extension of our approach could consider overwriting, instead of pre-defined fragments, a randomly chosen set of mini-blocks (ensuring coverage of all macro-blocks), to enforce a revocation. In this case, the probability of the user storing m_{loc} mini-blocks per macro-block (also randomly chosen) to be able to access the resource immediately after her revocation would be $(m_{loc}/(m \cdot b))^M$, which would become $(m_{loc}/(m \cdot b))^{M \cdot m_{miss}}$, (i.e., negligible), if she misses m_{miss} mini-blocks per macro-block. We note however that overwriting randomly picked mini-blocks across the resource would considerably increase the complexity in the management of fragments, and it would make it harder to provide an efficient physical structure for fragments (Section 2.5). Given the observations above about the high storage cost that would be required to the user and the low probability of her success as policy changes, we argue that the regular structure for the fragments is preferable.

Access

```
download R's descriptor and all its fragments
retrieve seed $s_l$ used for the last encryption
compute keys $k_0, \ldots, k_l$
for each downloaded fragment $F_i^x$ do
    if $x > 0$ then
        $F_i^0 := D(k_x, F_{\mathrm{i}}^x)$          /* retrieve the original version of fragments */
for $j = 0, \ldots, M-1$ do                          /* reconstruct and decrypt macro-blocks */
    $M_j$ := concatenation of mini-blocks $F_i^0[j]$, $i = 0, \ldots, (m \cdot b) - 1$
    decrypt $M_j$
```

Figure 2.10: Access to resource R

2.4.2 Keeping portions of all mini-blocks

Instead of locally storing some selected fragments, a user can opt for using storage to maintain portions of all the mini-blocks in each fragment. In this case, whatever the fragment overwritten in the revocation, the user will have to perform some effort to realize a brute-force attack to retrieve the missing bits (she does not have the complete fragment), but such an effort will be lower, given the availability of the locally stored bits. For instance, assuming the user to keep 50% (i.e., half of the bits) of each mini-block, the effort for reconstructing the resource given a missing fragment would now be $2^{(msize/2)}$ attempts for each of the M macro-blocks (in contrast to the 2^{msize} required if all the bits in the fragment were unknown). However, again, if more fragments are missing, the required effort would quickly escalate, being equal to $2^{(msize/2) \cdot f_{miss}}$ when f_{miss} fragments are missing. For each attempt, the verification that a guess is correct would require to apply all the decryption rounds until the plaintext is reconstructed, with a great cost. We note that the user can cut down on such cost if she locally maintains, in addition to the portions of the original mini-blocks, also some bits of the partial results of the computation (which would allow her to test correctness of a guess without performing all the encryption rounds). Availability of such partial results can help testing the guesses for a mini-block if the other mini-blocks in the same block are available (i.e., when the user misses only one fragment per block). However, from the birthday paradox, we note that the probability of two revocations hitting the same block (but a different fragment) quickly increases with the number of revocations. Then, after a few updates the advantage of the user keeping partial results of the computation will become ineffective. In addition to this, we note that, in this case as well, the storage and computational efforts required to the user do not seem to make this attack much preferable for her with respect to the choice of locally storing the whole plaintext resource itself in the first place.

2.4.3 A note on collusion

Collusion can happen when two users join effort to gain access to a resource that neither of them can access (we do not consider collusions with the server, which is assumed trustworthy to enforce the re-writing requested by the owner). In fact, if one of the users is authorized for the resource, she has no incentive and therefore there is no collusion. Also, the case of users working together to grant each other access to resources on which they individually have authorization cannot be considered collusion, since merging their knowledge they collectively do not go beyond their privileges. Collusion is then represented by users who join effort in maintaining portions of the resource (e.g., fragments or parts of mini-blocks as discussed above). For instance, each of the users could keep half of the fragments and they can merge their knowledge to patch for missing fragments. Such a situation does not add any complication with respect to the previous discussion, as it simply reduces to consider the group of colluding users as an individual attacker. We then note again that the collective effort, in terms of storage and/or computation, required to gain access would easily approximate the effort of locally storing the original plaintext resource itself. In other words, the attack strategy does not offer an advantages to users attempting to access the resources for which they are not authorized.

2.4.4 A note on erasure coding

Cloud storage providers have to offer reliability to their users. To reach this goal, the easiest approach is to replicate data [34]. Yet, this requires a not negligible storage overhead. Another less expensive approach is to leverage erasure coding [60]. With this solution, data are divided into fragments and stored replicated together with a certain number of parity fragments. Erasure codes offer the same fault tolerance property with lower replication.

Even if the cloud provider does not use erasure coding, an adversary could build a client-side erasure coding that permits to mitigate the loss of a fragment (i.e., the loss of the ability to decrypt a fragment that has been re-encrypted due to a policy update that revokes the user). This means that, given a resource divided into k fragments, the adversary can locally store t (with $t < k$) parity fragments to be able to recover the plaintext of the resource as long as no more than t different fragments have been re-encrypted. Yet, the resource owner has control over the number r of fragments that she is willing to re-encrypt, and this nullifies the efforts made by the adversaries that stored a number of parity fragments smaller than r.

When the policy changes, the owner can promptly re-encrypt a single fragment to revoke access for common adversaries without local erasure codes. After this, the owner can still re-encrypt other random fragments, later in time. An adversary that has used erasure coding would need to know that the re-encryption process is happening and to quickly download the resource that is being re-encrypted to make use of its erasure code. This is a very error-prone and lucky-based process for the adversary to complete. Moreover, it is important to note that the adversary had access to the whole resource previously, so if it was important and compact, she would have stored the resource in plaintext.

2.4.5 Comparison with other AONTs

We can now better illustrate the differences between *Mix&Slice* and other All-or-Nothing Transforms.

In general, other AONTs make use of two keys, the first is the encryption key, and the second is the AONT key. For example, the AONT proposed by

Rivest in [102] applies a transformation on the plaintext before encryption. This transformation is similar to encrypting in counter mode, except that the key is randomly chosen rather than fixed, and the last ciphertext block is the exclusive-or of the key and a hash of all the other ciphertext blocks. This effectively ensures that if a ciphertext block is missing, the AONT key is lost and the transformation is not invertible, thus realizing an AONT.

This schema works in the network transmission setting but can be easily circumvented in the access revocation scenario. An adversary could store the AONT key and thus be able to invert the transformation even if a fragment has been updated. The missing fragment implies missing parts in the plaintext, however, the majority of it would still be accessible. This scenario gets even worse in case the adversary has kept a local erasure code (as in Section 2.4.4). Let's assume that the adversary has kept an erasure code as big as $a\%$ of the resource size, and that the owner is re-encrypting $r\%$ due to a policy update.

In case $r < a$ (i.e., the policy update re-encrypts a portion of the resource whose size is smaller than the adversarial erasure code), then the revoked user can still access the file in its integrity in both Rivest's AONT and *Mix&Slice*. When $r \geq a$, instead, we have the following two scenarios.

Rivest's AONT The user can apply its erasure code, then use the AONT key to invert the function and re-gain access to a portion of the size as big as $1 - (r - a)$.

Mix&Slice Even after applying the erasure code, there is no key that can be used to invert the function, and the entire contents of the file are lost.

This property is depicted in Figure 2.11.

2.5 Implementation

In this section, we discuss the realization of our approach for its practical deployment. The components that have to be considered are the *client*, who decrypts resources to access them (Section 2.5.1), and the *protocol* used for the interaction between client and server. The protocol has a significant impact on

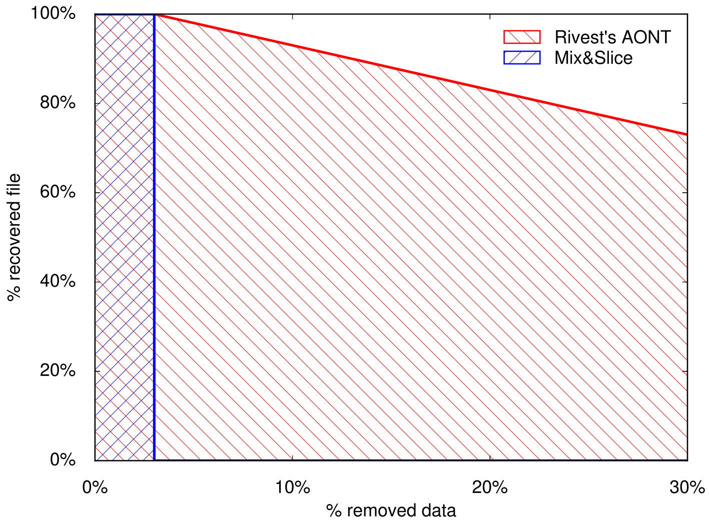


Figure 2.11: Comparison of percentage of recovered file between Rivest's AONT and *Mix&Slice* when the revoked user has kept an erasure code whose size is 3% of the resource size

the profile of the *server* responsible for hosting the resources and for authenticating the data owner who is the only party authorized to modify the data. In particular, we will consider two options for the realization of the interaction protocol: *i) Overlay* (Section 2.5.2), which operates on top of a common cloud object service (the server is unaware of the adoption of our approach and is a standard object server); and *ii) Ad-hoc* (Section 2.5.3), which directly supports the primitives to update a fragment and to get the current state of the resource (the server is aware of the features of our approach and attention will have to be paid to its internal structure).

We found from this analysis that the client is able to make use of our approach without restrictions, with a performance in the application of the technique for a common personal computer that does not represent a bottleneck when compared with any network bandwidth. For the protocol, when the technique is applied in a transparent way on top of existing object storage solutions (Overlay), we observe several orders of magnitude in performance improvement for some configurations. The realization of the technique using an ad-hoc protocol further improves the benefits with its greater flexibility, but it also requires to consider the mapping of the logical structure to its physical representation - for which we have identified an adequate solution. All these results prove the applicability of the technique in the current technological landscape and the benefits that it can provide for many application domains.

It is important to observe that the primary parameters influencing the performance are the size *msize* of mini-block and the number f of fragments. While the size of mini-blocks represents our security parameter and must be chosen by the data owner based on her security requirements, the number of fragments is chosen considering performance only. In the following, we will then focus on the tuning of the number of fragments, considering resources of variable sizes. (Note that the choice of the number of fragments implies also the definition of the number of macro-blocks, as the product of the number of macro-blocks by the number of fragments is equal to the number of mini-blocks of the resource.)

The evaluation of the best value for the number of fragments will have to consider a number of aspects that characterize the application domain. The major ones are: frequency of policy updates; frequency and average size of `get` requests; network bandwidth, for the upload and download direction. All these aspects have a direct impact on the overall throughput offered by our solution, which confirms its advantage in the prompt enforcement of revoke operations, measured by the average transfer rate for `get` requests.

The experimental results illustrated in this section have been obtained using, for the client, a machine with Linux Ubuntu 16.04 LTS, Intel i7-4770K, 3.50 GHz, 4 cores. For the server, we used an Amazon EC2 m4.large instance, with 4 CPUs and 8 GB of RAM. The client was connected to the Internet by a symmetric 100 Mbps connection.

2.5.1 Client

Our approach requires the client to execute a more complex decryption compared to the use of AES with a traditional encryption mode (e.g., CTR or CBC). The cost of decryption (which is comparable to the cost of encryption by the data owner) is nearly $\log_m(m \cdot b)$ times the cost of applying a single AES decryption, while the impact of reorganizing the data structure at each round is limited. Due to the high performance of modern processors

number of threads	*Mix&Slice* with AES-NI $(msize = 32)$	*Mix&Slice* with OAEP $(msize = 256)$	*Mix&Slice* with OAEP $(msize = 512)$	*Mix&Slice* with AES $(msize = 32)$
1	1.233 s	7.976 s	10.612 s	28.328 s
2	0.675 s	4.221 s	5.574 s	15.687 s
4	0.377 s	2.204 s	2.874 s	8.516 s
8	0.222 s	1.127 s	1.475 s	4.326 s
16	0.167 s	0.916 s	1.155 s	2.895 s

Figure 2.12: Performance comparison of mixing implementations

in the execution of block ciphers, this logarithmic cost factor is not critical. Also, decryption can be parallelized on multi-core CPUs, making the client processing even more efficient.

An aspect that has to be considered in the implementation of the client is the possible need to keep large amounts of data in memory. This may occur when fragments are downloaded one after the other and decryption can start only after the last fragment has been downloaded, which, for example, happens with the Overlay solution. If the resource size exceeds the available memory at the client, this leads to an extremely significant performance hit. The configuration of the system can (and should) avoid this possibility by splitting the resource into sub-resources (Section 2.5.2).

Experiments on the client

All code has been written in Python, because for all the functions the computational performance is not a constraint. The only component written in C was the invocation of the mixing for encryption and decryption functions. Since most current Intel x86 CPUs offer the support for a hardware implementation of AES, named AES-NI, we considered its adoption in our experiments.

We implemented both the mixing structure based on AES (described in Section 2.2.2) and the OAEP-based one (described in Section 2.2.3) to compare their performance profile. The results are presented in Figure 2.12. The OAEP-based implementation is faster than the AES one that does not leverage the hardware implementation. Even though the base application of AES and $SHA2_{512}$ share a similar performance profile, the OAEP approach ben-

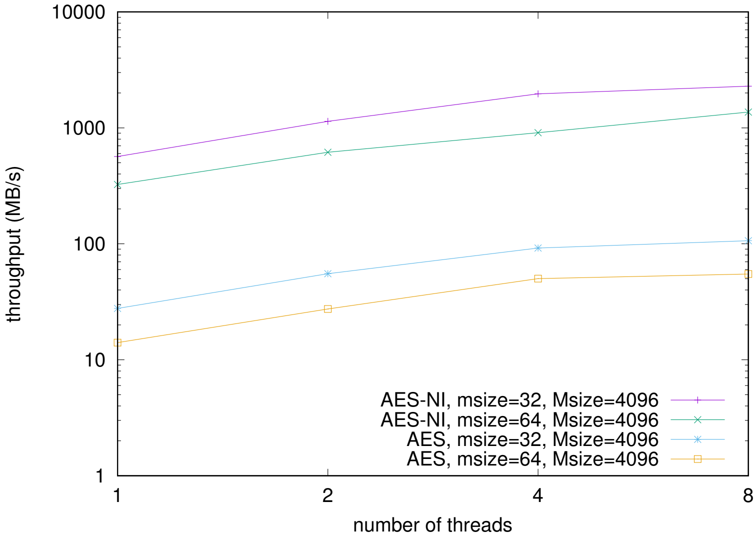


Figure 2.13: Throughput varying the number of threads

efits from the reduction of the number of rounds due to larger mini-blocks. On architectures that provide a native hardware implementation for AES, its use shows the best performance profile for *Mix&Slice* and should be preferred when compatible with the scenario.

Figure 2.13 shows that the cost of decryption is compatible with all reasonable scenarios for the application of our technique. In particular, the figure illustrates the throughput obtained, varying the number of threads, by the application of our approach in different configurations characterized by macro-blocks of size (*Msize*) 4 KiB, mini-blocks of size (*msize*) 32 and 64 bits (which imply 5 and 9 encryption rounds, resp.), when using AES-NI and when not using it (AES). Mixing was applied on data that were already available in memory. We notice that even the single-threaded 9-round non-hardware-supported implementation (line 'AES, *msize*=64, *Msize*=4096') offers a throughput that is greater than 100 Mbps. For the AES-NI multi-threaded 5-round implementation we reach a 2.5 GB/s throughput (line 'AES-NI, *msize*=32, *Msize*=4096'). The figure also shows that, increasing the number of threads, we reach a performance level that is 4 times the one obtained by the single-threaded implementation. This is consistent with the presence of 4 physical cores in the CPU we used, each with a dedicated AES-NI circuitry.

The performance, even for a large number of fragments, shows to be orders of magnitude better than the bandwidth of current network connections. Even without the hardware support (lines 'AES, *msize*=32, *Msize*=4096' and 'AES, *msize*= 64, *Msize*=4096'), the application of the cryptographic transformation shows greater throughput than the data transfer rate of most Internet connections. An experiment on 1 GiB size macro-blocks and 32 bit mini-blocks showed the expected slow down in throughput, managing the decryption in less than 5 seconds (still above the bandwidth of long-distance connections).

2.5.2 Overlay solution

The Overlay solution is analyzed using as a reference the Swift service. Swift has been selected due to its popularity, availability as open source, and technical features that are good representatives of what is offered by a modern object storage service for the cloud (resources are called *objects* in this discussion, to align with the Swift terminology). The Swift server instance has been installed on the Amazon EC2 platform. We consider two main alternatives for the realization of our approach on Swift[1] without any changes to the server.[2] The first option assumes to manage each fragment as a separate object. The second option makes use of the ability to access portions of objects and specifically considers the use of *Dynamic Large Objects* (DLOs). Our experiments show that this latter option provides significant benefits in performance with respect to managing fragments as separate objects. DLOs deserve then to be used when available.

Fragments as atomic separate objects. This approach is the most adaptable one, as it can be used with any object storage service. Also, the support for a policy update will be immediate, as it will be mapped to a single update to the object containing the corresponding fragment. However, these advantages come together with some potential restrictions. The client would be responsi-

[1] Swift organizes objects within *containers*. The current structure of Swift supports access control only at the level of containers. The analysis we present can be immediately adapted to the management of the access policy at the container granularity rather than the object granularity. We keep the analysis at the level of object for consistency with the chapter.

[2] We changed the server to support a large number of fragments in DLO mode.

ble for managing mixing and slicing. The approach requires the introduction of some metadata associated with each of the fragments or stored in a dedicated supporting object. The client has to be able to concurrently access all the fragments of the object to exhibit good performance when accessing large resources. If there are many fragments, this requires to create and keep open a large number of connections with the server.

Use of DLOs. The *Dynamic Large Objects*[3] (DLO) service of Swift has been introduced to support the management of large objects, going beyond the size limits of storage devices and providing finer granularity in the access. When using DLOs, an object is separated into a number of sub-objects that can be downloaded with a single request. The fragments of our approach can then be stored into separate DLO fragments. The Swift server is responsible for the management of the mapping from an object to its fragments, splitting a request for downloading an object into a number of independent requests to the server nodes that are responsible to store the data (the Swift architecture has a server node directly offering an interface to the clients and uses a number of independent storage nodes; this architecture provides redundancy and availability). In this way, the client only generates a single `get` request for the object, independently from the number of fragments. The descriptor of the object can be extended with the representation of the version of each fragment. A similar approach can be realized when the object service offers the flexibility to operate with `get` and `put` only on a portion of the object.

The major constraint of this approach is the need to wait for the download of all the fragments before the decryption of the first macro-block can start. As anticipated in Section 2.5.1, this causes delays and requires the client to keep available in RAM the complete encrypted representation of the object before it can be processed. To mitigate this problem, fragments can also be split into sub-fragments. In this way, the download will be organized with a serial download of all the sub-fragments representing the same set of macro-blocks. This is consistent with approaches used in cloud storage, where there is a common guideline to split resources larger than a few GiB (Swift forces a split at 5 GiB in its standard configuration). Experiments confirm that beyond

[3] `https://docs.openstack.org/swift/latest/overview_large_objects.html`

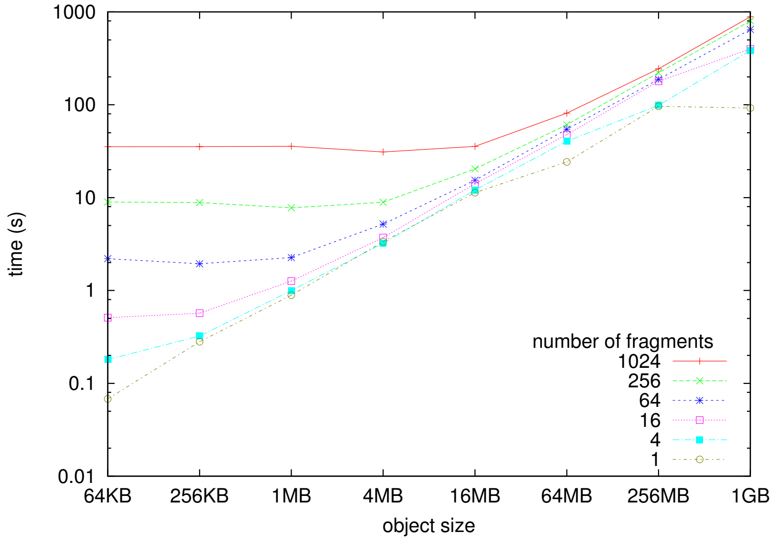


Figure 2.14: Time for the execution of get requests on Swift

1 GiB, the throughput remains stable even for configurations with a large number of fragments.

Experiments on the Overlay solution

We built a Swift client application in Python that implements the get and put_fragment methods that characterize our technique. We followed two implementation strategies, one using fragments as atomic separate objects, and the other adopting the DLO support offered by Swift.

Figure 2.14 compares, for different numbers of fragments, the time required for the execution of get requests assuming to map each fragment to a separate object. The lines correspond to distinct values for the number *f* of fragments (i.e., 1, 4, 16, 64, 256, and 1024). The parameters that drive the performance are the network bandwidth and the overhead imposed by the management of each request. For get requests, the overhead introduced by the management of one request for each fragment dominates when the resource is small, whereas the increase in object size makes the network bandwidth the bottleneck. The profile of put requests uploading the complete resource proved to be identical to the profile of get requests using a single fragment.

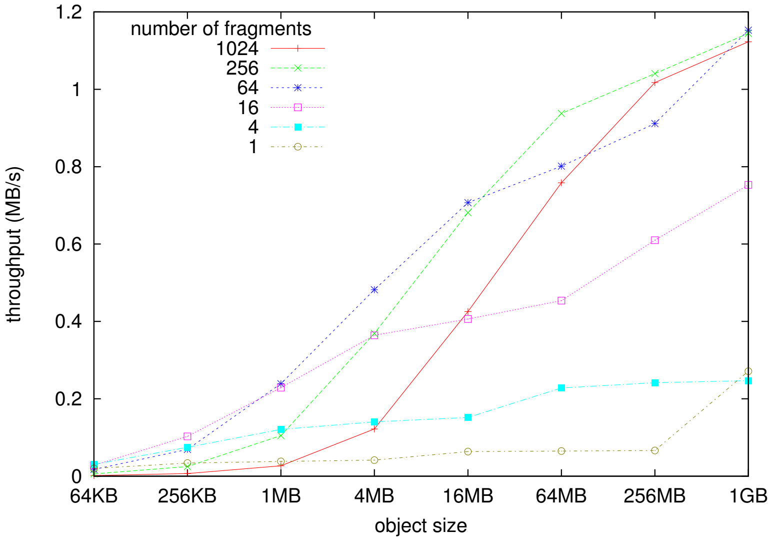


Figure 2.15: Throughput for a workload combining `get` and `put_fragment` requests on Swift

The execution of `put_fragment` requests grows linearly with the size of the fragment.

The identification of the best number of fragments requires to consider the profile of the scenario. We evaluated the behavior of a system on a collection of 1000 objects where, after each `put_fragment` request, a sequence of 50 `get` requests were executed on objects in the collection, all of the same size. Figure 2.15 reports the results of these experiments. As objects become larger, the benefits of fragmentation in the application of policy updates compensate for the overhead imposed on the retrieval of the objects. It is noted that the performance of the solution that does not use our technique corresponds to the line with one fragment. The throughput of the configurations using fragments is orders of magnitude higher already for medium-size objects. The graph also shows that the best number of fragments depends on the resource size. The identification of the value to use requires to consider the configuration of the system and the expected workload.

A second set of experiments followed the same approach, but considering the use of DLOs in Swift. The number of fragments still has a significant impact on the performance of the `get` request, because the server has to generate internally the mapping for the single request originating from the client and the multiple requests addressed to the storage nodes. The application of the same workload considered for the experiments in Figure 2.14, which interleaves `get` and `put_fragment` requests, produces the results presented in Figure 2.16. Comparing the cost with and without DLO we notice a significant benefit deriving from the use of DLOs.

2.5.3 Ad-hoc solution

The use of an ad-hoc protocol is able to provide the full range of benefits of our approach. The protocol will have to support the basic primitives to upload (`put`) and download (`get`) a resource. The `put` primitive, when used to upload the initial state of the resource, will have to provide a resource descriptor that defines: the identifier of the key k_0 used by the owner to encrypt the resource; the size of mini-blocks and the number of fragments (which determine the size of the macro-block); an array with an element for every fragment describing its version. In addition to the `put` primitive, the server will recognize the `put_fragment` primitive, which will allow the owner to update a fragment. Parameters of this primitive, in addition to the resource identifier and fragment content, will be the identifier of the fragment and its version number. The `put_fragment` primitive requires the authentication of the user issuing the request, in the same way as the `put` primitive.

The `get` primitive can return to the user the resource, one macro-block after the other. The client will be able to immediately start the decryption of macro-blocks, after a preliminary decryption with key k_i of the mini-blocks belonging to the fragments at version $i > 0$. In this way, the client does not have to wait for the completion of the download of all the fragments. The answer to the `get` request always provides first the resource descriptor, with the representation of the version of each of the fragments. Among the parameters of the `get` primitive we have the option to retrieve only a specific portion of the resource.

For this solution, we have to dedicate attention to the mapping of the logical structure to the physical representation of data. At the logical level, the resource is divided into fragments, and the content is represented by a sequence of macro-blocks. At the physical level, the resource can be stored as a collection of separate fragments or as a sequence of macro-blocks. In addition to these two options, there is a range of intermediate alternatives, with the interleaved representation of multiple fragments.

Experiments on the Ad-hoc solution

The advantage of a dedicated server is the ability to use an efficient protocol. The use of an ad-hoc server makes the management of fragments more flexible and avoids the overheads associated with the generation of a number of independent `get` requests equal to the number of fragments that are produced by the Overlay solution. Still, the use of a potentially large number of fragments can introduce non-negligible costs. In the extreme case where a large resource is managed with a single macro-block (i.e., the number of fragments corresponds to the number of mini-blocks of the whole resource), the client will have to wait for the download to complete to start decryption, and decryption will involve a high number of rounds. Also, when only a portion of the resource is needed, our approach requires the client to download the macro-blocks that contain the portion of interest; if macro-blocks are large, this may lead to a significant overhead. As already discussed, the identification of the optimal number of fragments has to consider several features of the application domain. In the current technological scenario, we notice that the use of an ad-hoc server can support a number of fragments larger than what is adequate for the Overlay solution, but extreme values cause inefficiencies.

As mentioned above, an important aspect that the implementation of the ad-hoc server has to consider is the mapping from the logical structure to its physical representation. In this analysis, we will consider a traditional scenario where the server uses the functions of the operating system to access the storage ability of mass memory devices. In the experiments we used the Amazon EC2 instance and its access to the Elastic Block Storage. The operating system offers an interface that allows to read and write physical blocks, typically a few KiB in size. The mapping of the bidimensional logical structure with macro-blocks and fragments to the concrete physical structure realized by a sequence of physical blocks can follow several strategies. To compare these alternatives, we assume a scenario where we have 1024 fragments and

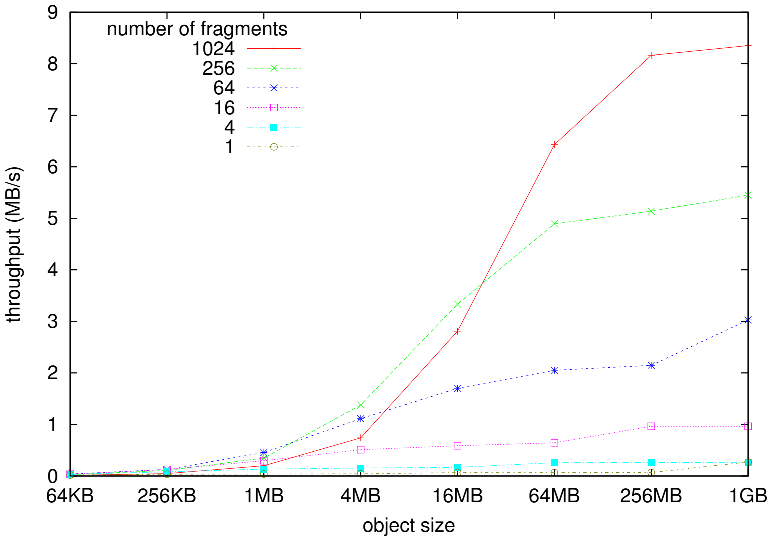


Figure 2.16: Throughput for a workload combining `get` and `put_fragment` requests with Swift DLOs

map the structure to 4KiB physical blocks. A first strategy consists in storing the resource one macro-block after the other. The dual strategy consists in storing the resource one fragment after the other. Between these two extremes, we have strategies that split each macro-block into a number of parts and store contiguously into a physical disk block all the macro-block portions that correspond to the mini-blocks in the same position. The rationale is that the organization along macro-blocks will be the most efficient to support `get` requests, but it will require to access all the physical disk blocks when a `put_fragment` request is received. The representation based on fragments will instead be the most efficient to support `put_fragment` requests, but it will introduce a significant overhead when managing `get` requests. For small resources these aspects do not have a large impact, whereas for large resources the performance benefit can be significant. Figure 2.17 illustrates the results obtained on a container with 1000 files, each of 1 GiB in size. The horizontal axis denotes the number of shares of each macro-block (1 represents the strategy with the macro-blocks stored in sequence, and 1024 represents the strategy with fragments stored in sequence). For a workload that interleaves a `get` request for every `put_fragment` request, the total cost is minimized when we use a solution with 256 fragments. Interestingly, the two extremes

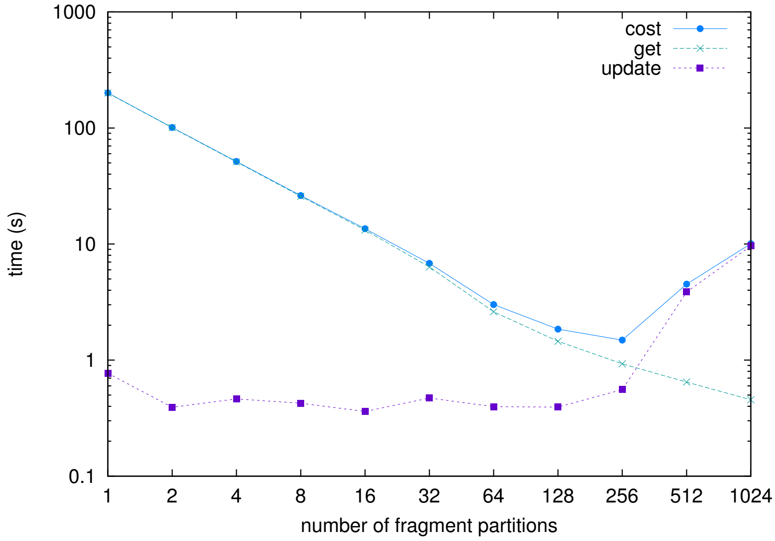


Figure 2.17: Configurations for physical blocks

with this workload do not represent the best option. In these experiments, we measured the time required to access the data from storage. In most systems we expect the network to be the bottleneck that limits the performance and the choice of physical representation will rarely be observed by the clients, but the performance benefit that is shown by the experiment can lead to a more efficient implementation of the server.

2.6 Related work

The idea of making the extraction of the information content of an encrypted resource dependent on the availability of the complete resource has been first explored by Rivest [102], who proposed the *all-or-nothing transform* (AONT). The AONT requires that the extraction of a resource where n bits of its transformed form are missing should require to attempt all the possible 2^n combinations. The AONT can be followed by encryption to produce an *all-or-nothing encryption schema*. In [102], the author proposes the *package transform*, which realizes an AONT by applying a CTR mode using a random key k. The ciphertext is then suffixed with the used key k XOR-ed with a hash of all the previous encrypted message blocks. In this way, a modification on the encrypted message limits the ability to derive the encryption key. This technique works under the assumption that the user who wants to decrypt the resource has never accessed the key before, but fails in a scenario where the user had previously accessed the key and now the access must be prevented (i.e., revocation of privileges on encrypted files). The user, in fact, could have stored key k and so she would be able (depending on the encryption mode used) to partially retrieve the plaintext. Key k can be seen as a digest: it is compact and its storage allows a receiver to access the majority of the file, even if one of the blocks was destroyed.

Most approaches for efficient secure deletion [33, 46] rely on the fact that the key is a digest for a resource and its content can be securely deleted by deleting the specific disk location that stores a piece of information that permits to derive the key used to encrypt the resource. Such approaches are already used by commercial storage devices [108] and recent proposals have considered the integration of such approaches with flexible policies [33]. All these approaches are not applicable in our scenario, where the encrypted resource is stored on a server that does not have access to (and hence does not store) the key and it is the user who has to decrypt the resource. Making the encryption key unavailable to the user does not limit her access.

In [66], the authors propose an algorithm for data protection based on information dispersal and fragmentation. The approach is similar to Mix&Slice (i.e., a first step for mixing and a second one for slicing), but a keyless algorithm based on a modification of Shamir's secret sharing [105] is used for the mixing. This has the disadvantage that each shard has the same size as the original data, thus incurring in significant storage overhead. The authors do not consider how to revoke access to previously authorized users.

Other approaches for enforcing access control in the cloud through encryption have been developed along two research lines: attribute-based encryption (ABE) and selective encryption approaches. ABE approaches (e.g., [54, 61, 96, 118]) provide access control enforcement by ensuring that the key used to protect a resource can be derived only by the users that satisfy a given condition on their attributes (e.g., age, role). The main shortcoming of these solutions is due to their evaluation costs (they rely on public key encryption), and to the hardness in the support of revocations [61, 118]. Approaches based on selective encryption (e.g., [43, 44, 57]) assume to encrypt each resource with a key that only authorized users know or can derive. In this scenario, policy updates are then either managed by the data owner, with considerable overhead, or delegated to the server through over-encryption [43, 44]. Although over-encryption guarantees a prompt enforcement of policy updates and demonstrates to offer good performance, it requires stronger trust assumptions on the server, which must provide dedicated support. On the contrary, our technique can be used also if the server is completely unaware of its adoption.

2.7 Conclusions

In this chapter, we presented an approach for efficiently enforcing access revocation on encrypted resources stored at external providers. Our solution enables data owners to effectively revoke access by simply overwriting a small portion of the (potentially large) resource and is resilient against attacks by users locally maintaining copies of previously-used keys. Our implementation and experimental evaluation confirm the efficiency and effectiveness of our proposal, which enjoys orders of magnitude of improvement in throughput with respect to resource re-writing, and confirms its compatibility with current cloud storage solutions, making it also immediately applicable to many application domains.

3

Securing Resources in Decentralized Cloud Storage

Decentralized Cloud Storage services represent a promising opportunity for a different cloud market, meeting the supply and demand for IT resources of an extensive community of users. The dynamic and independent nature of the resulting infrastructure introduces security concerns that can represent a slowing factor towards the realization of such an opportunity, otherwise clearly appealing and promising for the expected economic benefits. In this chapter, we present an approach enabling resource owners to effectively protect and securely delete their resources while relying on decentralized cloud services for their storage. In this chapter, we present a solution that combines an All-or-Nothing Transform (such as Mix&Slice) with two carefully designed allocation strategies to extend the guarantees obtained in the previous chapter to the decentralized cloud storage scenario. We address both availability and security guarantees, jointly considering them in our model and enabling resource owners to control their setting.

3.1 Introduction

A clear recent trend in information technology is the rent by many users and enterprises of the storage/computation services from other parties. With cloud technology, what was in the past managed autonomously now sees the involvement of servers, often in an unknown location, immediately reachable wherever an Internet connection is present. Today the use of these Internet services typically assumes the presence of a Cloud Service Provider (CSP) managing the service. There are a number of factors that explain the current status. In general, the procurement and management of IT resources exhibit significant scale economies, and large-scale CSPs can provide services at costs that are less than those incurred by smaller players. Still, many users have an excess of computational, storage, and network capacity in the systems they own and they would be interested in offering these resources to other users in exchange of a rent payment. In the classical behavior of markets, the existence of an infrastructure that supports the meeting of supply and demand for IT services would lead to a significant opportunity for the creation of economic value from the use of otherwise under-utilized resources.

This change of landscape is witnessed by the increasing attention of the research and development community toward the realization of *Decentralized Cloud Storage* (DCS) services, characterized by the availability of multiple nodes that can be used to store resources in a decentralized manner. In such services, individual resources are fragmented in *shards* allocated (with replication to provide availability guarantees) to different nodes. Access to a resource requires retrieving all its shards. The main characteristics of a DCS is the cooperative and dynamic structure formed by independent nodes (providing a multi-authority storage network) that can join the service and offer storage space, typically in exchange of some reward. This evolution has been facilitated by blockchain-based technologies providing an effective low-friction electronic payment system supporting the remuneration for the use of the service. On platforms such as *Storj* [115], *SAFE Network Vault* [63, 95], *IPFS* [23], and *Sia* [112], users can rent out their unused storage and bandwidth to offer a service to other users of the network, who pay for this service with a network crypto-currency [93].

However, if security concerns and perception of (or actual) *loss of control* have been an issue and slowing factor for centralized clouds, they are even more so for a decentralized cloud storage, where the dynamic and independent nature of the network may hint to a further decrease of control of the owners on where and how their resources are managed. Indeed, in centralized cloud systems, the CSP is generally assumed to be honest-but-curious and is then trusted to perform all the operations requested by authorized users (e.g., delete a file when requested by the owner) [56]. The CSP is discouraged to behave maliciously, since this would clearly impact its reputation. On the contrary, the nodes of a decentralized system may behave maliciously when their misbehavior can provide economic benefits without impacting reputation (e.g., sell the content of deleted files). Client-side encryption typically assumed in DCSs provides a first crucial layer of protection, but it leaves resources exposed to threats, especially in the long term. For instance, resources are still vulnerable in case the encryption key is exposed, or in case of malicious nodes not deleting their shards upon the owner's request to try reconstructing the resource in its entirety.

Protection of the encryption key is therefore not sufficient in DCS scenarios, as it remains exposed to the threats above. A general security principle is to rely on more than one layer of defense. In this chapter, we propose an additional and orthogonal layer of protection, which is able to mitigate these risks.

On the positive side, however, we note that the decentralized nature of DCS systems also increases the reliability of the service, as the involvement of a collection of independent parties reduces the risk that a single malfunction can limit the accessibility to the stored resources. In addition to this, the independent structure characterizing DCS systems - if coupled with effective resource protection and careful allocation to nodes in the network - makes them promising for actually strengthening security guarantees for owners relying on the decentralized network for storing their data.

In this chapter, we present a solution to enable resource owners to securely store their resources in DCS services, to share them with other users, while still being able to securely delete them. Our contribution is threefold. First, leveraging the protection guarantees offered by *All-Or-Nothing-Transform*

(AONT), we devise an approach to carefully control *resource slicing* and *allocation* to nodes in the network, with the goal of ensuring both *availability* (i.e., retrieval of all slices to reconstruct the resource) and *security* (i.e., protection against malicious parties jointly collecting all the slices composing a resource). The proposed solution also enables the resource owners to securely delete their resources when needed, even when some of the nodes in the DCS misbehave. Second, we investigate different strategies for slicing and distributing resources across the decentralized network, and analyze their characteristics in terms of availability and security guarantees. Third, we provide a modeling of the problem enabling owners to control the granularity of slicing and the diversification of allocation to ensure the aimed availability and security guarantees. We demonstrate the effectiveness of the proposed model by conducting several experiments on an implementation based on an available DCS system. Our solution provides an effective approach for protecting data in decentralized cloud storage and ensures both availability and protection responding to currently open problems of emerging DCS scenarios, including secure deletion. In fact, common secret sharing solutions (e.g., Shamir [105]), while considering apparently similar requirements are not applicable in scenarios where the whole resource content (and not simply the encryption key) needs protection, because of their storage and network costs (e.g., each share in Shamir's method has the same size as the whole data that has to be protected).

3.2 Basic concepts and scenario

The basic building block enabling the development of our solution is the application, at the client-side, of an *All-Or-Nothing-Transform* (AONT) encryption mode that transforms resources for their external storage. This mode requires the use of an encryption key. The encryption driven by the key represents the primary protection, and the use of AONT encryption mode further strengthens security. An AONT-encryption mode transforms a plaintext resource (original content in whatever form) into a ciphertext, with the property that the whole result of the transformation is required to obtain back the original plaintext. AONT guarantees in fact complete interdependence (*mixing*) among the bits of the encrypted resource in such a way that the unavailability of a portion of the encrypted resource prevents the reconstruction of any portion of the original plaintext. A party having access to a portion of the encrypted resource (but not to the encrypted resource in its entirety): *i)* if knowing the encryption key, it will not be able to reconstruct any portion of the resource (i.e., it will not be able to derive any information from the AONT-encrypted portions it has; the only option would be to attempt a brute force attack on the possible configurations of the missing portions, but their possible large size makes this attack unfeasible); *ii)* if not knowing the encryption key, it will not be able to perform brute-force attacks for guessing such a key, as any key (even the correct one) will be ineffective if not applied to the complete resource. AONT protection schemes can be built with the use of common cryptographic functions, like symmetric encryption and hash functions. An example of AONT scheme that guarantees complete mixing, which has also been used in the implementation of our prototype, is the Mix&Slicealgorithm presented in Chapter 2. As already discusseed, Mix&Slice works by applying different rounds of encryption, each operating on a carefully designed combination of the bits resulting from the previous round. Mix&Slice guarantees that each bit in the encrypted resource depends on the value of each bit in its plaintext representation. In our context, the use of AONT guarantees protection to the individual slices (and shards) composing the resource, and therefore to the resource itself (in its entirety as well as any of its portions). In fact, Mix&Slice makes each portion of the resource needed, in terms of information theory, to reconstruct any of the portions of the resource. The protection is then provided by the absence of information content.

Figure 3.1 illustrates our reference scenario. The focus of this chapter is the design of proper *slicing* of resources and the *allocation* of the produced slices to different nodes in the DCS system. Note that in the chapter we use the term *slicing* to refer to the cutting of a resource and the term *slices* to refer to the result of such a process. A slice is therefore a chunk of the resource and represents a unit of allocation, in contrast to a shard that represents a portion of the resource allocated to a node (a shard can include several slices). Our approach focuses on *slicing* and *allocation* and is agnostic with respect to the

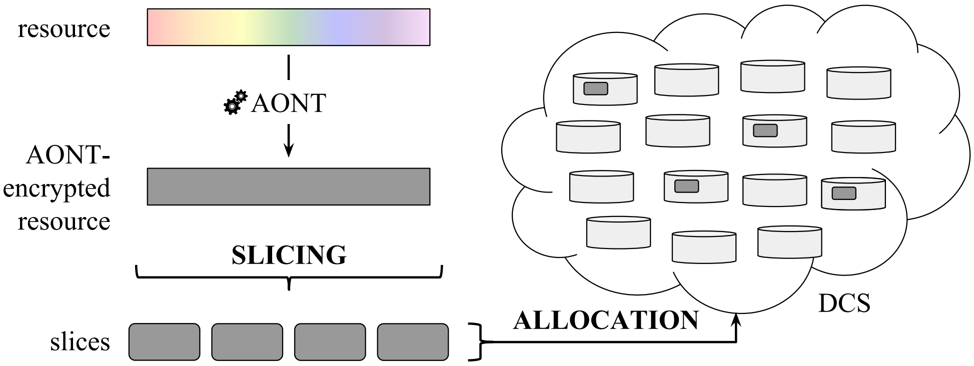


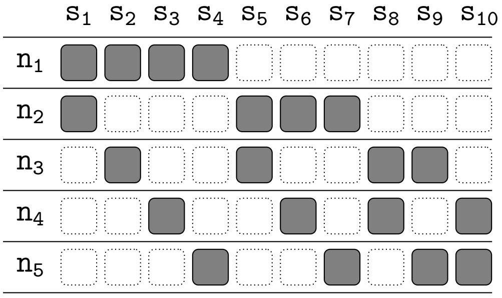
Figure 3.1: Reference scenario

specific AONT technique to be used, as long as the aimed strong protection guarantees are ensured, and with respect to the specific DCS adopted.

3.3 Allocation properties

In our approach, the slicing of the resources into several slices to be distributed at the different nodes is guided by the availability and protection properties that need to be guaranteed. Availability (despite nodes failure or temporary unreachability) is provided through replication, security is provided through protection against malicious coalitions. Malicious nodes (and coalitions thereof) are interested in making the resource unavailable, by not returning the slices of the resource they store, or in providing access to a resource even after its deletion, by not removing the slices of the resource they store and returning such slices to (not authorized) users who pay for it. Before addressing slicing, we then characterize the replication and coalition resistance properties of the distribution of a resource.

We assume a (transformed) resource that has undergone AONT encryption (as described in the previous section) at the client side. For simplicity, we will omit such an explicit remark on transformation and we will simply use the term resource to denote an AONT-encrypted resource. Also, we assume a resource to be composed of different slices, for distribution in a DCS. We will address the problem of producing such slices in Section 3.4.



	$\mathtt{s}_1$	$\mathtt{s}_2$	$\mathtt{s}_3$	$\mathtt{s}_4$	$\mathtt{s}_5$	$\mathtt{s}_6$	$\mathtt{s}_7$	$\mathtt{s}_8$	$\mathtt{s}_9$	$\mathtt{s}_{10}$
$\mathtt{n}_1$	■	■	■	■	□	□	□	□	□	□
$\mathtt{n}_2$	■	□	□	□	■	■	■	□	□	□
$\mathtt{n}_3$	□	■	□	□	■	□	□	■	■	□
$\mathtt{n}_4$	□	□	■	□	□	■	□	■	□	■
$\mathtt{n}_5$	□	□	□	■	□	□	■	□	■	■

Figure 3.2: An example of a minimal 3-protected and 2-replicated allocation function

We model a resource as a set $\mathcal{S} = \{\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{s}\}$ of slices to be allocated to the nodes, denoted $\mathcal{N}$, of the DCS. The following definition formalizes slice allocation.

Definition 1 (Allocation function) *Let $\mathcal{S}$ be a set of slices composing a resource and $\mathcal{N}$ be a set of nodes. An* allocation function $\varphi : \mathcal{S} \to 2^{\mathcal{N}} \setminus \emptyset$ *assigns each slice $\mathtt{s}_\mathtt{i} \in \mathcal{S}$ to a set of nodes $\varphi(\mathtt{s}_\mathtt{i}) = \mathtt{N}_\mathtt{i} \subseteq \mathcal{N}$, $\mathtt{N}_\mathtt{i} \neq \emptyset$.*

The allocation function dictates how slices are allocated to nodes in the DCS. The consideration of sets of nodes (in contrast to individual nodes) in the co-domain accommodates replication. The exclusion of the empty set of nodes ensures lossless distribution (i.e., each slice is allocated to at least one node). Figure 3.2 illustrates an example of an allocation function, considering a resource split into ten slices ($\mathcal{S} = \{\mathtt{s}_1, \ldots, \mathtt{s}_{10}\}$) allocated to five nodes ($\mathtt{n}_1, \ldots, \mathtt{n}_5$) in the DCS (nodes not used in the allocation are not reported in the figure). The figure has a row for each node and a column for each slice. The allocation of a slice to a node is represented by a gray box at the intersection between the row representing the node and the column representing the slice. Empty boxes with a dotted frame represent the fact that the slice is not allocated to the node. For example, $\varphi(\mathtt{s}_1) = \{\mathtt{n}_1, \mathtt{n}_2\}$.

We identify two main properties of an allocation, characterizing the availability, provided by *replication*, and the *protection* against possible malicious coalitions of nodes, provided by the diversification of the allocation.

We characterize availability provided by replication in terms of the number of replicas maintained in the system. While in principle the number of replicas maintained for each slice can differ, we assume the same number of replicas is used for all the slices. This derives from the fact that we assume that nodes are not associated with individual reliability profiles (Section 3.5). Since all slices are needed to reconstruct the resource, using fewer replicas for any of the slices would decrease the availability of the resource, which will be dictated by such a lower bound. The following definition formalizes the replication degree of an allocation function.

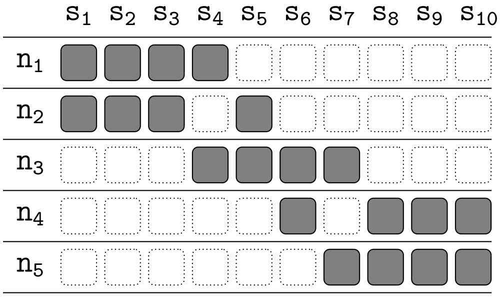
Definition 2 (r-Replicated allocation function) *Let $\mathcal{S}$ be a set of slices composing a resource, $\mathcal{N}$ be a set of nodes, and φ be an allocation function. Function φ is r-*replicated *iff $\forall \mathtt{s_i} \in \mathcal{S}$, $|\varphi(\mathtt{s_i})| \geq r$.*

For instance, the allocation function in Figure 3.2 is 2-replicated, as two copies are maintained for each slice.

We characterize the protection offered by an allocation in terms of the minimum number of nodes required to reconstruct a resource, as formalized by the following definition.

Definition 3 (k-Protected allocation function) *Let $\mathcal{S}$ be a set of slices composing a resource, $\mathcal{N}$ be a set of nodes, and φ be an allocation function. Function φ is k-*protected *iff for each $\mathtt{N_i} \subset \mathcal{N}$, with $|\mathtt{N_i}| \leq k$, $\exists \mathtt{s_j} \in \mathcal{S}$ s.t. $\varphi(\mathtt{s_j}) \cap \mathtt{N_i} = \emptyset$.*

A k-protected allocation function guarantees distribution of slices to nodes in such a way to dictate the cooperation of no less than $k + 1$ nodes to collect all the slices composing the resource (and hence enabling retrieving its plaintext). In other words, a k-protected allocation function guarantees protection of the resource against malicious (i.e., colluding) behavior of up to k nodes. In fact, with a k-protected allocation function, for each coalition of k nodes in $\mathcal{N}$, there is at least a slice that is not stored at any of the nodes in the coalition. Hence, such a coalition can neither decrypt the resource with a brute-force attack, nor prevent its deletion. The allocation function in Figure 3.2 is 3-protected: any subset of 3 out of the 5 nodes misses at least a slice. For instance, coalition $\{\mathtt{n_1},\mathtt{n_2},\mathtt{n_3}\}$ misses slice $\mathtt{s_{10}}$, while coalition $\{\mathtt{n_1},\mathtt{n_2},\mathtt{n_4}\}$



	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
n_1	■	■	■	■	□	□	□	□	□	□
n_2	■	■	■	□	■	□	□	□	□	□
n_3	□	□	□	■	■	■	■	□	□	□
n_4	□	□	□	□	□	■	□	■	■	■
n_5	□	□	□	□	□	□	■	■	■	■

Figure 3.3: An example of 2-replicated allocation function that is not 3-protected

misses slice $\mathtt{s}_9$. On the contrary, the allocation function in Figure 3.3, on the same slices and nodes, is not 3-protected (but only 2-protected): coalition $\{\mathtt{n}_1,\mathtt{n}_3,\mathtt{n}_4\}$ jointly possesses all the slices.

We refer to an allocation function that is r-replicated, according to Definition 2, and k-protected, according to Definition 3, as a (k, r)*-allocation*.

Definition 4 ($(\boldsymbol{k}, \boldsymbol{r})$-allocation) *Let $\mathcal{S}$ be a set of slices composing a resource, $\mathcal{N}$ be a set of nodes, and φ be an allocation function. Function φ is a (k, r)-*allocation *iff it is* k-protected *and* r-replicated.

According to Definitions 2 and 3, a (k, r)-allocation is also a (k', r')-allocation, for any $r' \leq r$ and any $k' \leq k$. In fact, trivially, an allocation function providing r replicas also provides $r' < r$ replicas. Analogously, an allocation function protecting a resource from coalitions of k nodes also protects the resource from coalitions of $k' < k$ nodes. Among all (k, r)-allocations, we are interested in identifying those for which k and r represent the highest values satisfying the availability and protection properties (i.e., satisfying the properties in a minimal way). We call such allocation functions minimal, as formalized by the following definition.

Definition 5 (Minimal $(\boldsymbol{k}, \boldsymbol{r})$-allocation) *Let $\mathcal{S}$ be a set of slices composing a resource, $\mathcal{N}$ be a set of nodes, and φ be a (k, r)-allocation. Function φ is* minimal *iff:*

1. it is not $(k + 1)$-protected ;

1. $\forall \mathtt{s}_\mathtt{i} \in \mathcal{S}$, $|\varphi(\mathtt{s}_\mathtt{i})| = r$.

According to Definition 5, a *minimal* (k, r)-allocation is an allocation that guarantees protection against coalitions of up to k (but no more) nodes and that uses exactly r replicas. The allocation function in Figure 3.2 is an example of minimal $(3, 2)$-allocation. In the following, we will restrict our attention to minimal allocation functions and, when talking about a (k, r)-allocation, we will implicitly assume such minimality.

3.4 Slicing and allocation strategies

In the absence of replication, producing an allocation that guarantees k-protection, that is, a $(k, 1)$-allocation, is straightforward: it is sufficient to split the resource into $k + 1$ slices and allocate each slice to a different node. When considering replication, different approaches can be taken for allocation, differing in the granularity of slicing and in how allocation diversifies the storage at different nodes. In the following, we discuss these options. In the discussion, in addition to parameters k and r introduced before, we will use parameters s, denoting the number of slices in which a resource is split, and n, denoting the number of nodes to be involved in the allocation of a resource. Different approaches vary in the number s of slices to be considered and in the number n of nodes to be involved for providing a (k, r)-allocation. We note that, with respect to nodes, the only parameter to be considered in the allocation strategies is the number n of nodes to be involved (the specific nodes to be involved can be selected randomly). We identify and study the behavior of two approaches for producing a (k, r)-allocation. The first approach aims to minimize the number of slices (*MinSlices*), while the second aims to minimize the number of nodes (*MinNodes*). We analyze these two approaches as they represent the two extremes with respect to granularity of slicing and diversification of allocation. Their analysis permits to highlight the characteristics of fine-grained (*MinNodes*) and coarse-grained (*MinSlices*) slicing, and can also represent a reference for intermediate configurations.

3.4.1 Minimizing the number of slices

We start noting that the number s of slices involved for guaranteeing a (k, r)-allocation must be such that $s \geq k + 1$. In fact, there should be at least $k + 1$ slices to guarantee k-protection, as formally captured by the following theorem.

Theorem 1 (Minimum number of slices) *Let k be a protection parameter and r be a replication factor. The number s of slices necessary to define a (k, r)-allocation is $s \geq k + 1$.*

A simple approach for determining a (k, r)-allocation extends the natural approach of producing $k + 1$ slices, by simply considering their replication at different nodes. Such an approach is characterized by a *coarse-slicing*, since minimizing the number of slices clearly entails a larger size for them, and by *consistent replication* (i.e., nodes have no intersection or complete intersection of stored slices).

We observe that a (k, r)-allocation function using the minimum number ($s = k + 1$) of slices implies that:

1. a node maintains at most one slice, that is, $|\varphi^{-1}(\mathtt{n_i})| = 1$, $\forall \mathtt{n_i} \in \mathcal{N}$ involved in the allocation;
2. the number of nodes involved in the allocation is exactly r times the number of slices, that is, $n = r \cdot (k + 1)$.

The first observation derives from the fact that, since there are only $k + 1$ slices, placing more than one slice on a node would imply the existence of a set of k nodes able to reconstruct the resource and therefore would not guarantee k-protection anymore. The second observation naturally derives from the first, considering that every slice needs to be replicated r times. The following theorem proves the observations above.

Theorem 2 *Let k be a protection parameter and r be a replication factor. A (k, r)-allocation $\varphi : \mathcal{S} \to 2^{\mathcal{N}} \setminus \emptyset$ that adopts the minimum number of slices*

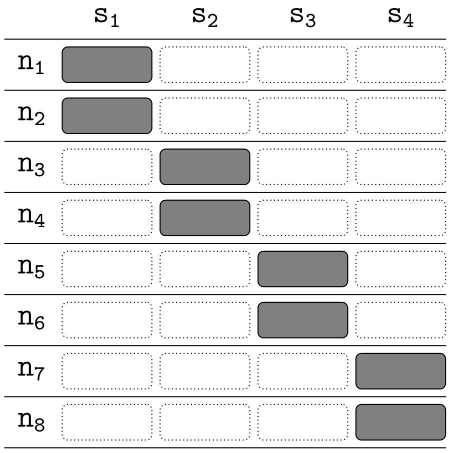


Figure 3.4: An examples of $(3, 2)$-allocation that minimizes the number of slices

$s = k + 1$ *is such that:*

1. $|\varphi^{-1}(\mathtt{n_i})| = 1$, $\forall \mathtt{n_i} \in \mathcal{N}$ *involved in the allocation* ;
2. *the number of nodes involved in the allocation is* $n = r \cdot (k + 1)$.

As an example, a $(3, 2)$-allocation using the minimum number of slices would imply splitting the resources into 4 $(= 3+1)$ slices, generating 2 copies of each slice, to be distributed at 8 different nodes. Figure 3.4 illustrates an example of allocation function enforcing this.

A (k, r)-allocation that uses the minimum number of slices $s = k + 1$ well resists to failures. Indeed, $k + 1$ nodes out of $r \cdot (k + 1)$ are sufficient to reconstruct the resource content, as long as one replica of each slice is available. However, the number of nodes used by such an allocation function quickly grows with k and r. For instance, a $(10, 5)$-allocation would need 55 $(= 5 \cdot (10 + 1))$ nodes.

3.4.2 Minimizing the number of nodes

At the other end of the spectrum of possible strategies for defining and distributing slices to guarantee a (k, r)-allocation, there are functions minimizing the number of nodes to be involved in the distribution (and deriving the number of slices in which the resource needs to be split based on this).

A trivial lower bound on the number of nodes that need to be involved in a (k, r)-allocation is $n \geq max(k + 1, r)$, since there should be at least r nodes to hold r replicas and at least $k + 1$ nodes to guarantee k-protection. The minimum number of nodes to be involved to guarantee (k, r)-allocation is actually higher than that as it needs to be at least the sum of the protection and replication parameters (k and r), as stated by the following theorem.

Theorem 3 (Minimum number of nodes) *Let k be a protection parameter and r be a replication factor. The number n of nodes necessary to define a (k, r)-allocation is $n \geq k + r$.*

The minimum number of nodes stated by Theorem 3 derives from two simple observations. First, to guarantee k-protection, for each coalition of k nodes, there must exist at least one slice that is not stored at any of the nodes in the coalition. Second, to provide r-replication, such a slice should be stored at (at least) r nodes that are not in the coalition. Hence, at least $k+r$ nodes need to be involved. As we will illustrate in the following, $k + r$ nodes, besides been necessary, are also sufficient to define a (k, r)-allocation.

While using the minimum number of slices applies a coarse slicing with consistent replication, using the minimum number of nodes applies a *fine-grained slicing* with *diversified replication* across nodes. Intuitively, instead of splitting the resource into slices and allocating to each node a single slice, minimizing the number of nodes requires slicing the resource into more fine-grained slices and allocating the slices to nodes in a diversified manner, to guarantee that no set of k nodes jointly possesses all the slices. The definition of the allocation requires then to identify the number of slices in which a resource needs to be split, which must be sufficient to distribute the r replicas to nodes while ensuring k-protection. The minimum number of slices needed for ensuring that no set of k nodes is able to reconstruct the resource when using $k + r$ nodes, clearly happens when any set of k nodes misses exactly one slice (which, given r-replication, would instead be stored at the r nodes not belonging to the set) and no two coalitions miss the same slice. In fact, if two sets of k nodes miss the same slice, such a slice could not have r replicas when using only $k + r$ nodes. The number of required slices can then be identified as the number of coalitions of k nodes out of $k+r$, that is $\binom{k+r}{k}$, as formally proved by the following theorem.

Theorem 4 *Let k be a protection parameter and r be a replication factor. Each (k,r)-allocation that adopts the minimum number of nodes $n = k+r$ uses $s = \binom{n}{k} = \binom{k+r}{k}$ slices.*

A (k,r)-allocation that uses $k+r$ nodes and $\binom{k+r}{k}$ slices has two interesting properties. The first one, already noted, is that any coalition of k nodes *misses exactly one* slice. The second one, deriving from the fact that the missing slice is different for different coalitions, is that *any* set of $k+1$ nodes is sufficient to reconstruct the resource (differently from the *MinSlices* approach where at least $k+1$ nodes are needed to reconstruct the resource but not any set of $k+1$ nodes guarantees that). The following theorem proves these two properties.

Theorem 5 *Let k be a protection parameter and r be a replication factor. Each (k,r)-allocation that adopts the minimum number of nodes $n = k+r$ and $s = \binom{k+r}{k}$ slices guarantees that:*

1. $\forall \mathtt{N_i} \subset \mathcal{N}$ *with* $|\mathtt{N_i}| = k$, $\nexists\, \mathtt{s_j}$ *s.t.* $\varphi(\mathtt{s_j}) \cap \mathtt{N_i} = \emptyset$;
2. $\forall \mathtt{N_i} \subseteq \mathcal{N}$ *with* $|\mathtt{N_i}| = k+1$, $\bigcup\limits_{\mathtt{n_j} \in \mathtt{N_i}} \varphi^{-1}(\mathtt{n_j}) = \mathcal{S}$.

A (k,r)-allocation that minimizes the number of nodes can be obtained by assuming $\mathcal{N}$ to comprise $k+r$ nodes and proceeding as follows. Let $2^{\mathcal{N}}_k = \{\mathtt{N_i} \in 2^{\mathcal{N}} : |\mathtt{N_i}| = k\}$ be all subsets of k nodes in $\mathcal{N}$. For each slice $\mathtt{s_i} \in \mathcal{S}, i = 1, \ldots, \binom{k+r}{k}$, $\varphi(\mathtt{s_i}) = \{\mathcal{N} \setminus \{\mathtt{N_i}\}$: with $\mathtt{N_i}{\in}2^{\mathcal{N}}_k\}$. Intuitively, for each slice $\mathtt{s_i}$, $\varphi(\mathtt{s_i})$ selects a coalition of k nodes that misses $\mathtt{s_i}$ and allocates slice $\mathtt{s_i}$ to all the other nodes. This guarantees that each coalition ($\mathtt{N_i}$) of k nodes misses at least one slice ($\mathtt{s_i}$), providing k-protection. Slice $\mathtt{s_i}$, which represents the missing slice for coalition $\mathtt{N_i}$, is stored at all the other $n-k = r$ nodes in $\mathcal{N}$, providing r-replication. Intuitively, in a (k,r)-allocation using the minimum number of nodes, no two slices are allocated exactly to the same set of nodes (i.e., $\forall \mathtt{s_i}, \mathtt{s_j} \in \mathcal{S}$, $\varphi(\mathtt{s_i}) \neq \varphi(\mathtt{s_j})$). In fact, the possible subsets of r nodes in $\mathcal{N}$ is $\binom{k+r}{r}$ and $\binom{k+r}{k} = \binom{k+r}{r}$.

For example, a $(3, 2)$-allocation using the minimum number of nodes requires $n = k + r = 3 + 2 = 5$ nodes and the use of $s = \binom{k+r}{k} = \binom{5}{3} = 10$ slices. Figure 3.2 illustrates an example of $(3, 2)$-allocation distributing 10 slices over 5 nodes. The allocation is a $(3, 2)$-allocation since it replicates each slice twice while guaranteeing that no coalition of 3 nodes possesses all the slices. More precisely, any coalition of 3 nodes misses exactly one slice and the missing slice is different for any of such coalitions. For instance, coalition $\{\mathtt{n_1}, \mathtt{n_2}, \mathtt{n_3}\}$ misses slice $\mathtt{s_{10}}$, while coalition $\{\mathtt{n_1}, \mathtt{n_2}, \mathtt{n_4}\}$ misses slice $\mathtt{s_9}$.

3.4.3 Discussion

We have discussed two alternative strategies for producing a (k, r)-allocation, aimed to minimize the number of slices (*MinSlices*, with coarse-grained slicing and consistent replication) and to minimize the number of involved nodes (*MinNodes*, with fine-grained slicing and diversified replication). When no replication is used (i.e., $r = 1$) these two strategies are equivalent, as each would imply the use of the same number of slices $s = k + 1$ and nodes $n = k + 1$. On the contrary, when replication is adopted (i.e., $r > 1$) the two strategies differ in the number of nodes n and slices s used and in the distribution of slices to nodes. Besides these two extreme configurations, the resource owner can decide to adopt other allocation strategies. The analysis presented in this section can then represent a reference for the definition and analysis of intermediate configurations.

We note that the structure of the *MinNodes* strategy has a correspondence with *secret sharing* [105]. In (m,d) secret sharing, the goal is to build d shares of a secret such that at least m of them are necessary to reconstruct a secret. Given a *MinNodes* (k, r)-allocation, with $k + r$ shares, the use of a ($k + 1$, $k + r$) secret sharing scheme would then satisfy the requirement that at least $k + 1$ nodes have to cooperate to access the resource, tolerating the loss of up to $r - 1$ nodes. Compared to the well-known Shamir's technique for secret sharing [105], the approach we propose shows a significant advantage with respect to storage and network capacity. In terms of computational cost, Shamir's technique requires to identify the roots of a polynomial, while our approach requires the application of symmetric encryption algorithms. The performance of symmetric encryption algorithms is so high, particularly for algorithms implemented in hardware by the CPU, that the potentially simpler computational structure of Shamir's technique does not provide an advantage and turns out to be slower when considering large resources. However, the computational cost is in any case a marginal element in this domain. In terms of storage, Shamir's approach offers security if each of the shares has the same size as the secret. With a $(k + 1,\ k + r)$ secret sharing scheme, there is therefore the need to store in the network $k + r$ times the amount of plaintext data, whereas our solution is characterized by the replication factor r. In terms of the minimum amount of data that has to be accessed by the owner, Shamir's solution asks the owner to read $k + 1$ times the size of the plaintext, whereas our technique, if the storage nodes support access to portions of the resource, does not require to access more than the size of the plaintext. We can then conclude that Shamir's technique, which is quite interesting for domains where the secret has a small size (e.g., encryption keys), is not convenient in the domain considered in this work. When Shamir's method is used to protect only the encryption key and then encryption is used to protect the resource, Shamir's method can be assumed to be only a key management strategy, making encryption of the resource the only protection measure, without offering the level of protection provided by AONT.

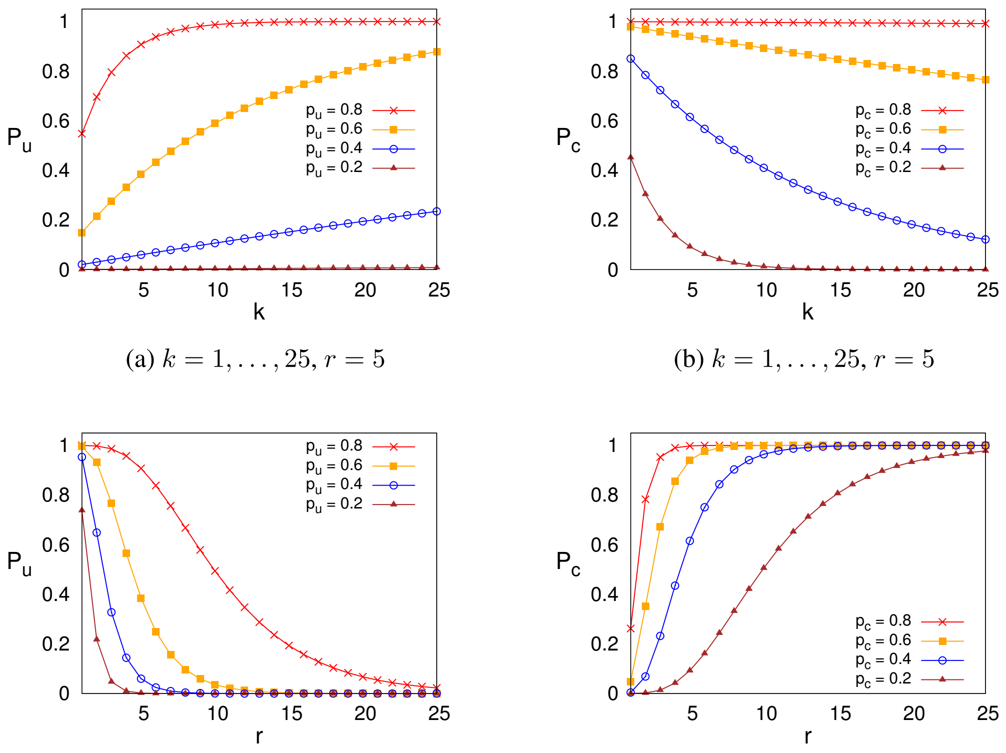
Note also that, for simplicity, we have assumed that the owner can arbitrarily split her resource as needed for the definition of a (k, r)-allocation. However, thanks to its flexibility, our approach can be adopted also when the encrypted resource is already organized in *chunks* that cannot be split for allocation (e.g., blocks resulting from the AONT algorithm adopted), or in general when slicing is constrained. Indeed, even if in the discussion, for simplicity, we consider slices of equal size, our approach can be adopted also if the size varies. Also, slices can contain non-contiguous chunks of the resource. Clearly, the number of chunks should be sufficient for the definition of a (k, r)-allocation (e.g., $k + 1$ and $\binom{n}{k}$ in our two alternative configurations). If the resource includes fewer chunks, it needs to be padded. If the resource includes more chunks than necessary, the resource owner can combine the chunks in s slices and apply the chosen allocation function over these slices. As an example, to define a (3,2)-allocation for a resource organized in 20 chunks using 5 nodes, chunks can be arbitrarily combined to identify 10 slices for allocation. Alternatively, k-protection and r-replication can be obtained by considering each chunk as a different slice and interpreting the allocation function as periodic in s, or simply by randomly allocating the chunks after the first s (which are the ones necessary to guarantee k-protection). For instance, a (3,2)-allocation for a resource with 20 chunks using 5 nodes can be obtained by applying the allocation function in Figure 3.2 twice (on slices $\mathtt{s}_1, \ldots, \mathtt{s}_{10}$ and $\mathtt{s}_{11}, \ldots, \mathtt{s}_{20}$), or by using it for slices $\mathtt{s}_1, \ldots, \mathtt{s}_{10}$ while arbitrarily allocating slices $\mathtt{s}_{11}, \ldots, \mathtt{s}_{20}$ at two nodes each.

3.5 Availability and protection guarantees

Parameters r and k introduced in the previous section characterize the degree of replication and of protection against malicious coalitions of nodes. Such parameters provide a clean and precise modeling and allow reasoning about properly setting the number of slices and the number of nodes to be involved in the allocation. The setting of k and r to provide given security and availability guarantees clearly depends on the specific characteristics of the network. For instance, in a stable network a low number of replicas may suffice to provide high availability, while in a highly dynamic and non-resilient network a higher number of replicas should be used to enjoy the same guarantee. In the same vein, actual protection against possible exposure of a resource to malicious coalitions depends on the nature of nodes involved in the allocation. Consistently with these observations, we note that a natural way for the resource owner to express and reason about availability and protection guarantees is the probability of the resource to become unavailable and the probability of a coalition of malicious nodes to jointly possess all the resource slices. In this section, we illustrate how to derive proper r and k settings to be then used for splitting resources into slices and for slices allocation, starting from the aimed guarantee of availability and security expressed in terms of such probabilities. Clearly, the probability of a resource to become unavailable, or exposed to malicious coalitions, depends on the probability of individual nodes to become unavailable or behaving maliciously. We then introduce the probability of a single node to fail, and hence to become unavailable, denoted p_u, and the probability of a node to behave maliciously, and hence to participate in a malicious coalition compromising protection, denoted p_c. We assume, as common in decentralized systems, the probability p_u of failure to be the same for all nodes and the failure of any node to be not influenced by the failure of the other nodes. This assumption enables a clean modeling, which can be taken as a reference for reasoning on different probability distributions. Since the selection of storage nodes is driven by a pseudorandom function, we also consider a uniform probability p_c of compromise and assume independence of compromise events on different nodes. We introduce the probability of a resource to become unavailable, denoted P_u, and of being exposed to a malicious coalition, denoted P_c, when using a (k, r)-allocation. The analysis will then guide the identification of the values for k and r to be used to guarantee that P_u and P_c do not exceed a given threshold. We discuss separately the *MinSlices* and *MinNodes* allocation strategies introduced in the previous section, which, as we will see, exhibit a different behavior with respect to availability and security guarantees.

3.5.1 *MinSlices* allocation

Using a (k, r)-allocation with the minimum number of slices, unavailability of a resource happens when, for any of the $k+1$ slices composing the resources, all the r nodes storing the replica of the slice fail. The probability of such an event to happen is $P_u = 1-(1-(p_u)^r)^{k+1}$, where $(1-(p_u)^r)$ is the probability that one of the r replicas of a slice is available and, for the assumption on the independence of the failure events, $(1-(p_u)^r)^{k+1}$ is the probability that one replica of each of the $k+1$ slices is available. In the same vein, the resource becomes exposed (and hence a compromise happens and deletion cannot be guaranteed) when a coalition of malicious nodes collectively possesses all the $k+1$ slices, that is, when the coalition contains $k+1$ nodes each possessing a different slice. The probability of such an event to happen is $P_c = (1 -$



(a) $k = 1, \ldots, 25$, $r = 5$

(b) $k = 1, \ldots, 25$, $r = 5$

(c) $k = 5$, $r = 1, \ldots, 25$

(d) $k = 5$, $r = 1, \ldots, 25$

Figure 3.5: Probability that the resource is unavailable (a,c) and that it is exposed (b,d) using a (k, r)-allocation that minimizes the number of slices, with r=5 varying k between 1 and 25 (a,b), and with k=5 varying r between 1 and 25 (c,d)

$(1 - p_c)^r)^{k+1}$, where $(1 - p_c)^r$ is the probability that one replica is stored on a node that is not part of a coalition and, consequently, $1 - (1 - p_c)^r$ is the probability that one replica is exposed. Since such an exposure must involve all the $k + 1$ slices, the probability that a coalition possesses all the slices is $(1 - (1 - p_c)^r)^{k+1}$. The following theorem proves such observations.

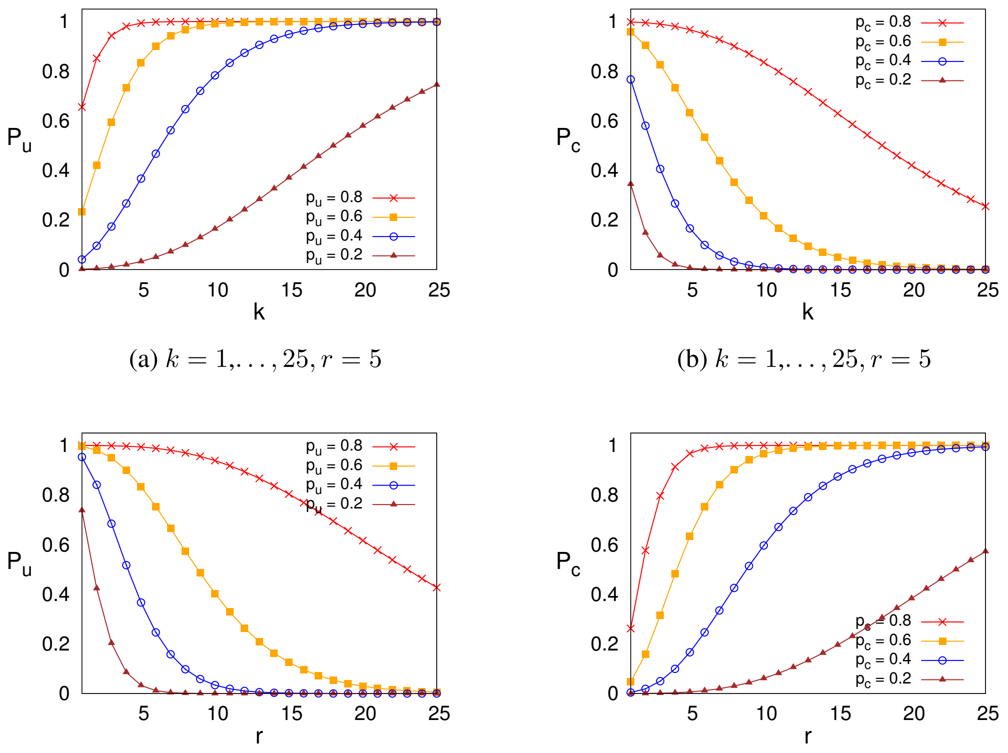
Theorem 6 *Given a set $\mathcal{S}$ of slices composing a resource, a set $\mathcal{N}$ of nodes with probability of failure p_u and probability of being compromised p_c, and a (k, r)-allocation using the minimum number of slices:*

$$P_u = 1 - (1 - ({p_u}^r))^{k+1}$$
$$P_c = (1 - (1 - p_c)^r)^{k+1}$$

Figure 3.5 illustrates how k and r affect the values of P_u (Figure 3.5(a,c)) and P_c (Figure 3.5(b,d)), considering different values of p_u and p_c, respectively. The values considered for p_u and p_c are 0.2, 0.4, 0.6, and 0.8. These values, extremely pessimistic with respect to what can be expected in real systems, have been chosen to study the behavior of the probabilistic formulas. Figure 3.5(a) reports the values of P_u assuming a fixed number $r = 5$ of replicas and varying k between 1 and 25. Figure 3.5(c) reports the values of P_u assuming a fixed $k = 5$ and varying the number r of replicas between 1 and 25. Figures 3.5(b,d) report the values of P_c in the same settings of Figures 3.5(a,c). As it can be seen from Figure 3.5(a), P_u increases as the value of k increases, because the number of nodes used in the allocation increases and therefore the probability of availability of a larger number of slices decreases. Indeed, the number of nodes necessary to reconstruct a resource grows with k (it is $k + 1$), and the probability of availability of all the nodes necessary to reconstruct the resource decreases. However, P_u remains low if the failure probability of a single node p_u is low. Probability P_u instead decreases as the value of r increases (Figure 3.5(c)), because each slice will be stored on a larger number of nodes, reducing the risk of unavailability. Figure 3.5(b) shows that P_c decreases as k increases because the number of nodes that should be part of a coalition increases, meaning that the probability of forming a coalition decreases. Probability P_c increases as r increases (Figure 3.5(d)), because the number of replicas of each slice increases and therefore also the probability that one replica is stored on a compromised node increases.

3.5.2 *MinNodes* allocation

Using a (k, r)-allocation with the minimum number of nodes, the unavailability of the resource occurs when any combination of r (or more) nodes becomes unavailable. In fact, regardless of the slices that those nodes store, such an event causes at least one slice to be unavailable. The probability P_u that a resource becomes unavailable is then $P_u = \sum\limits_{i=r}^{k+r} \binom{k+r}{i}(p_u)^i(1-p_u)^{k+r-i}$, where the binomial coefficient $\binom{k+r}{i}$ is the number of all possible combina-



(a) $k = 1,\ldots, 25, r = 5$

(b) $k = 1,\ldots, 25, r = 5$

(c) $k = 5, r = 1, \ldots, 25$

(d) $k = 5, r = 1, \ldots, 25$

Figure 3.6: Probability that the resource is unavailable (a,c) and that it is exposed (b,d) using a (k, r)-allocation that minimizes the number of nodes, with $r = 5$ varying k between 1 and 25 (a,b), and with $k = 5$ varying r between 1 and 25 (c,d)

tions of i nodes over $k + r$, with i varying in the range $r, \ldots, k + r$, that can be unavailable; $(p_u)^i$ is the probability that i nodes are unavailable; and $(1 - p_u)^{k+r-i}$ is the probability that the remaining nodes (i.e., $k + r - i$) are available. In the same vein, any coalition of $k + 1$ nodes causes an exposure of the resource, regardless of the slices they store. Relying on the minimum number of nodes, in fact, implies that any coalition of $k + 1$ nodes possesses all the slices (Theorem 5). The probability P_c of a compromise is then $P_c = \sum_{i=k+1}^{k+r} \binom{k+r}{i} (p_c)^i (1 - p_c)^{k+r-i}$, where the binomial coefficient $\binom{k+r}{i}$ is the number of all possible coalitions of i nodes over $k + r$ nodes, with i varying in the range $k + 1, \ldots, k + r$; $(p_c)^i$ is the probability that i nodes form a coalition; and $(1 - p_c)^{k+r-i}$ is the probability that the remaining nodes (i.e., $k + r - i$) are not compromised. The following theorem proves such observations.

Theorem 7 *Given a set $\mathcal{S}$ of slices composing a resource, a set $\mathcal{N}$ of nodes with probability of failure p_u and probability of being compromised p_c, and a (k, r)-allocation using the minimum number of nodes:*

$$P_u = \sum_{i=r}^{k+r} \binom{k+r}{i} (p_u)^i (1 - p_u)^{k+r-i}$$

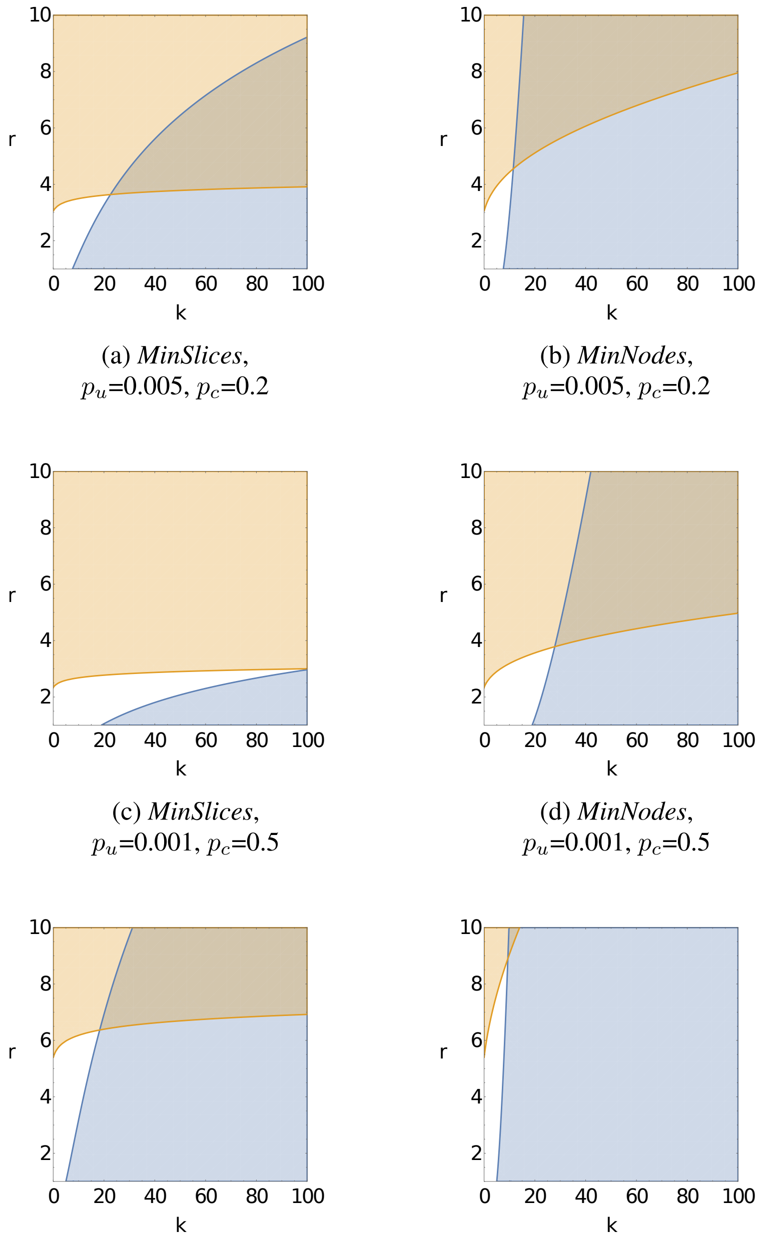
$$P_c = \sum_{i=k+1}^{k+r} \binom{k+r}{i} (p_c)^i (1 - p_c)^{k+r-i}$$

Figure 3.6 illustrates how k and r affect the values of P_u (Figures 3.6(a,c)) and P_c (Figures 3.6(b,d)), considering different values of p_u and p_c, respectively. The values considered for p_u and p_c are 0.2, 0.4, 0.6, and 0.8. Figure 3.6(a) reports the values of P_u assuming a fixed number $r = 5$ of replicas and varying k between 1 and 25. Figure 3.6(c) reports the values of P_u assuming a fixed $k = 5$ and varying the number r of replicas between 1 and 25. Figures 3.6(b,d) report the values of P_c in the same settings as Figures 3.6(a,c). From the figures, it is immediate to see that P_u and P_c present a similar behavior when adopting a configuration minimizing the number of slices and of nodes (i.e., P_u increases as k grows and decreases as r grows, while P_c decreases as k grows and increases as r grows).

3.5.3 Setting k and r

Our modeling of the probability that a resource is not available (P_u) and that it is exposed (P_c) can be used to set appropriate values for parameters k and r. To this purpose, fixing the maximum threshold P_u^{max} of resource unavailability and P_c^{max} of resource exposure, we compute all the configurations of k and r that guarantee $P_u \leq P_u^{max}$ and $P_c \leq P_c^{max}$ through the formulas in Theorems 6 and 7. Clearly, the values of k and r for the configurations satisfying the thresholds depend on the chosen allocation function.

Comparing the evolution of the probability P_u that a resource becomes unavailable using the *MinSlices* and *MinNodes* allocation strategies, varying



(a) *MinSlices*, p_u=0.005, p_c=0.2

(b) *MinNodes*, p_u=0.005, p_c=0.2

(c) *MinSlices*, p_u=0.001, p_c=0.5

(d) *MinNodes*, p_u=0.001, p_c=0.5

(e) *MinSlices*, p_u=0.05, p_c=0.1

(f) *MinNodes*, p_u=0.05, p_c=0.1

Figure 3.7: *MinSlices* and *MinNodes* (k, r)-allocations that guarantee $P_u \leq 10^{-7}$ and $P_c \leq 10^{-6}$ with different values for p_u and p_c

k (Figure 3.5(a) and Figure 3.6(a)), we can easily see that *MinSlices* is more robust against node failure (i.e., P_u increases slowly) than *MinNodes*. This is due to the fact that even if the number of nodes involved in the allocation increases in both configurations, with an allocation that minimizes the number of nodes the impact of a node failure on the availability of the resource is significant. A similar comment applies when comparing how P_u evolves in the two configurations varying the number r of replicas (Figure 3.5(c) and Figure 3.6(c)). In this case, the decrease of P_u with *MinSlices* is faster than the decrease of P_u with *MinNodes*. Therefore, we can conclude that, for configurations with the same values for r and k, *MinSlices* exhibits higher availability.

Comparing the evolution of the probability P_c that a resource is exposed due to a coalition of at least $k + 1$ nodes using *MinSlices* and *MinNodes* allocation strategies, varying k (Figure 3.5(b) and Figure 3.6(b)), we can easily see that *MinNodes* is more robust (i.e., P_c decreases faster) than *MinSlices*. This is due to the fact that, with an allocation that minimizes the number of nodes, the probability of forming a coalition of at least $k+1$ nodes among the $k + r$ nodes is smaller than the probability of controlling at least one of the r nodes for each of the $k + 1$ slices of the allocation that minimizes the slices. A similar comment applies when comparing how P_c evolves in the two configurations varying the number of replicas (Figure 3.5(d) and Figure 3.6(d)). The increase of probability P_c using *MinSlices* is faster than the increase of P_c using *MinNodes*, because it is more difficult to control at least $k + 1$ of the $k + r$ nodes than to control at least one node in each of the distinct $k + 1$ groups of r nodes. Therefore, we can conclude that, for configurations with the same values for r and k, *MinNodes* exhibits higher security.

When the resource owner has chosen the preferred allocation function, given the maximum threshold P_u^{max} of resource unavailability and P_c^{max} of resource exposure, different configurations of k and r guarantee that $P_u \leq P_u^{max}$ and $P_c \leq P_c^{max}$. Among all these configurations, the ones with low replication factor (r) require less storage and have lower economic costs, while the ones involving a limited number (n) of nodes enjoy simplicity in the management of the system and better performance of access operations (less connections have to be established). Figure 3.7 considers three different network configurations, characterized by a different probability p_u for single nodes to fail and a different probability p_c to behave maliciously, and illustrates the configurations of k and r satisfying the above thresholds using *MinSlices* and *MinNodes* allocation strategies. In the figure, the orange area on the top-left represents the configurations of k and r that satisfy the availability requirement (i.e., $P_u \leq 10^{-7}$), while the blue area on the bottom-right represents the configurations that satisfy the security requirement (i.e., $P_c \leq 10^{-6}$). We chose these thresholds because the overall availability guarantee ($P_u^{max} = 10^{-7}$) is the same declared in the specification of the system used in our experiments (i.e., Storj). We chose a higher value for P_c^{max} than P_u^{max} because protection against coalitions represents a security layer adding to the protection already offered by encryption. The intersection between the orange and blue areas represents configurations that provide both availability and security guarantees within the thresholds set by the owner. Among these configurations, the one located on the left/bottom corner of the intersecting area is the one to be preferred as the number of nodes and replicas is minimum.

Figures 3.7(a,b) consider nodes with $p_u = 0.005$ and $p_c = 0.2$. The optimal configuration for *MinSlices* it is $k = 26$ and $r = 4$ (i.e., $n = 108$), while for *MinNodes* allocation is $k = 12$ and $r = 5$ (i.e., $n = 17$). The second allocation, although more expensive on storage, due to one additional replica, considerably reduces the number of nodes involved in the storage of the resource compared to the adoption of the first allocation function. Our analysis demonstrates that this is a general behavior: *MinNodes* requires the same (or a slightly higher) number r of replicas and a significantly lower number n of nodes than *MinSlices*. This observation is confirmed by the extreme scenarios illustrated in Figures 3.7(c,d), considering highly reliable ($p_u = 0.001$) but lowly trusted ($p_c = 0.5$) nodes, and in Figures 3.7(e,f), considering unreliable ($p_u = 0.05$) but relatively trusted ($p_c = 0.1$) nodes. The optimal configurations in Figures 3.7(c,d) are $k = 100$ and $r = 3$ for *MinSlices* (i.e., $n = 303$), and $k = 27$ and $r = 4$ for *MinNodes* (i.e., $n = 31$, meaning that the number of nodes is ten times smaller). The optimal configurations in Figures 3.7(e,f) are $k = 10$ and $r = 9$ (i.e., $n = 99$) for *MinSlices*, and $k = 18$ and $r = 7$ (i.e., $n = 25$) for *MinNodes*.

Our analysis confirms that, for a wide range of values for P_u and P_c and assumptions on the node availability p_u and compromise risk p_c, our approach is able to identify a configuration of r and k with manageable complexity (i.e., a reasonable number of replicas and of nodes). We note that, even when r and k grow, the minimum number of slices composing a resource remains limited.

3.6 Implementation and experiments

To verify the benefit of our proposal we applied it into an existing DCS network. Among the existing DCS networks (e.g., *Storj* [115], *Sia* [112], *IPFS* [23], and *Maidsafe Safe-network* [71]), we selected Storj since, to the best of our knowledge, it is currently the most advanced and supported DCS. The market valuation of the cryptocurrencies [40] associated with these DCSs (Storj for Storj, Siacoin for Sia, Filecoin for IPFS, and Maidsafecoin for Maidsafe) supports the importance that these solutions are rising: at the date of submission, the global market capitalization of these initiatives is more than 400 million dollars. There are currently more than 100,000 nodes offering capacity in the Storj network, with more than 100PB of data available and a planned goal of 10 times growth in 2019.

Storj is a protocol that coordinates a decentralized network to create and enforce storage contracts between peers. Each peer can negotiate contracts with other peers, upload and download data from other peers, and periodically verify the availability and integrity of her data. Storj leverages a Distributed Hash Table (DHT) to connect parties interested in forming a storage contract. In the discussion, we maintain the terminology of our model and refer to parties outsourcing their resources to the decentralized network as owners (*renters* in Storj), and to parties offering storage space in exchange for a remuneration in a digital currency as storage nodes (*farmers* in Storj). *Bridge nodes* support the correct operations in the system and can take responsibility for the verification of the integrity and availability of resources. In the following, we describe the technical choices characterizing our implementation (Section 3.6.1), the experimental results (Section 3.6.2), and a few considerations about the impact produced by fine granularity retrieval (Section 3.6.3).

3.6.1 Implementation

The enforcement of *MinSlices* and *MinNodes* allocation strategies in Storj required changing the client library of the open source implementation. In particular, Storj currently offers three main clients, one written in C that must be built from source, one written in JavaScript and designed to be executed by a node.js runtime, and one written in Python and compatible with any Python environment. We integrated our technique within the Python implementation, also for easy integration with the implementation of Mix&Slice, which in addition of being an AONT-encryption supports other protection requirements (e.g., encryption-based access control and policy revocation). The design of Storj makes the client independent from the bridge and the storage nodes. Our work on the Python client allowed us to access the services of the whole network.

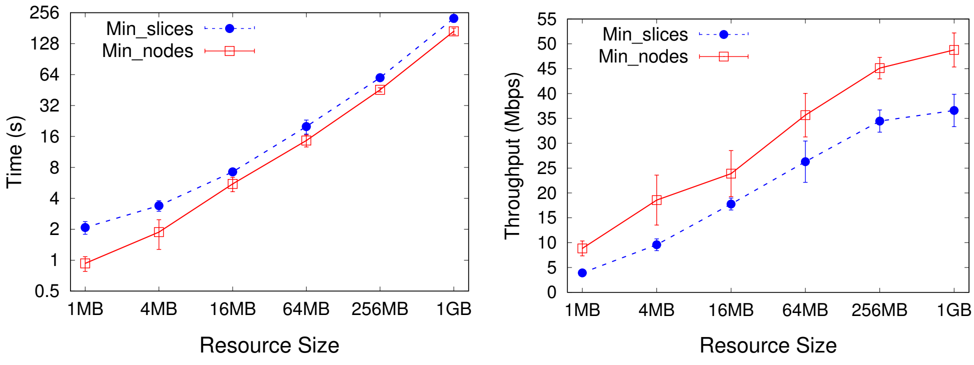
We implemented the *MinSlices* and *MinNodes* allocation strategies in the client and assigned slices (in the Storj terminology all the slices allocated to a node form a *shard*) to nodes. The performance of shard creation and resource reconstruction is orders of magnitude greater than the throughput of storage nodes in the Storj network (Mix&Slice operates at several hundred MB/s, whereas the maximum throughput we observed in Storj is around two orders of magnitude lower). In our experiments, we focused on evaluating the time required to complete the access request since the time requested by decryption does not have significant impact. We note that the use of the AONT forces each access request to be able to proceed with a client-side decryption only when the complete resource is available on the client. Should this be a problem for the specific application domain (e.g., resources are very large), mitigation can be provided by splitting the large resource and applying our approach to the resulting (smaller) chunks. Each chunk can then be downloaded and decrypted independently. This would reduce the access times to resources, but it may also delay the completion of the transfer, because the overhead for the management of a greater number of access requests reduces the effective bandwidth. The experiments confirm this observation (see Section 3.6.2), as they show that there is a performance benefit in managing large resources.

3.6.2 Experimental results

To evaluate the performance of the *MinSlices* and *MinNodes* allocation strategies, we introduced into the client a module that activates a number of parallel threads (in the considered configuration, we used 10 concurrent threads) to open access requests to the storage nodes. In Storj, access requests from the owner involve both storage nodes and bridge nodes. In fact, each time an owner needs to retrieve a shard, she makes a request to the bridge, which returns a token together with the IP address of the node storing the required shard (note that this access request is recorded and a crypto-currency payment is created by the owner for the node). The token is then used by the client as a parameter of an HTTP request directed to the node. Our experiments considered the performance of the system in the management of the dialogue between owner and node. In particular, we compared the access times observed for the two allocation strategies, varying the resource size.

An important restriction of the current implementation of Storj is that requests for shards are atomic and it is not possible to access only a specific portion of a shard managed by a node. This restriction cannot be removed operating only on the client, as it has a great impact not only on storage nodes but on the overall structure of the system. We then implemented the access requests for the *MinSlices* and *MinNodes* techniques as follows. For *MinNodes*, we implemented concurrent requests to the nodes. As soon as a node completes the delivery of its shard, a new request is started for another shard. The request is considered completed as soon as the client has received $k + 1$ complete shards. For *MinSlices*, for each shard a number t of parallel threads ($t \leq r$) are activated to manage a request to distinct nodes managing the same shard (which coincides with a slice for this allocation strategy), for a number of shards compatible with the number of concurrent threads (e.g., in the experiments we set $t = 2$ and we had 5 shards processed at the same time by the 10 threads). As soon as a shard is fully delivered to the client, the group of t threads is dedicated to another missing shard.

Figure 3.8 reports the results of our experiments, where we used resources of size varying from 1MB to 1GB. Figure 3.8(a) shows the time required for the completion of the access requests and Figure 3.8(b) shows the through-



(a)

(b)

Figure 3.8: Completion time (a) and overall throughput (b) in the *MinSlices* and *MinNodes* allocation strategies

put in terms of bandwidth. The graphs include two curves, one for *MinSlices* allocation, with $k = 26$ and $r = 4$, and one for *MinNodes* allocation, with $k = 12$ and $r = 5$, which correspond to the configurations considered in Figure 3.7(a) and Figure 3.7(b), respectively. The graphs present the average and the standard deviation of the values obtained with 10 executions. We note that the *MinNodes* strategy exhibits a moderate benefit compared to the *MinSlices* strategy. The benefit derives from the savings in overhead associated with the interaction with multiple nodes. An element that also contributes to the throughput is the natural variety of performance in nodes, with some being faster than others. The *MinNodes* strategy works well as long as the number of nodes with limited performance (*slow* nodes) is less than $r - 1$ and there are at least $k + 1$ nodes with good performance (*fast* nodes) serving the shard. For *MinSlices*, it is sufficient to have one of the $k + 1$ shards assigned to a group of nodes where the t nodes contacted in parallel happen to be slow to suffer from a significant delay in the access.

3.6.3 Further considerations

We noted that a limitation of the current implementation of the Storj node is that the request for a shard is atomic. The realization of a mechanism that permits to manage partial access requests would offer the opportunity for a significant improvement in the management of the access requests. For a generic server in a file sharing protocol this can be expected to be a relatively simple change in the server code; for a DCS system, this change would require a revision of the model used for the remuneration of access requests, as follows. The bridge should not consider each request received by the owner as an access to the complete resource (which also implies a payment to access the whole resource), but it should return to the owner the IP address and the authorization token of the node storing the shard. The owner and the node should then commence a protocol in which the owner issues signed confirmations in exchange of pieces of the resource. These confirmations can then be submitted to the bridge to receive the payment. Allowing the owner to pay a node only for the downloaded portion of the resource results in better performance and stronger competition among the nodes; best performing nodes would be preferred by the owner and would serve more traffic, receiving a correspondingly greater remuneration for their storage service.

The flexible structure of the *MinNodes* assignment would be particularly suited to this model. Under the assumption that nodes exhibit a high variability in access times, each slice could be retrieved by any of the r nodes storing it, with the possibility to adapt the amount of data transferred from each node depending on the response time and in case a group of r nodes happens to be all composed of slow nodes, the impact would be limited to the single or few slices that cannot be retrieved from fast nodes.

3.6.4 A note on DCS dynamicity

Decentralized cloud storage networks rely on voluntary effort of a considerable number of (possibly untrusted) nodes, which may dynamically join and leave the network at any time. In [17] we have studied how to combine the method presented in this chapter with Fountain Codes [80, 82, 106] to better exploit the dynamicity of decentralized cloud storage networks.

Fountain codes are a class of erasure codes preventing that the loss of one of the transmitted or stored blocks of a resource causes a data loss. Fountain codes, unlike other erasure codes (e.g., Reed-Solomon [100]), offer probabilistic reconstruction guarantees, meaning that with a probability $p < 1$, k of the n slices are sufficient for reconstructing the resource. The reconstruction probability p exponentially increases by retrieving additional shards. Although probabilistic, fountain codes have two main characteristics that allow us to profitably use them in the DCS context. First, they are *rateless*, that is, using these codes it is possible to create a new (i.e., different from each other) shard on the fly and therefore the number n of encoded shards is not fixed a priori. Second, each shard depends on a subset of (and not on all) the k original fragments of the resource and then only a subset of the original fragments are needed for generating a new shard.

The rateless characteristic of fountain codes allows the adaptive adjustment of the number of shards available for reconstructing a resource, thus impacting the availability and security guarantees. We use this characteristic in such a way that, when nodes go offline and the availability goes below a given threshold, we generate a new slice. Analogously, when the risk of confidentiality exposure is above a given threshold due to the presence of a high number of slices, we re-encrypt the resource and generate a new set of slices.

In DCS systems we can rely on the coordinator node to verify that the state is compliant with the thresholds. This can be achieved by offloading Proof-of-Retrievability [29] challenges and responses to the coordinator node. The challenges will then be issued to the storage nodes, and their responses compared with the expected ones. In case a response does not match, the coordinator will select another storage node and move the slice accordingly. The next chapter illustrates how the role of the coordinator can be removed by leveraging delegated challenge-response protocols.

3.7 Related work

RAID [94] is one of the main contributions aimed at the construction of reliable systems. RAID is normally deployed on local drives. With the advent of the cloud, RAID has been extended to take adversarial failures into consideration. Along this line of works, *HAIL (High-Availability and Integrity Layer)* [28] extended RAID with multiple cloud storage providers and a *Proof of Retrievability (PoR)* [29] scheme to verify that a provider still holds a certain piece of information. *HAIL* is however not well-suited for DCS systems, where the nodes are less reliable than well-established cloud service providers. Also, *HAIL* does not take into account the possibility of adversarial users trying to reconstruct the resources for their own personal profit. The works closest to ours are the solutions aimed to offer reliability and security of data in DCS. Many DCS networks that have recently been proposed, already include a certain degree of security guarantees. (i.e., protection against malicious parties jointly collecting all the slices composing a resource). Among them, Storj [115] and Sia [112] adopt client-side encryption and do not protect the outsourced data against coalitions of malicious nodes. SAFE Network [63] instead adopts a self-encryption technique: the resource is divided into shards and a weak AONT among 3 shards is applied before uploading them. In [95] the design of the SAFE Network and the possible attack vectors are analyzed. The solution proposed in [63, 95] is predetermined and the interaction between redundancy and security is not analyzed. Our proposal could be applied to improve the flexibility and security of these networks.

Another line of works is security of outsourced data (e.g., [2, 4, 91, 109]), which can be improved using AONT. Existing solutions however consider domains different from DCS. We have discussed before the proposal in [14], where the goal was to support policy evolution for outsourced resources where the access control policy is mapped to an encryption policy. Another approach using AONT and Reed-Solomon codes is AONT-RS [101]. Apart from the use of AONT, there is a limited similarity with the structure of our proposal. In fact, the work in [101] does not explicitly consider the structure of current DCS systems and does not provide an approach for the identification of the parameters to use in the configuration of the system. An evolution of the work on AONT-RS is CAONT-RS [76] that has been used by CDStore [75], which also uses two-stage deduplication to achieve both bandwidth and storage savings and robustness against side-channel attacks, while DepSky [24] addresses the privacy requirements using Shamir's scheme. All these proposals consider cloud-of-clouds environments, which see the integration of the services of cloud providers. Their adaptation to the DCS scenario requires significant attention and a model for the identification of the parameters to use in the configuration of the system. Also, the interaction between security and availability is not analyzed.

A precursor of DCS is represented by P2P systems. The P2P system closer to our proposal, which considers reliability and security, is Tangler [113]. The goal of Tangler is censorship resistance, which is a potential application of DCS, but not its main goal. Several of the assumptions at the basis of the design of Tangler have also been considered in the realization of DCS systems. A crucial difference between Tangler and our proposal is that Tangler uses Shamir's method, so it is quite expensive in terms of storage and bandwidth. Also, it does not aim at combining availability and confidentiality requirements in data allocation.

The novelty of our approach with respect to all above-mentioned techniques is the combination of AONT with different strategies for slicing and allocating resources in DCS systems and the joint consideration of security and availability guarantees. Our analysis of the characterization, interplay, and settings of the parameters guiding slicing and allocation can be used by all existing solutions to enhance their security and availability properties.

3.8 Conclusions

We presented an approach for providing effective secure protection to resources in decentralized cloud storage services. Our approach enables resource owners to protect their resources and to control their decentralized allocation to different nodes in the network. We investigated different strategies for splitting and distributing resources, analyzing their characteristics in terms of availability and security guarantees. We also provided a modeling of the problem enabling owners to control the granularity of slicing and diversification of allocation to ensure aimed availability and security guarantees. Enabling effective control for resource owners, our solution helps in removing natural reluctance due to security concerns and moves a step forward in the realization of novel services effectively benefiting from technological evolution. Our work leaves room for extensions, such as the consideration of error correcting codes and information dispersal algorithms to reduce the spatial overhead.

4

I *Told* You Tomorrow: Practical Time-Locked Secrets using Smart Contracts

Decentralized networks can rely on an honest party - the coordinator node - to verify that the allocation properties are respected. If the coordinator node notices that a node is offline or failing to prove the possession of a resource slice, it can reassign the slice to another node, so that the security and availability guarantees are still enforced.

When dealing with fully distributed networks, instead, it is pivotal to be able to detect misbehaving nodes autonomously. When a misbehaving node is detected, the network itself can reconstruct the slices that were previously assigned to it (for example by leveraging erasure coding techniques) and redistribute them to other nodes. This process has to work even when the data owner is offline and without imposing trust or honesty assumption on any of the involved parties.

To address this problem, this chapter introduces *I Told You Tomorrow (ITYT)*, a novel way of deploying Time-Locks based on smart contracts. A Time-Lock enables the release of a secret at a specific future point in time. Most current research proposals bind the recovery of the secret to the solution of cryptographic puzzles. These solutions, however, are impractical, as solving the puzzle requires a significant computational effort and provides no timing guarantees. Our Time-Lock technique can be used to implement delegated challenge-response protocols that, in turn, can be used to extend the data confidentiality and retrievability properties discussed in Chapter 3 to fully-distributed systems. ITYT relies on blockchain to measure the elapse of time, and it combines threshold cryptography with economic incentives and penalties, to effectively replace cryptographic puzzles.

We implemented a prototype of ITYT on top of the Ethereum blockchain. Our prototype leverages secure Multi-Party Computation to avoid any single point of trust. We also analyzed resiliency to attacks in the context of rational adversaries. The experiments demonstrate the low cost and resource consumption associated with our approach.

4.1 Introduction

Humans have always been trying to monitor the passing of time and to hand down information for posterity. The 1989 Oxford English Dictionary [45] defines a *time capsule* as: "*a container used to store for posterity a selection of objects thought to be representative of life at a particular time*".

After storing documents and memories inside the time capsule, they are generally buried underground, sometimes even sent out to space [64]. Another interesting application of time capsules is to unconditionally reveal secrets at a future date. Starting from the 90's, researchers have been designing techniques to create a digital counterpart of the physical time capsule [87] and they named these techniques *Time-Locks*.

A *Time-Lock* (TL) is a primitive that implements the following scheme: a data owner entrusts a notary with a secret to be disclosed at a future time; the notary keeps the secret private until then; and finally, the notary discloses it publicly. Once the notary has been entrusted with the secret, the owner is no longer needed for disclosure to happen. Such primitive can be used in many real-world scenarios, such as voting, inheritance management, and delegated challenge-response protocols.

Notaries [65] [99] are not suitable for all the scenarios [1], as they require the owner to completely entrust a single third party. Therefore, cryptographers have been designing encryption techniques, known as Time-Lock Encryption (TLE) [87], that enable the deployment of Time-Lock *Puzzles* [27,83]. In the TLE setting, the sending party (i.e., the owner) can encrypt a secret so that the receiving party is required to perform multiple decryptions to recover the secret. The number of times the decryption algorithm has to be run can be tuned by the sending party during the encryption process. As long as the time to run the decryption algorithm can be considered approximately constant, it is possible to protect a secret for arbitrarily long periods of time. In other terms, TLE can be used to create Time-Locks. Yet, it is worth noting that the disclosure time is not fixed, but it depends on when the decryption process starts.

The first TLE scheme was proposed by Rivest, Shamir, and Wagner [104]. The scheme relies on a trapdoor function so that the receiver has to undergo a computing effort that is orders of magnitude greater compared to the one of the sending party. Their approach also imposed the decryption algorithm to be executed in inherently sequential order. Such kinds of techniques are also known with the name *Proof of Sequential Work* (*PoSW*) [38,84].

An alternative to the use of trapdoor functions is the use of *weak hash-chains* [31], specifically, it requires the receiver to brute-force a chain of weak hashes to obtain the secret. The reason a chain is used in place of a single stronger hash is that it permits to reduce the variance associated with the time of the decryption process. However, in this setting the decryption process can be parallelized, thus the estimated disclosure time is far less reliable. These approaches are often associated with *Proof of Work (PoW)* [48] algorithms.

Yet, two aspects of TLE make them impractical. First, the owner has to make assumptions on future computing power. This is far from trivial. As an example, Rivest's LCS35 time-lock puzzle [103], whose decryption time was estimated to be 35 years, was cracked in 2019 in just 2 months using dedicated hardware [39]. Secondly, TLE schemes require the receiving party to run the decryption procedure continuously for a long time, which poses a question about incentives.

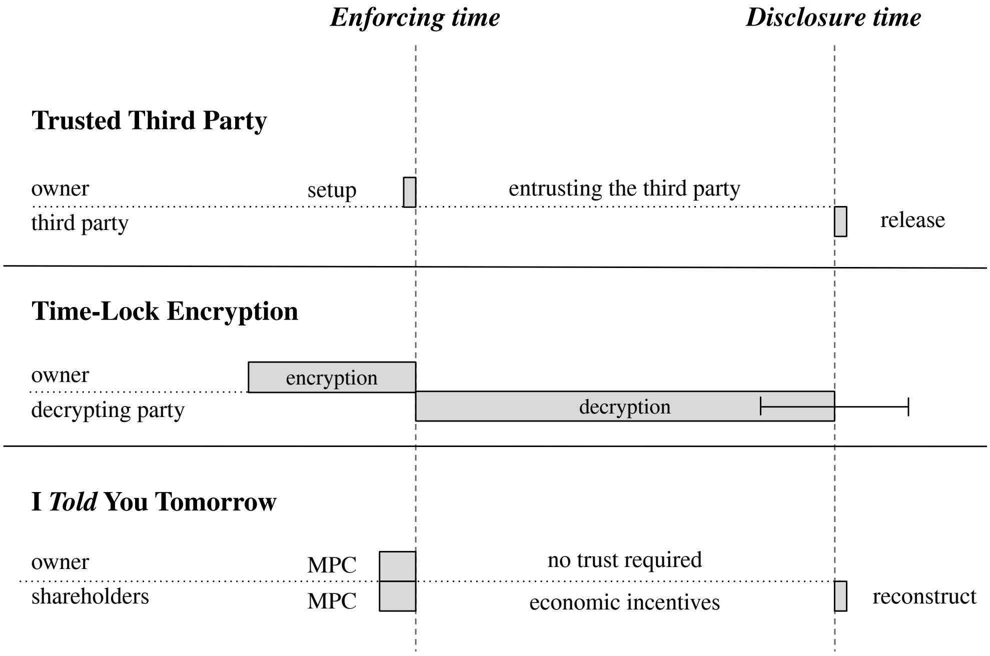


Figure 4.1: Activities comparison for subjects involved in different TL approaches

Our approach. In this chapter, we present a practical Time-Lock protocol named *I Told You Tomorrow* (*ITYT*), based neither on trust assumptions nor on cryptographic puzzles.

The basic idea of our approach is to first split a secret into shares using *threshold cryptography* (i.e., Shamir's Secret Sharing [105]), and assign them to users so that no one can recover the secret unless k*-of-*n shares are available. To ensure the TL behavior, we rely on economic incentives and penalties enforced by a smart contract. The contract rewards users for revealing their share only after the disclosure time and penalizes any other misbehavior. Rational users will always comply with the protocol, as we constraint that only correctly behaving is economically convenient, thus effectively deploying a distributed time-lock mechanism.

As rewards and penalties are associated with the correct management of the shares, it is important to confidentially generate and distribute them, even in the presence of a dishonest secret's owner. In our prototype, we address this by using *secure Multi-Party Computation (sMPC)*. We show a comparison between ITYT and other Time-Lock techniques in Figure 4.1.

- Trusted Third Party benefits from a short setup time but relies on a strong trust assumption.

- Time-Lock Encryption does not depend on trust assumptions, but it requires the receiving party to continuously execute the decryption routine from enforcing till disclosure time.

- *I Told You Tomorrow* allows the owner of a secret to practically disclose it at a future point in time without requiring her to take part in the disclosure protocol. It relies only on economic incentives.

The main contributions of this chapter can be summarized as follows.

1. We define a practical and abstract method to deploy Time-Locked secrets on the blockchain by leveraging an economic model in which every actor (or coalition of them) has an expected negative payoff associated with any possible misbehavior.

2. We address the problems that arise when combining secure Multi-Party Computation and protocols based on economic incentives and penalties.

3. We describe how to technically realize an implementation based on the Ethereum blockchain [116] and the FRESCO secure Multi-Party Computation framework [41], characterized by low overhead and limited gas cost of execution.

4.2 Background

Threshold Cryptography Threshold cryptography [105] enables the owner of a secret to share it among a group of participants. In a k-*of*-n threshold scheme, n shares of the secret are created and, usually, one is given to each participant. To reconstruct the secret, at least k different shares have to be combined. Hence, the advantages of threshold cryptography are (i) distribution of trust, and (ii) fault tolerance.

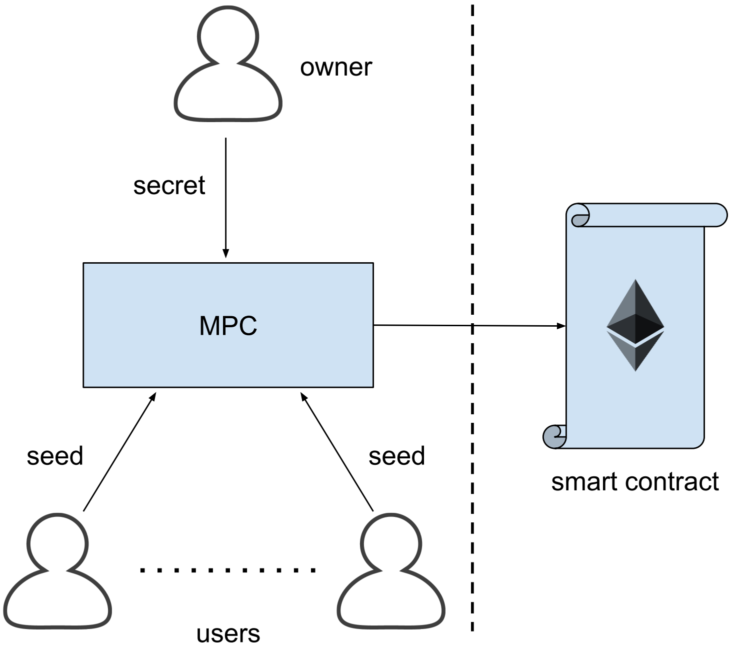


Figure 4.2: I Told You Tomorrow (ITYT) reference diagram

Secure Multi-Party Computation A *secure Multi-Party Computation (*sMPC) protocol [111, 117] is a cryptographic protocol that allows multiple parties to jointly compute a function over their inputs while keeping them private. Current sMPC frameworks are able to execute binary or modular arithmetic algorithms computed among several parties (even with dishonest majority) by leveraging *semi-homomorphic encryption* [42, 69] and *oblivious transfer* [68, 98].

Smart contracts A blockchain is an append-only list of blocks linked together via cryptographic properties. The blocks in the blockchain are non-mutable and are used to keep a permanent history of transactions. Smart contracts [107] are programming frameworks built on top of blockchains. A smart contract permits to program tamper-proof protocols whose outcome is verifiable by the whole network. We rely on smart contracts to pay incentives, trigger penalties, and in general to enforce the correct execution of our scheme without the need of a trusted party. Specifically, our approach makes use of the *time* and *hash* primitives, which are used to conditionally execute actions based on time and submitted data, respectively. We remark that the *time* primitive differs from TL in that it does not keep any data confidential.

Rational adversaries A malicious adversary [59] is someone who is willing to perform any action to attack a protocol. A rational adversary [55], instead, subverts the protocol only if it is economically convenient. Modeling the participants as rational enables the use of game theory concepts to analyze cryptographic protocols [6, 8, 32]. ITYT models all the participants as rational, and it does not rely on any trust or honesty assumption on them.

4.3 The ITYT protocol

ITYT is an instance of the TL abstraction: a mechanism that keeps a secret $\mathcal{S}$ private until its *disclosure time* t_d and publishes it afterward. ITYT implements TL by distributing the secret, provided by the *owner*, among several users (named *shareholders*, each obtaining a share h), so that a single user can not see $\mathcal{S}$ before the disclosure time t_d.

ITYT leverages economic incentives and penalties to ensure that (i) each user keeps its share secret until t_d, and (ii) users disclose the shares to reconstruct the secret after t_d. ITYT achieves practical Time-Locks as a consequence of the rational economic behavior of the involved parties (see Section 4.4).

4.3.1 Definitions

Principals

We denote by $\mathcal{U}$ the set of users that take part in an instance of ITYT. Additionally, we denote by $\mathcal{SC}$ the set of the smart contract identifiers. We then denote by $\mathcal{P}$ the set of principals consisting of $\mathcal{U} \cup \mathcal{SC}$.

Wallets

Each principal $p \in \mathcal{P}$ is associated with a wallet $wlt(p)$, accessible only by p, that can be used to receive or issue payments.

Protocol parameters

This table defines all the parameters that characterize an ITYT instance.

$\mathcal{S}$	secret
V	value of the secret
h_i	share of the secret issued to the i-th shareholder
n	number of shareholders
k	number of shares needed to reconstruct $\mathcal{S}$
t_d	disclosure time
t_{term}	termination time
$P_{\mathcal{O}}$	pawn deposited by the owner
$B_{\mathcal{H}}$	bid deposited by the shareholder
$R_{\mathcal{H}}$	reward paid to the shareholder in case of success
W_h	reward paid when whistleblowing a share
$W_{\mathcal{S}}$	reward paid when whistleblowing the secret

Smart Contract Data

The ITYT smart contract is a data structure with functions. Its initialization and functions will be described in Section 4.3.3 and 4.3.4 respectively. The data structure is composed by the following entries. Moreover, we will use the notation $sc.x$ to refer to the smart contract variable x.

$P_{\mathcal{O}}$, $B_{\mathcal{H}}$	economic pawn and bids
t_d, t_{term}, n, k	technical parameters
$R_{\mathcal{H}}$, $W_{\mathcal{S}}$, W_h	economic rewards
$[shares]$	array of deposited shares
$C_{\mathcal{S}}$	commitment of the secret
$[C_h]$	array of share commitments
$state$	state of the ITYT protocol
$[states]$	array of states for each share
$num\_pending$	number of pending shareholders
$num\_disclosed$	number of disclosed shares

Primitives

We now define the primitives used in the ITYT smart contract algorithms.

- `time()`: returns the current time as witnessed by the blockchain (generally defined in terms of block height).

- `hash(`d`)`: returns the result of the application of a chosen cryptographic hash function over the data d.

- `pay(`p_1, p_2, v`)`: transfers v from $wlt(p_1)$ to $wlt(p_2)$.

- `allow_withdraw(`$sc, p, v, predicate$`)`: allows principal p to withdraw the amount v from $wlt(sc)$, if $predicate$ holds. The predicate can also involve time conditions.

- `generate_shares(`$\mathcal{S}, [users]$`)`: generates the shares $h_1, \dots, h_n$ using k-*of*-n secret sharing [105], and it securely distributes them to the parties. The primitive guarantees that the (a) i-th shareholder is the only principal who learns the share h_i, and (b) the owner learns only the commitment $hash(h_i)$, for all the shares. We discuss in Section 4.5 how our prototype implements this primitive by leveraging sMPC.

- `initialize_sc(`$[params]$`)`: instantiates an ITYT smart contract, with parameters $[params]$, and deploys it to the blockchain. The primitive is executed by the owner and returns the smart contract identifier $sc \in \mathcal{SC}$.

4.3.2 Roles

In ITYT, each user $u \in \mathcal{U}$ plays one of the following roles: *owner*, *shareholder* and *whistleblower*.

- **Owner:** An owner $\mathcal{O}$ is someone willing to delegate the disclosure of a secret $\mathcal{S}$ to a Time-Lock, so that the owner is *not* involved in the disclosure of $\mathcal{S}$ at disclosure time t_d. The owner $\mathcal{O}$ sets up the TL as well as the corresponding parameters. In particular, $\mathcal{O}$ sets (i) n the total number of shares h of $\mathcal{S}$, (ii) k, the number of shares needed to recover $\mathcal{S}$, (iii) the disclosure time t_d, and (iv) all the bids and the rewards that define the instance.

- **Shareholder:** A shareholder $\mathcal{H}$ is entrusted by the owner $\mathcal{O}$ to keep a share h of the secret $\mathcal{S}$ confidential until t_d, and to publicly disclose it afterward. In exchange for her service, $\mathcal{H}$ will receive a reward paid by the

Algorithm 1 Protocol initialization (executed by the owner)

```
input
    $[params]$                          economic values of the ITYT instance
    $\mathcal{S}$                       secret
    $\mathcal{O}$                       user identifier of the owner
    $[\mathcal{H}_1, \dots, \mathcal{H}_n]$   list of shareholder identifiers
procedure INIT($[params], \mathcal{S}, \mathcal{O}, [\mathcal{H}_1, \dots, \mathcal{H}_n]$)
    $sc \leftarrow$ initialize_sc ($[params]$)                      ▷ Create $sc$
    pay ($\mathcal{O}, sc, sc.P_{\mathcal{O}}$)                ▷ Transfer owner's pawn to $sc$
    $C_h \leftarrow$ generate_shares ($\mathcal{S}, [\mathcal{O}, \mathcal{H}_1, \dots, \mathcal{H}_n]$)
    $sc.C_{\mathcal{S}} \leftarrow$ hash ($\mathcal{S}$)                     ▷ Set secret commitment
    $sc.state \leftarrow$ PENDING
    for $i \leftarrow 1, n$ do                               ▷ Set share commitments
        $sc.C_h[i] \leftarrow C_h[i]$
        $sc.states[i] \leftarrow$ PENDING
    end for
    $sc.num\_pending \leftarrow n$
    $sc.num\_disclosed \leftarrow 0$
    return $sc$                                         ▷ The smart contract id
end procedure
```

smart contract whenever the following two conditions hold: (i) her share h is disclosed only after t_d, and (ii) the secret $\mathcal{S}$ is not revealed before t_d.

- **Whistleblower:** A whistleblower $\mathcal{W}$ reports misbehavior of other parties in return for economic incentives. Whenever $\mathcal{W}$ possesses a share h or the secret $\mathcal{S}$ before t_d, $\mathcal{W}$ can submit it and receive a reward.

4.3.3 Smart contract setup

We assume that the owner already selected the shareholders that will take part to the TL instance[1]. To start the setup, the owner instantiates a smart contract, sc using `initialize_sc` ($[params]$), where $[params]$ are the desired economic parameters. Then, by collaborating with the shareholders, she executes the `generate_shares` primitive and updates sc with the commitments of

[1] How to randomly choose competing players in adversarial settings such as blockchains has already been addressed by other publications [47, 90].

Algorithm 2 Shareholder commitment to participate in ITYT

```
input
    $sc$      smart contract identifier
    $i$       index of the current shareholder
    $\mathcal{H}_i$     user identifier of the $i$-th shareholder
▷ precondition: $\mathcal{H}_i$ checks that hash($h_i$) $= sc.C_h[i]$
procedure PARTICIPATE($sc, i, \mathcal{H}_i$)
    if $sc.state$ = PENDING then
        if $sc.states[i]$ = PENDING then
            pay($\mathcal{H}_i, sc, sc.B_\mathcal{H}$)
            $sc.states[i] \leftarrow$ PAID
            $sc.num\_pending - = 1$
            if $sc.num\_pending = 0$ then
                $sc.state \leftarrow$ LOCKED
            end if
        end if
    end if
end procedure
```

the secret and of the shares. She also transfers the amount $P_\mathcal{O}$ to sc, used to pay the rewards. The steps are illustrated in Algorithm 1.

A shareholder $\mathcal{H}$ is incentivized in taking part in the TL by the possibility of earning a reward $R_\mathcal{H}$. Once each shareholder gets the share, she verifies that the corresponding commitment written into sc matches. If so, she agrees to deposit her bid, $B_\mathcal{H}$ ($< R_\mathcal{H}$). This process is shown in Algorithm 2. As soon as all the shareholders have executed this algorithm, the state of the instance is set to LOCKED, and the TL is active. If any party does not commit before a defined deadline, the deposited funds are returned to their proprietaries and the instance setup aborts. This will be discussed with more details in Section 4.5.

The algorithms do not check for the economic parameters to be well-formed (see Section 4.4). As all the parameters are public and immutable, a malformed game will not be played by any participant, in the same way as a lottery whose top prize is worth less than the ticket price will not sell tickets.

Algorithm 3 SC function to whistleblow a share before t_d

```
input
    $sc$      smart contract identifier
    $i$       index of the whistleblown share
    $h_i$     the $i$-th share to be whistleblown

procedure WhistleblowShare($sc, i, h_i$)
    if $sc.state$ = LOCKED and time() $< sc.t_d$ then
        if $sc.states[i]$ = PAID then
            if hash($h_i$) $= sc.C_h[i]$ then
                $sc.shares\ [i] \leftarrow h_i$
                $sc.states[i] \leftarrow$ WHISTLEBLOWED
                $sc.num\_disclosed + = 1$
                if $sc.num\_disclosed = sc.k$ then
                    $sc.state \leftarrow$ FAILED
                end if
                pay($sc, caller, sc.W_h$)
            end if
        end if
    end if
end procedure
```

4.3.4 Smart contract functions

Before presenting the functions (actions), it is important to note that the intrinsic transaction ordering of the blockchain guarantees that each function invocation is executed atomically, thus locking mechanisms are not required.

`WhistleblowShare`: This function enables the whistleblower to report the misbehavior of a single shareholder. $\mathcal{W}$ sends the share h_i to the contract. If the commitment matches, the share whistleblow bonus W_h ($<B_{\mathcal{H}}$), is paid to the whistleblower. In that event, the shareholder $\mathcal{H}_i$, would lose the chance of earning her reward, $R_{\mathcal{H}}$. Another consequence, is that if the number of whistleblown shares would equal k, then the TL would be marked as failed (i.e., no further actions allowed). The function is presented is Algorithm 3.

`WhistleblowSecret`: It enables the whistleblower $\mathcal{W}$, to prove the possession of the secret ahead of the disclosure time t_d. In detail, $\mathcal{W}$ sends a secret, $\mathcal{S}'$ to the contract. If the commitment matches, then the TL is marked

Algorithm 4 SC function to whistleblow the secret before t_d

```
input
    $sc$    smart contract identifier
    $\mathcal{S}$    the secret to be whistleblown
procedure WhistleblowSecret($sc, \mathcal{S}$)
    if $sc.state$ = LOCKED and time() $< sc.t_d$ then
        if hash($\mathcal{S}$) $= sc.C_{\mathcal{S}}$ then
            $sc.state \leftarrow$ FAILED
            pay($sc, caller, sc.W_{\mathcal{S}}$)
        end if
    end if
end procedure
```

as failed and the secret whistleblow bonus $W_{\mathcal{S}}$ ($> R_{\mathcal{H}}$) is paid to the whistleblower. In that event, all the shareholders would be subject to an economic penalty that results in the loss of the bid already paid, and the remaining smart contract funds are destroyed. The function is illustrated is Algorithm 4. The need for both `WhistleblowShare` and `WhistleblowSecret` will be discussed in the following sections.

`Disclose`: After the disclosure time t_d has passed, each shareholder $\mathcal{H}_i$ submits her share h_i to the contract. The submission is successful if (a) the TL is not marked as `FAILED`, (b) the share is marked as `PAID`, and (c) `hash`(h_i) matches the value stored into the sc, otherwise it fails. Conditionally to the outcome of the TL instance, and immediately after the termination time t_{term}, the rewards are paid for each shareholder that correctly completed the disclosure procedure. The disclosure procedure is detailed in Algorithm 5.

4.4 Economic model

This section illustrates how to constraint the protocol parameters to achieve the desired TL scheme behavior.

Preamble As previously mentioned, ITYT assumes all the participants to be rational agents. This means they are driven by economic interests and they will always try to maximize their profit. So, there is no interest in $\mathcal{S}$ itself, but only in its economic value. Also, as ITYT is built upon secret sharing, it is pivotal to analyze the scenario by referencing to the behavior of groups of adversaries, and not of single users. Thus, we reference by $\mathcal{M}$ a generic malicious coalition of users that team up to break the TL ahead of disclosure time.

Method To limit the attack surface, we model ITYT as a negative-sum game to all the possible adversarial coalitions. In other words, we impose that anyone who tries to break the TL schema has to face costs greater than the maximum achievable revenues. To ensure this condition, we develop a set of constraints among the economic parameters. However, to define each of them we need to identify the maximum revenue an attacker can achieve. For this purpose, we focus on the best possible attack scenario, or rather the one in which an adversarial coalition ideally completes its entire strategy without interference by other parties.

Before going into details, we fix the first trivial constraint. It simply captures, in a single expression, the relation among the amounts discussed in the previous section:

$$W_h < B_{\mathcal{H}} < R_{\mathcal{H}} < W_{\mathcal{S}} \tag{4.1}$$

In the following, we address how coalitions could try to attack the protocol based on the role of its members and the time at which the attack could be performed. For the sake of clarity, Subsections 4.4.1 and 4.4.2 do not take into account the `WhistleblowShare` function, which will be discussed in as special case in Subsection 4.4.3. Finally, Section 4.4.4 addresses how to constraint $P_{\mathcal{O}}$.

4.4.1 Protection against malicious shareholders

Being the shareholder rational agents, they will consider if is worth to try breaking the TL before t_d or not. To perform a successful attack, a shareholder

Algorithm 5 SC function to disclose the share after t_d

```
input
    $sc$        smart contract identifier
    $i$         index of the submitted share
    $h_i$       the $i$-th share
procedure DISCLOSE($sc, i, h_i$)
    if $sc.state$ = LOCKED and $sc.states[i]$ = PAID then
        if time() $\geq sc.t_d$ and time() $< sc.t_{term}$ then
            if hash($h_i$) $= sc.C_h[i]$ then
                $sc.shares\,[i] \leftarrow h_i$
                $sc.num\_disclosed + = 1$
                $sc.states[i] \leftarrow$ DISCLOSED
                $p_1 \leftarrow sc.num\_disclosed \geq sc.k$
                $p_2 \leftarrow$ time() $> t_{term}$
                $p \leftarrow p_1 \wedge p_2$
                allow_withdraw($sc, caller, R_{\mathcal{H}}, p$)
            end if
        end if
    end if
end procedure
```

has to team up with other $k-1$ ones in order to reconstruct $\mathcal{S}$, sell it to a buyer, and finally whistleblow it. In that event, $\mathcal{M}$ would earn at most by selling and by whistleblowing,[2] leading the protocol to failure. The alternative, i.e. do not break the TL and submit the k shares after t_d, would lead to a gain equal to the sum of the rewards. To avoid the first scenario we can impose: $k \cdot R_{\mathcal{H}} > V + W_{\mathcal{S}}$.

Yet, in order to earn $k \cdot R_{\mathcal{H}}$, the coalition $\mathcal{M}$ should wait until t_d, whereas $V + W_{\mathcal{S}}$ could be collected earlier. For this reason, in ITYT we use a stricter formulation of the constraint, obtained by comparing the revenue earned by whistleblowing and the total amount of bids already paid $\mathcal{M}$ to get the shares:

$$k \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}} \tag{4.2}$$

[2] The coalition $\mathcal{M}$ could setup an additional external contract between $\mathcal{M}$ and the buyer to be sure to gain both V and $W_{\mathcal{S}}$.

Constraint (4.2) addresses secrecy (time $t < t_d$), however, when the disclosure time passes, the objective of the TL turns into facilitating the disclosure. In this setting, threshold cryptography may obstacle the release of $\mathcal{S}$, offering additional opportunities to malicious shareholders and coalitions of them. In fact, $n - k + 1$ shareholders could wait for $k - 1$ others to submit their share and then lead the TL instance to a stall by refusing to submit their ones and wait for a buyer willing to pay V to gain access to the secret. To avoid this issue, we introduce the constraint:

$$(n - k + 1) \cdot R_{\mathcal{H}} > V \tag{4.3}$$

Here the contribution of the term $W_{\mathcal{S}}$ disappears as the role of the whistleblower is no longer admissible after t_d. The inequality holds because the shareholders are authorized to collect their rewards $R_{\mathcal{H}}$ only in case the TL results in a successful outcome[3] (i.e., at least k shares are submitted before the termination time t_{term}).

4.4.2 Protection against malicious owners

All the considerations made about the rationality of the shareholders must also be applied to the owner. As $\mathcal{O}$ knows the secret in advance, she could setup fake TL instances for the sole purpose of invoking the `WhistleblowSecret` function and obtain $W_{\mathcal{S}}$, if this would end up having a positive economic return. To enforce a negative outcome for this scenario, we need to add the following constraint:

$$P_{\mathcal{O}} > W_{\mathcal{S}} \tag{4.4}$$

4.4.3 Impact of share whistleblowing function

The ability to whistleblow shares, which is needed to prevent them to be sold, opens up to more complex strategies that can be performed by coalitions. A

[3] An alternative approach to reduce stalls without the need of a termination time and deferred rewards would be to grant an extra reward β to the first k shareholders that submit the share after t_d.

coalition $\mathcal{M}$ could submit some of the controlled shares to the contract before whistleblowing the secret, to maximize its revenue.

As the coalition is composed of rational agents, it is possible to determine the optimal number j^* of shares that $\mathcal{M}$ could submit before incurring into penalties or advantaging other participants. Whistleblowing a share is a public event, observable by anyone, so it could itself trigger other participants' strategies.

The values n and k, the number of total shareholders and the threshold, identify two cases to compute j^*.

General case. In general, a k-shareholders coalition $\mathcal{M}$ is not the only one able to break the TL. Each time a share is whistleblown the threshold k is weakened, and all other $n - k$ shareholders and their possible coalitions are triggered as a side effect. Overall, when i shares have been whistleblown, a quiescent coalition $\mathcal{M}^i$ formed by $k - i$ shareholders would gain the ability to reconstruct the secret $\mathcal{S}$ having paid only $(k - i) \cdot B_{\mathcal{H}}$ to get its shares. For this reason, the initial coalition $\mathcal{M}$ formed by k shareholders needs to determine the optimal number of shares j^* to be whistleblown so that there is no other coalition that can profit from the weakening of k. In particular, $\mathcal{M}$ can determine j^* by solving the following:

$$j^* = \max_i \{i | (k - i) \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}},\ i \in 1, \ldots, k - 1\}$$

If such a j^* exists, by keeping the assumption of rational agents, the coalition $\mathcal{M}$ can submit j^* shares while still being sure that no other smaller coalition will whistleblow the secret.

Special case $k > \lfloor n/2 \rfloor$. In this case, the k-shareholders coalition $\mathcal{M}$ is the only one able to break the TL. Specifically, $2k - n - 1$ shares can be submitted while still preventing the other $n - k$ shareholders to reconstruct $\mathcal{S}$. Please note that such number of submissions could be smaller compared to the one identified by the general case, which is only cost dependent. To compute the optimal number of submissions, it is then required to select:

$$j^* = \max \{(2k - n - 1)\ ;\ {j^*}_{\text{general\_case}}\}$$

Whistleblowing shares permits to gain the extra-revenue $W_{er} = j^* \cdot W_h$. To address it, we formulate a stricter version of Equation (4.2) by constraining:

$$k \cdot B_{\mathcal{H}} > V + W_{\mathcal{S}} + W_{er} \tag{4.5}$$

4.4.4 Evaluating Costs

Now that we've discussed how to constrain the parameters to prevent misbehavior, we discuss the additional requirements to be satisfied by $P_{\mathcal{O}}$. In fact, it is from this amount the owner pays the shareholder profits or the whistleblower bonuses. Overall, the sum of $P_{\mathcal{O}}$ and the shareholder bids $n \cdot B_{\mathcal{H}}$ must be able to cover the costs for every possible evolution of the TL instance. The easiest evolution is represented by all the shareholders correctly executing the protocol. To accommodate for this case, we need to impose that the amount stored in the smart contract is enough to pay the shareholder rewards:

$$P_{\mathcal{O}} + n \cdot B_{\mathcal{H}} \geq n \cdot R_{\mathcal{H}} \tag{4.6}$$

On the contrary, the TL could fail with $k - 1$ shares whistleblown before the secret itself is whistleblown. To ensure the smart contract has enough value to be able to pay all the whistleblower bonuses, we impose the following constraint:

$$P_{\mathcal{O}} + n \cdot B_{\mathcal{H}} \geq (k - 1) \cdot W_h + W_{\mathcal{S}} \tag{4.7}$$

As $B_{\mathcal{H}} > W_h$, we do not need to discuss mixed evolutions of the TL. In fact, the right side of Inequality (4.7) reports the worst-case amount. In any other circumstance (i.e., less than $k - 1$ shares whistleblown), the Inequality (4.7) still holds.

The constraints (4.1-4.7) must all hold for any well-formed instance of ITYT. They identify the acceptance area for the variables. The owner may desire to minimize $P_{\mathcal{O}}$, while the shareholders would like to maximize the profit $(R_{\mathcal{H}} - B_{\mathcal{H}})$. A more thorough analysis of the ratio among these variables and the different strategies to set them is out of scope in this chapter. In

V	W_h	$B_{\mathcal{H}}$	$R_{\mathcal{H}}$	$W_{\mathcal{S}}$	$P_{\mathcal{O}}$
1	0.025	0.275	0.3	0.325	0.35
10	0.25	2.75	3	3.25	3.5
100	0.5	25.5	26	26.5	27

Figure 4.3: Sample configurations with $k = 5$ and $n = 8$ and $V \in 1, 10, 100$ that respect all the constraints (optimized for minimizing $P_{\mathcal{O}}$)

Table 4.3 we show three possible configurations for $k = 5$, $n = 8$, and $V \in 1, 10, 100$, which minimize $P_{\mathcal{O}}$, obtained by constraint programming. It is worth to remark that in all of the cases we have $P_{\mathcal{O}} < V$, which is a desirable property.

4.5 Implementation

This section illustrates how to practically implement ITYT.
As we anticipated in Sections 4.1 and 4.2, the following two technologies are the pillars of our construction:

- the blockchain, the technological layer which allows us to publicly store the evolution of each instance of ITYT and persistently disclose the secret associated to it;
- smart contracts, the framework that permits to trigger economic incentives and penalties in a fully decentralized way, thus avoiding the need of trusted parties.

The combination of the two with the model described in Section 4.3, constitutes ITYT: a generic protocol to deploy instances of the TL abstraction. Several realizations of the ITYT protocol can be obtained by changing the implementation of the `generate_shares` primitive and the cryptographic techniques (e.g., oblivious transfer, probabilistic encryption, and sMPC) used to provide the shares only to the legitimate shareholder while sending the commitments to the owner. In this chapter we implement the `generate_shares` primitive leveraging secure Multi-Party Computation. In particular, we employ a modular arithmetic sMPC to be able to compute the

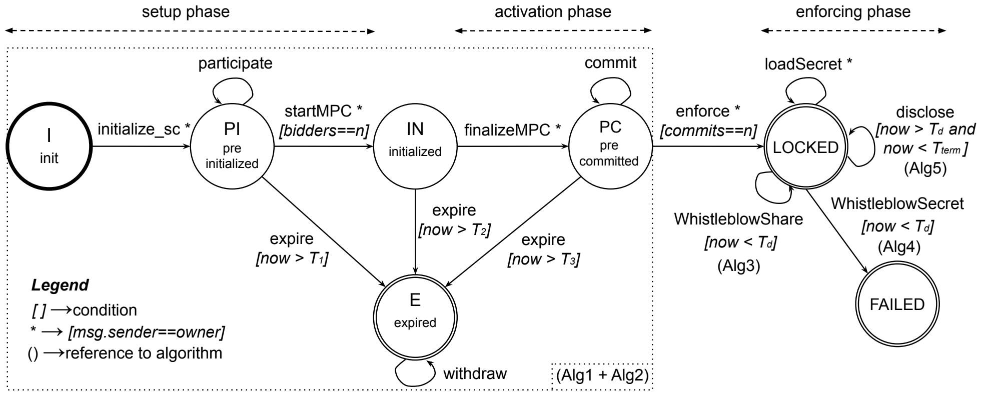


Figure 4.4: State machine representing the valid transitions of the $ITYT$ protocol. Each transition name maps to an action (an Ethereum smart contract function) that can be invoked by each participant to modify the state. Square brackets contain additional conditions to be met to make valid transitions

shares using Shamir's Secret Sharing. In addition, we use the *MiMC-hash* [3] algorithm to implement the `hash` primitive used to compute the commitments. MiMC-hash is a minimal multiplicative complexity hashing primitive that has the advantage of being composed of only modular arithmetic operations, and is thus suitable to be used in our arithmetic sMPC (further details on this in Section 4.7).

4.5.1 The ITYT state machine

We designed $ITYT$ as a finite state machine implemented in an Ethereum smart contract. The state of each of its instances can be evolved by performing actions that map to specific smart contract functions. Before performing any update, each function checks for the state of the machine to be the expected one and terminates otherwise. A simplified version of the state machine that only shows valid transitions is presented in Figure 4.4. The state machine is characterized by three main phases: the `setup` phase, in which the owner has to deploy the contract and the shareholders have to commit to take part to it; the `activation` phase, in which the shareholders declare that they received their shares and give their go-ahead; and the `enforcing` phase, in which the participants comply with the constraints of the contract thus effectively realizing the TL primitive as a consequence. In between the `setup` and the `activation` phases, the owner and the shareholders collaboratively execute the *off-chain* sMPC protocol that generates and distributes the shares and the commitments. These phases are discussed in details in the following.

Setup phase Initially, the owner deploys a smart contract instance of $ITYT$ on the Ethereum blockchain. As detailed in Algorithm 1, the owner invokes the `initialize_sc` primitive, she transfers P_O to the contract and she specifies all the instance parameters detailed in Section 4.3. This advances the protocol to the `PRE_INITIALIZED` state. To demonstrate the will to correctly participate, each shareholder invokes the smart contract function `participate`, as part of Algorithm 2, and transfers to the contract the proper amount of Ether corresponding to the bid, $B_{\mathcal{H}}$. The owner observes the actions taken by all the participants by reading the state of the contract and, after she ensured that all the shareholders have deposited their bids, she invokes the function `startMPC` which permits to advance the state to `INITIALIZED`. This opens the off-chain sMPC window, in which all the participants cooperate to realize the `generate_shares` protocol.

Off-chain sMPC The sMPC permits to generate the shares of the secret, confidentially distribute them to the shareholders, while forwarding the commitments to the owner. However, it is executed when the TL instance is still inactive, or rather when the economic penalties have not been activated yet. To avoid the exposition of $\mathcal{S}$, the owner introduces a fictitious random key, $\mathcal{K}$. The key is submitted to the sMPC instead of the secret, and a set of n shares $\{h_1, \ldots, h_n\}$, is produced. Specifically, the shares are derived from the key using the Shamir's Secret Sharing algorithm [105], so that it is possible to reconstruct $\mathcal{K}$ out of k-*of*-n shares. Accordingly, the owner can wait to the activation of the TL and then upload into the smart contract the encrypted version (i.e., the ciphertext) of the secret, $CT = \mathcal{E}nc_{\mathcal{K}}(\mathcal{S})$. Figure 4.5 illustrates the scenario.

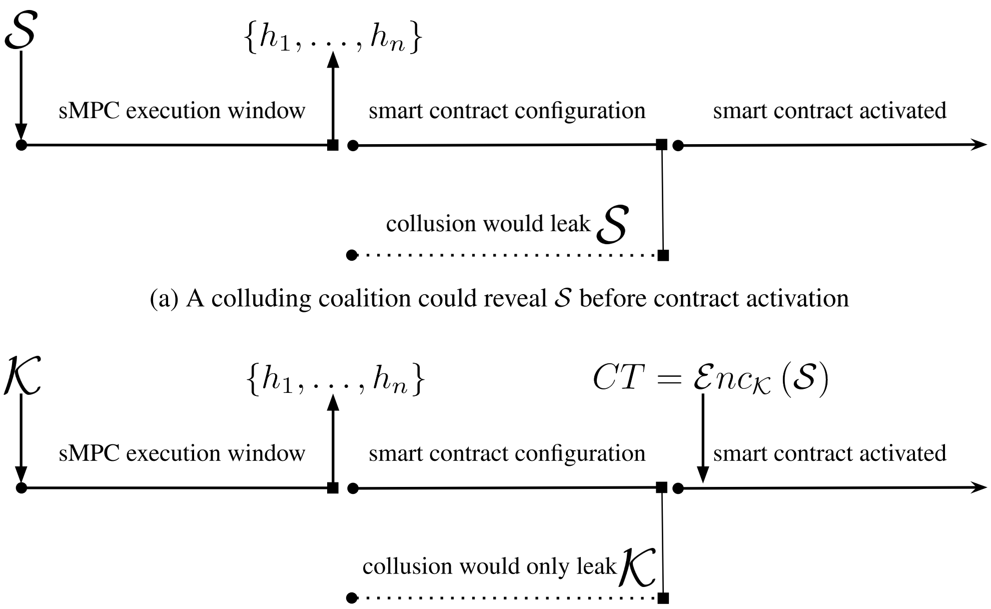
To execute the sMPC protocol the owner has to input the random key, $\mathcal{K}$, together with the total number of shareholders, n, and the reconstruction threshold, k. Each shareholder, instead, submits only a random seed. The sMPC generates internally random values which are interpreted as the x coordinates by the Shamir's Secret Sharing algorithm. Each i-th share is then generated by the concatenation of x_i and y_i coordinates. As output of the sMPC protocol, $\mathcal{O}$ receives a commitment C_i of any share generated,[4] while each shareholder receives her share h_i, a commitment of the key $C_{\mathcal{K}}$, n and k. A graphical view of the discussed sMPC protocol is shown in Figure 4.6.

Activation phase When the sMPC terminates the owner invokes the function `finalizeMPC` and updates the smart contract state with the commitment of the key, $C_{\mathcal{K}}$, along with the commitments of all the shares, $\{C_1, ..., C_n\}$. As a consequence, the state of ITYT turns to `PRE_COMMITTED`. Each shareholder is now asked to verify the correctness of the data deposited in the contract by $\mathcal{O}$. In particular, if the commitment written by the owner matches to the one given by the shareholder (i.e., the owner did not tamper C_i), then each shareholder would invoke the `commit` function. After each of them committed, the owner invokes `enforce`, which activates the TL (i.e., activation of all economic incentives and penalties). The state turns to `LOCKED`, and the `activation` phase ends.

The `setup`, `off-chain sMPC`, and `activation` phase together realize the Algorithms 1 and 2, needed to conclude the setup of the ITYT instance.

Enforcing phase Before disclosure time, the smart contract permits to: (i) whistleblow a single share, and (ii) whistleblow the secret. To whistleblow a single share (Algorithm 3) a participant invokes `WhistleblowShare` and submits h'. If the commitment matches the value stored in the contract, then W_h is immediately paid to the whistleblower. A share whistleblow is permitted only $k-1$ times, as a greater number of submission leads the TL to the `FAILED` state. When that happens the remaining amount locked by the contract is destroyed. To whistleblow the secret (Algorithm 4) it is required to invoke the `WhistleblowSecret` function and submit $\mathcal{K}'$. If the commit-

[4] By share commitment we denote the MiMC hash of the share.


$\mathcal{S}$
$\{h_1, \ldots, h_n\}$
sMPC execution window
smart contract configuration
smart contract activated
collusion would leak $\mathcal{S}$
$\mathcal{K}$
$\{h_1, \ldots, h_n\}$
$CT = \mathcal{E}nc_{\mathcal{K}}(\mathcal{S})$
sMPC execution window
smart contract configuration
smart contract activated
collusion would only leak $\mathcal{K}$

(a) A colluding coalition could reveal $\mathcal{S}$ before contract activation

(b) A colluding coalition could only reveal $\mathcal{K}$ before contract activation

Figure 4.5: Preventing secret exposure before smart contract activation

ment of $\mathcal{K}'$ matches the one of $\mathcal{K}$, then $W_{\mathcal{S}}$ is paid to the whistleblower, the TL instance is marked as FAILED, and the remaining amount destroyed.

If no adversarial coalition reconstructs $\mathcal{K}$ before the disclosure time t_d, then all the shareholders will be able to disclose their shares by invoking the function disclose (Algorithm 5). Each share is considered valid only if it matches the corresponding commitment. In that case, a reward will be paid to the shareholder conditionally to the outcome of the instance (i.e., the TL ends successfully only if at least k valid shares are submitted before t_{term}). There is no need to materialize $\mathcal{K}$ or $\mathcal{S}$ in the smart contract, as all the valid shares will be permanently and publicly accessible. Anyone can recover $\mathcal{K}$ from the shares and then extract the plaintext performing the decryption operation, $\mathcal{S}$ $= \mathcal{D}ec_{\mathcal{K}}(CT)$.

Up until now, the protocol has only been based on the key $\mathcal{K}$, but the $\mathcal{S}$ has not been offloaded to the TL yet. The explanation is straightforward: the outcome of $ITYT$ is only bounded to the disclosure of the key, to which all the incentive and penalties are related to. As soon as the state of the TL advances to LOCKED and before going offline, the owner encrypts the secret

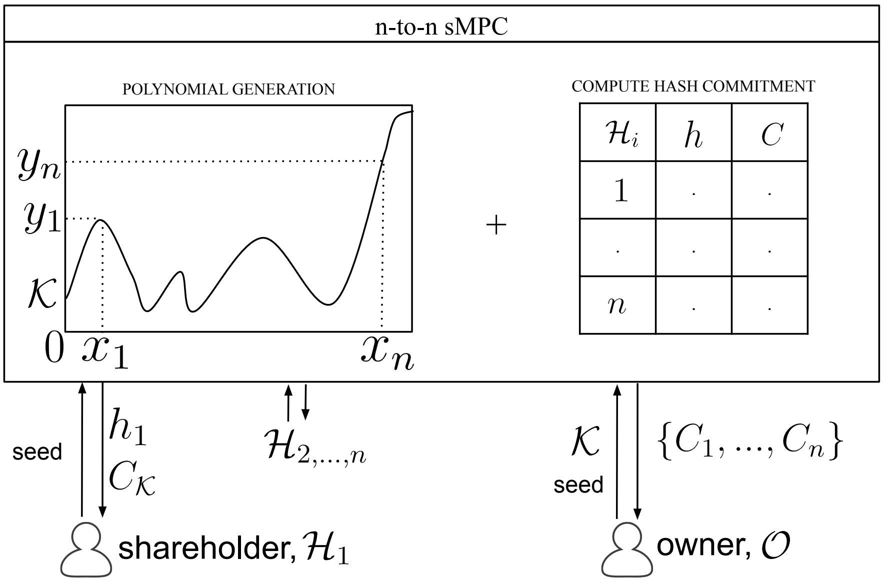


Figure 4.6: Single-phase sMPC protocol jointly executed by all the participants

$\mathcal{S}$ as shown in Section 4.5.1 to obtain the CT. Then she invokes the smart contract function `loadSecret` to store the CT in the smart contract.

4.6 Discussion

In this section, we highlight the vulnerabilities to which the implementation of ITYT can be subject to, and then we discuss the proper countermeasures.

4.6.1 Protection against adversarial sMPC protocols

The discussion so far has focused on the single user perspective regarding the sMPC cryptographic technique. In fact, its usage guarantees to each shareholder $\mathcal{H}$, that no one but her (i.e., neither the owner) will have visibility on her share, h. Thus, it ensures to each shareholder that no one, except herself, will be able to submit her share at disclosure time. This capability limits the knowledge of every single user about an instance ITYT, hence it makes possible to model ITYT as an economic game.

Nevertheless, ITYT is a game played by rational adversaries, so the participants could look at sMPC techniques from another perspective. Actually, the malicious coalition of shareholders, $\mathcal{M}$, could devise a strategy that permits to recover and sell $\mathcal{S}$ without incurring in economic penalties. Such a strategy would have been successful if it did not allow anyone to satisfy the smart contract commitments. In our setting, it means to recover $\mathcal{S}$ without exposing $\mathcal{K}$, and therefore preventing anyone to invoke the `WhistleblowSecret` function. The malicious coalition can instantiate an offensive sMPC that receives as input at least k-of-n shares plus the ciphertext, CT. Then, it performs internally both the reconstruction of $\mathcal{K}$ and the extraction of the secret by $\mathcal{D}ec_{\mathcal{K}}\left(CT\right)$. Finally, it outputs to all the attackers only $\mathcal{S}$.

To prevent the execution of this attack it is sufficient to select a cryptosystem $\mathcal{C}\left(\mathcal{E}nc, \mathcal{D}ec\right)$ such that there is no sMPC protocol that permits to execute $\mathcal{D}ec$ by producing $\mathcal{S}$ as the only output. We identified two possible alternatives to satisfy the previous requirement.

1. Use a cryptosystem that ensures that the exposure of $\mathcal{S}$ implies exposing also the $\mathcal{K}$ (by the key it is possible to perform the whistleblow protocol).

2. Use a cryptosystem whose execution is not compatible with sMPC protocols.

We can satisfy the first condition by selecting $\mathcal{C} = \mathit{OneTimePad}\left(\oplus, \oplus\right)$. It is easy to prove that given two among $\{CT, \mathcal{S}, \mathcal{K}\}$ the third is implied. Thus, each member of the malicious coalition could invoke the `WhistleblowSecret` function after obtaining $\mathcal{S}$. The drawback of using $\mathit{OneTimePad}\left(\oplus, \oplus\right)$, is that $|\mathcal{S}| = |\mathcal{K}|$ by construction. This limitation can be overcome by selecting a cryptosystem that satisfies the second condition; however, devising a strategy to construct such a cryptosystem goes beyond the scope of this chapter.

The consideration that stems from the use of adversarial sMPC is interesting, as this kind of attacks (i.e., programming the logic of a protocol inside an sMPC to bypass hashlocks) are applicable to all protocols that involve rewards, penalties and are played by rational adversaries.

n	gas cost $\mathcal{O}$ (\$)	gas cost $\mathcal{H}$ (\$)
3	162872 [5.80\$]	23971 [0.85\$]
4	171793 [6.11\$]	25545 [0.91\$]
5	180707 [6.43\$]	25856 [0.92\$]
6	189647 [6.75\$]	26167 [0.93\$]
7	198587 [7.07\$]	26478 [0.94\$]
8	207501 [7.38\$]	26789 [0.95\$]
9	216442 [7.70\$]	27100 [0.96\$]
10	225337 [8.02\$]	27411 [0.98\$]

Figure 4.7: ITYT cost for each role, with $k = 2$.
Conversion units: 1 gas = $20 \cdot 10^{-9}$ ETH; 1 ETH = 178 $

4.6.2 DOS Attacks and Deadlocks prevention

A denial of service attack could be performed by some users who take part in many ITYT protocol instances refusing to deposit the bids, to commit or to correctly execute the sMPC. To mitigate these kinds of disruptions it is possible to introduce a reputation system, so that the owner can select parties that are willing to co-operate. This would be an ideal choice that could benefit also other parts of our model. However, as the introduction of a reputation model is not always possible, we decided to model the smart contract `PRE_CONSIDERED` state. Specifically, all the participants (including the owner) are required to pay an additional small service pawn that will be returned only if the smart contract will reach the `LOCKED` state. It has been proved that introducing a small fee to access a service can prevent many DOS attacks [78, 85]. Furthermore, some malfunctions or network errors may imply the time thresholds set by the owner not being met. The deadlock to which the protocol would lead to can be solved by the presence of the final state `EXPIRED`. In that event, all the participants are allowed to withdraw their funds locked in the contract.

4.7 Experimental Results

Test setup We performed two types of experiment on $ITYT$: (i) simulation of $ITYT$ smart contract instances, and (ii) execution of sMPC network protocols. For both cases, the size of the shares was set to 256 bits (the maximum supported by the FRESCO sMPC framework). The tests have been executed on a dual Intel Xeon E5 server with 256 GB memory and 512 GB SSD drive running Ubuntu 18.04 LTS.

Simulation of ITYT instances A preliminary version of the $ITYT$ smart contract was derived by the use of FSolidM [86], a tool which automatically generates Ethereum smart contracts code from high-level Finite State Machine (FSM) representations. To deploy, test, and debug the contract created, we relied on Brownie [58], a Python framework that permits to create wallets, inspect transactions and automate the execution of simulations. To provide such functionalities, Brownie interacts with Ganache [110]: a local Ethereum blockchain instance used for development purposes.

The experiments mainly focused on estimating the total cost of execution of each $ITYT$ instance. Figure 4.7 shows the total execution cost in gas (a proxy for the number of operations) and dollars, for each role, depending on the number of participants. Since many ITYT actions imply writing a state change to the blockchain, the first shareholder to perform the operations will incur in greater costs compared to the other ones. We name this role *active* shareholder. The shareholder cost in Figure 4.7 takes into account only the calls needed to collect the reward. A shareholder that only performs these operations is named *passive* shareholder. In our experiments, the active shareholder was paying approximately 0.40 USD more compared to the passive one. It is easy to setup a higher reward for the *active* shareholder to offset the additional execution fees. From the owner's perspective, the total cost of execution is linear in the number of shareholders, while the contract deployment itself costs 7.39 USD. All the costs were calculated by applying the conversion rate $20 \cdot 10^{-9}$ ETH and the exchange rate 1 ETH $=$ 178 USD (as of September 12, 2019).

sMPC protocols The sMPC protocols were implemented using the FRESCO [5, 41] framework. First, we implemented a single-phase sMPC protocol compliant with the description in Section 4.5.1. The sMPC algo-

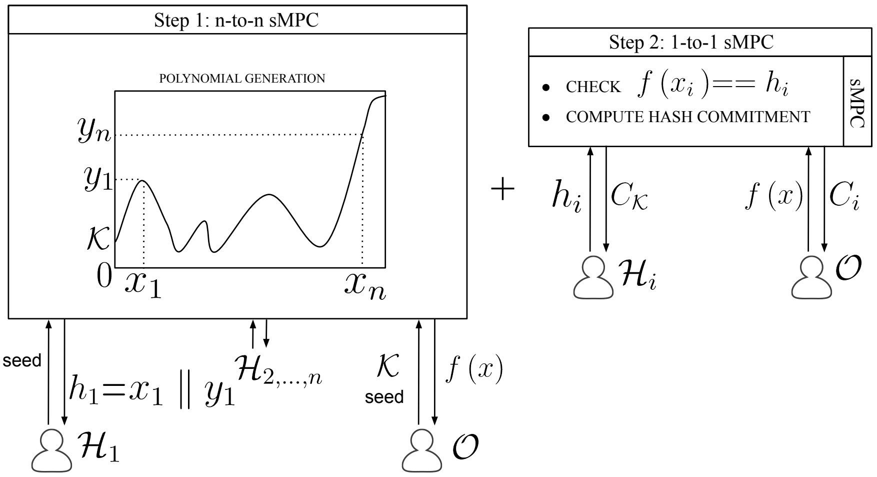


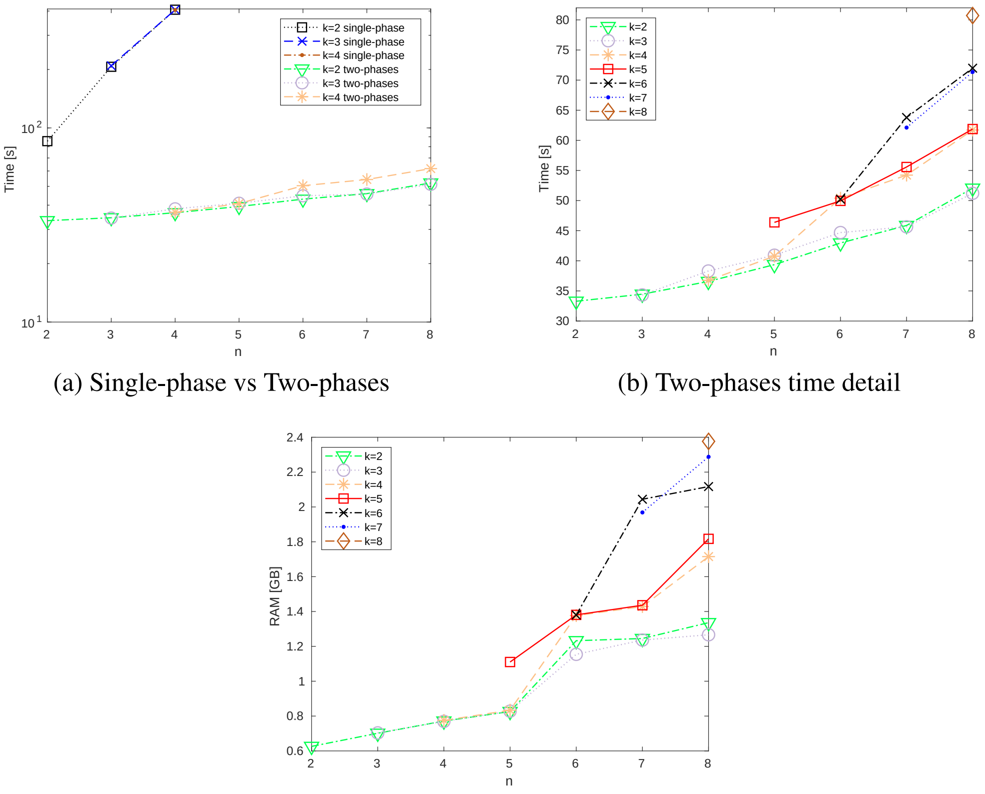
Figure 4.8: Two-phases sMPC protocol: step 1 jointly computed between all the parties, step 2 between the owner and each shareholder

rithm receives as input the key $\mathcal{K}$ from the owner $\mathcal{O}$ and the seeds from all the participants. Then, it creates the polynomial and generates random shares. After that, the algorithm proceeds in computing the commitments with the MiMC function. As illustrated in Figure 4.9a, strictly adhering to this protocol leads to quick performance degradation, as the number of shareholders increases. This is because computing MiMC hashes among several participants is computationally expensive and highly affected by network latency (the participants have to exchange several messages to carry out even simple operations in the sMPC setting).

To improve performance, we implemented a two-phases sMPC protocol. This, in turn, is made up of two subsequent sMPC protocols:

- *Step 1* n-to-n sMPC jointly computed by all participants. The owner inputs $\mathcal{K}$ and receives the polynomial function $f(x)$ randomly generated inside the sMPC, while each shareholder inputs a seed and receives her share (see Figure 4.8).

- *Step 2* 1-to-1 sMPC computed between the owner and each shareholder i. The owner inputs $f(x)$, and receives the commitment of the i-th share, C_i, while the i-th shareholder inputs her share h_i, and receives the commitment of the key (see Figure 4.8). It is easy to verify inside



(a) Single-phase vs Two-phases

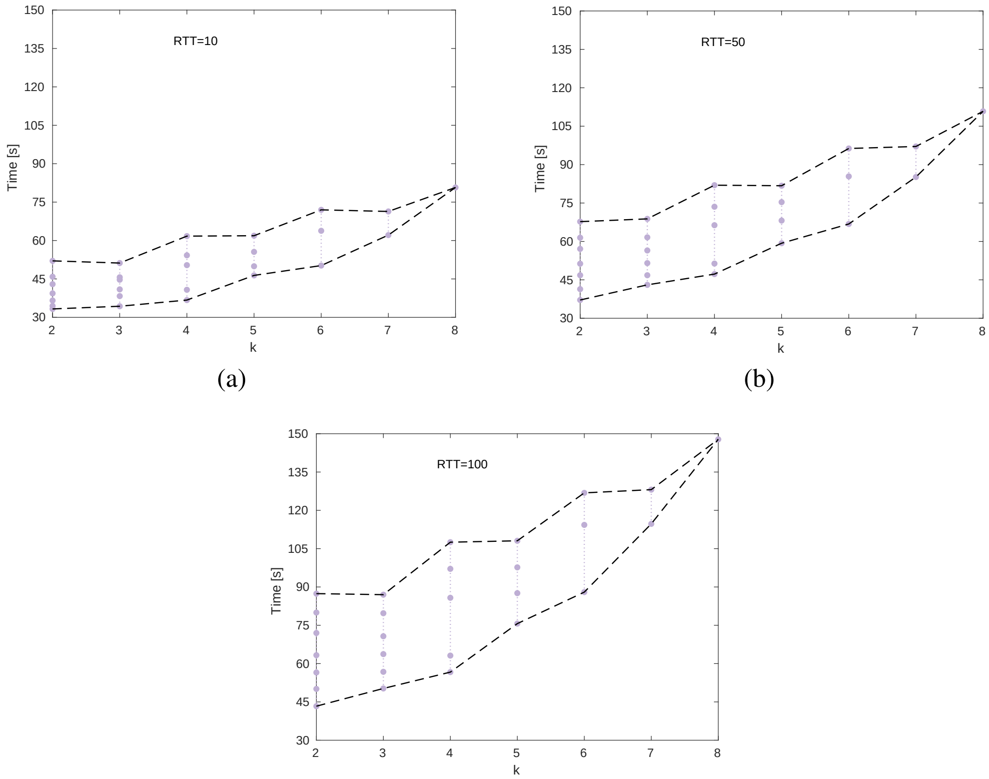
(b) Two-phases time detail

(c) Two-phases maximum memory usage

Figure 4.9: sMPC execution time and maximum memory consumption for each participant, in detail: (a) comparison between single-phase and two-phases maximum execution time, (b) two-phases execution time and (c) maximum memory consumption for higher SSS polynomial degree

the sMPC that neither the owner nor the shareholder changed the value received as a result of the first step.

The difference between the single-phase and the two-phase sMPC lies in the calculation of the hash primitive. Unlike the single-phase version, the two-phases solution permits to separate the generation of the shares from the production of commitments. It follows that the first step can be carried out even in scenarios with several participants, as it is not computing-intensive, whereas the second step, which instead is computing-intensive, is always performed among two users. The improvement of total sMPC execution time between the two variants (i.e., single-phase vs two-phases) is shown in Fig-



(a)

(b)

(c)

Figure 4.10: sMPC total execution time for different round trip times (RTT) and reconstruction threshold k

ure 4.9a. More details about the two-phases maximum execution time and maximum memory consumption in case of higher polynomial degree, for each participant, are illustrated in Figures 4.9b-c, respectively.

For each participant, a different server was established, and the network communication round trip time (RTT) was set to 10 ms (Figure 4.9). To measure the effect of RTT, we repeated the execution of all network protocols with different latency values. Figure 4.10 shows a subset of the data collected for the two-phases solution (see the repository for full detail). However, for an exhaustive analysis on the impact of environmental parameters over sMPC implementing secret sharing cryptography, we refer the reader to [111].

4.8 Related Work

The use of cryptography to solve the problem of unveiling private data at a specific time in the future was first envisioned in 1993 by Timothy May [87]. Since then many researchers have proposed solutions to this problem. Based on the assumptions and technologies used, we can classify the proposals into four main categories.

Trust and Honesty — Chan et al. [35] and Cheon et al. [37] proposed single point of trust schemes, in which users can encrypt secrets using public keys, while the decryption keys are released at predefined times by a trusted time server. Rabin et al. described Time-lapse cryptography [92,99] that overcomes the single point of trust (and failure) assumption by splitting the single authority into a group of users that have to cooperate to release the keys. Li et al. [73] proposed a solution that relies on Distributed Hash Tables to route the secret among peers. Both these proposals entail the peers to be honest as they do not consider the possible economic benefits that the participants would obtain by colluding.

Time-Lock Puzzles — The second category requires the recipient to solve an inherently sequential mathematical puzzle to prove the elapse of time. Starting from Rivest et al. seminal work [104], many other puzzles have been proposed [27, 38, 83]. All these techniques are not practical as the required time might differ on different machines, and the computation costs for the recipient are significant.

Smart Contracts — Similarly to our proposal, the third category leverages smart contracts [107] to replace the trusted party. Kimono [51, 88] and Keep Network [81] rely on threshold cryptography to split the secret among participants that can earn a remuneration by keeping their shares private until disclosure time. However, they do not introduce security deposits, thus failing at preventing misbehaviors. Li et al. proposal [74] overcomes some of these limitations by modeling the protocol as an extensive-form game with imperfect information [72]. Yet, as each peer is a single point of failure, and as the owner has perfect information about the shares, they require every participant to pay a security deposit that exceeds the value of the secret, limiting the applicability of the protocol. Compared to our protocol, all the proposals in this category do not consider the fact that malicious coalitions could reconstruct the secret ahead of time inside an sMPC protocol without exposing the shares, thus effectively avoiding penalties and safeguarding remunerations.

Witness Encryption — The last category of solutions leverages witness encryption [53], in which the sender can encrypt a message so that it can only be opened by a recipient who knows a witness to an NP relation. Liu et al. [77] showed how to construct a computational reference clock from large public computations, such as those made by the Bitcoin network, and couple it with witness encryption to realize a time-lock encryption mechanism. Yet, due to the lack of practical witness encryption schemes, their proposal is still far from being concrete.

Other Contributions — Several recent proposals try to address the problem of dealing with secret data on public blockchains. Enigma [119] and Hawk [70] leverage sMPC to allow multiple actors to execute an algorithm on private inputs and store the proof of correct execution on the blockchain. However, these proposals require the data holder to actively participate in the computation, thus they can not be used to solve the problem of data disclosure at a future point in time. *Proof of Elapsed Time* (*PoET*) is a network consensus algorithm often used in permissioned blockchains, like Hyperledger Sawtooth [36, 62] that avoid wasting computational resources by using a fair lottery system run inside a Trusted Execution Environment (TEE), such as Intel SGX. Each participant runs an algorithm in the TEE that waits for a random amount of time, thus proving the elapse of time without the need of PoW. Even if this approach resembles ITYT, as it prevents cheating on the chosen time, it is not able to store secret data. Another recent contribution that showed how economic constraints enforced by time-lock primitives can be successfully integrated with blockchains is the *Bitcoin Lightning Network* [97]. Lightning Network can be used to instantly exchange bitcoins among peers by using off-chain transactions while effectively preventing any possible misbehavior.

4.9 Conclusions

In this chapter, we presented I Told You Tomorrow (ITYT), a practical schema to deploy Time-Locked secrets on the blockchain by leveraging an economic model in which every rational actor (or coalition of them) is economically incentivized to correctly participate. As opposed to other Time-Lock mechanisms, ITYT requires neither trust assumptions, nor a receiving party willing to run the decryption algorithm until the disclosure time, nor guesses about future computing power. Our implementation and experimental evaluation showed that our approach is practical, as the cost and the required computing power associated are limited.

As future work, we are considering the use of homomorphic encryption for the share generation process. This will not require any changes to the protocol itself, but only to the share generation algorithm. By doing so, we expect that the time and resources needed to setup the protocol will be further reduced. We are also interested in conducting a more thorough analysis of the inequalities in Section 4.4 from an economic standpoint, to better characterize the correlations among the parameters.

5

Conclusions and Future Work

In this thesis, we presented novel techniques for providing data confidentiality, access control, and availability guarantees to resources stored in centralized, decentralized, and distributed cloud storage systems.

As regards centralized cloud storage providers, we presented an approach for efficiently enforcing access revocation on encrypted resources stored at external providers. Our solution enables data owners to effectively revoke access by re-encrypting a small portion of the resource. As opposed to techniques based on other *All-or-Nothing Transforms* in which an actor that has access to the file could try to maintain a local copy of the keys to counter access revocation, *Mix&Slice* enforces that the amount of data that the malicious actor would have to store can not be determined a priori, as the data owner could re-encrypt a varying portion of the resource. Our implementation and experimental evaluation confirm the efficiency and effectiveness of our proposal.

We then analyzed how to extend the guarantees that we obtained with *Mix&Slice* to provide effective secure protection to decentralized cloud storage providers. Our approach enables resource owners to protect their resources and to control their decentralized allocation to different nodes in the network. We investigated different strategies for splitting and distributing resources and analyzed them in terms of availability and security guarantees.

Our proposal allows the data owner to control the granularity of slicing and diversification of allocation to ensure aimed availability and security guarantees even in presence of malicious coalition of nodes that are willing to disobey the owner's requests to maximize their revenue.

Finally, we presented I Told You Tomorrow (ITYT), a practical schema to deploy Time-Locked secrets on the blockchain. Our proposal leverages an economic model in which every rational actor (or coalition of them) is economically incentivized to comply with the protocol. This is a fundamental piece to create delegated challenge-response protocols in which the challenges and responses are offloaded in an encrypted form by the data owner and then decrypted at different time. The use of delegate challenge-response protocols permits to remove the coordinator nodes and bring the security and availability guarantees to the fully distributed settings without the need of the data owner or a trusted third party to be online.

5.1 Future Work

We conclude the thesis with a discussion of the future work that can be done in the three considered areas: *centralized*, *decentralized*, and *distributed* storage systems.

Centralized storage systems – Chapter 2 illustrated how to use *Mix&Slice* to enable the use. The two implementations discussed in the evaluation section were based on *AES* and *OAEP*. We showed how the use of *OAEP* enables to obtain micro-blocks with size up to 256 bits. We used the *SHA2* class of functions to compute the hashes; however, these functions are not optimized in common CPUs.

As future work, it is possible to compare the results with other functions that also comply with the requirements in Section 2.2.3. A preliminary evaluation with the linear transformations proposed by Karame et al. for *Bastion* [67] and by Naor et al. [89] showed promising results.

Decentralized storage systems – Chapter 3 described how to bring security and availability guarantees to the decentralized cloud storage scenario.

We also briefly mentioned how these properties could be integrated with fountain codes to leverage the dynamicity of DCS networks. In [17] we showed how the integration seems to be a good fit to address the dynamicity problem. This idea has yet to be explored in details; Future work in this area consists of implementing and evaluating different codes and protocols to fully exploit the dynamicity of the system.

Distributed storage systems – Chapter 4 showed how to deploy time-locked secrets on the blockchain, and briefly discussed how this primitive can be used for delegated challenge-response protocols. Future work in this area can formalize *delegated challenge-response* protocols and show how they can be used in several scenarios such as inheritance management, voting processes, autonomous organizations, and Proof-of-Retrievability for cloud storage.

Moreover, as already discussed in the chapter, the use of secure Multi-Party Computation is not the only way to realize a secure share generation and distribution protocol. We are already considering the use of homomorphic encryption, which would further reduce the time and resources required to setup the ITYT protocol.

Acknowledgments

The research documented in this thesis was supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo) and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644579 (EscudoCloud) and No. 825333 (Mosaicrown).

I would like to thank Prof. Stefano Paraboschi for the guidance during this journey, together with all the people that had a positive impact on my life, both social and academic.

Bibliography

[1] H. Abelson, R. Anderson, S. M. Bellovin, J. Benalob, M. Blaze, W. Diffie, J. Gilmore, P. G. Neumann, R. L. Rivest, J. I. Schiller, and B. Schneier, "The risks of key recovery, key escrow, and trusted third-party encryption," *World Wide Web J.*, vol. 2, pp. 241–257, 1997.

[2] M. Albanese, S. Jajodia, R. Jhawar, and V. Piuri, "Dependable and resilient cloud computing," in *Proc. of IEEE SOSE*, March 2016.

[3] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2016, pp. 191–219.

[4] A. Aldribi, I. Traore, and G. Letourneau, "Cloud slicing a new architecture for cloud security monitoring," in *Proc. of IEEE PACRIM*, Victoria, Canada, August 2015.

[5] Alexandra Institute, "FRESCO - a framework for efficient secure computation," https://github.com/aicis/fresco.

[6] J. Alwen, C. Cachin, O. Pereira, A.-R. Sadeghi, B. Schoenmakers, A. Shelat, and I. Visconti, "Summary report on rational cryptographic protocols," 2007.

[7] E. Andreeva, A. Bogdanov, and B. Mennink, "Towards understanding the known-key security of block ciphers," in *Proc. of FSE*, Hong Kong, November 2014.

[8] G. Asharov, R. Canetti, and C. Hazay, "Toward a game theoretic view of secure computation," *J. Cryptol.*, vol. 29, pp. 879–926, 2016.

[9] M. Atallah, K. Frikken, and M. Blanton, "Dynamic and efficient key management for access hierarchies," in *Proc. of CCS*, Alexandria, VA, USA, November 2005.

[10] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Access control management for secure cloud storage," in *Proc. of the 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2016)*, 2016.

[11] E. Bacis, A. Barnett, A. Byrne, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Distributed shuffle index: Analysis and implementation in an industrial testbed," in *Proc. of the 5th IEEE Conference on Communications and Network Security (CNS 2017).* IEEE, 2017, poster.

[12] E. Bacis, S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Livraga, S. Paraboschi, M. Rosa, and P. Samarati, "Multi-provider secure processing of sensors data," in *Proc. of the 17th IEEE International Conference on Pervasive Computing and Communications (PerCom 2019).* IEEE, 2019.

[13] E. Bacis, S. De Capitani di Vimercati, S. Foresti, D. Guttadoro, S. Paraboschi, M. Rosa, P. Samarati, and A. Saullo, "Managing data sharing in openstack swift with over-encryption," in *Proc. of the 3rd ACM Workshop on Information Sharing and Collaborative Security (WISCS 2016).* ACM, 2016.

[14] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Mix&Slice: Efficient access revocation in the cloud," in *Proc. of the 23rd ACM Conference on Computer and Communication Security (CCS 2016).* ACM, 2016.

[15] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Distributed shuffle index in the cloud: Implementation and evaluation," in *Proc. of the 4th IEEE International Conference on Cyber Security and Cloud Computing (IEEE CSCloud 2017)*. IEEE, 2017.

[16] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Protecting Resources and Regulating Access in Cloud-based Object Storage," in *From Database to Cyber Security: Essays Dedicated to Sushil Jajodia on the Occasion of his 70th Birthday*. Springer, 2018.

[17] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Dynamic allocation for resource protection in decentralized cloud storage," in *Proc. of the 2019 IEEE Global Communications Conference (GLOBECOM 2019)*. IEEE, 2019.

[18] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Securing resources in decentralized cloud storage," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 15, no. 1, pp. 286–298, 2019.

[19] E. Bacis, D. Facchinetti, M. Rosa, M. Guarnieri, and S. Paraboschi, "I *Told* You Tomorrow: Practical time-locked secrets using smart contracts," (under submission).

[20] E. Bacis, M. Rosa, and A. Sajjad, "EncSwift and key management: an integrated approach in an industrial setting," in *3rd Workshop on Security and Privacy in the Cloud (SPC 2017)*. IEEE, 2017, pp. 483–486.

[21] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology—CRYPTO'98*. Springer, 1998, pp. 26–45.

[22] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *Advances in Cryptology—EUROCRYPT'94*. Springer, 1995, pp. 92–111.

[23] J. Benet, “IPFS-content addressed, versioned, P2P file system,” Protocol Labs, Tech. Rep., 2014.

[24] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, “DepSky: Dependable and secure storage in a cloud-of-clouds,” *ACM TOS*, vol. 9, no. 4, pp. 12:1–12:33, 2013.

[25] A. Biryukov and D. Khovratovich, “PAEQ — reference v1,” University of Luxembourg, Tech. Rep. CryptoLUX, 2014.

[26] A. Biryukov and D. Khovratovich, “PAEQ: Parallelizable permutation-based authenticated encryption,” in *Proc. of ISC*, Hong Kong, China, Oct. 2014.

[27] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters, “Time-lock puzzles from randomized encodings,” in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ser. ITCS ’16, 2016, pp. 345–356.

[28] K. D. Bowers, A. Juels, and A. Oprea, “HAIL: A high-availability and integrity layer for cloud storage,” in *Proc. of ACM CCS*, Chicago, IL, USA, November 2009.

[29] K. D. Bowers, A. Juels, and A. Oprea, “Proofs of Retrievability: Theory and implementation,” in *Proc. of ACM CCSW*, Chicago, IL, USA, November 2009.

[30] V. Boyko, “On the security properties of oaep as an all-or-nothing transform,” in *Proceedings of the 19th International Cryptology Conference (CRYPTO)*, 1999.

[31] G. Branwen, “Time-lock encryption,” https://www.gwern.net/Self-decrypting-files, 2018.

[32] P. Caballero-Gil, C. Hernández-Goya, and C. Bruno-Castañeda, “A rational approach to cryptographic protocols,” *CoRR*, vol. abs/1005.0082, 2010.

[33] C. Cachin, K. Haralambiev, H. Hsiao, and A. Sorniotti, "Policy-based secure deletion," in *Proc. of CCS*, Berlin, Germany, November 2013.

[34] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP' 11)*, Cascais, Portugal, 2011.

[35] A. C.-F. Chan and I. F. Blake, "Scalable, server-passive, user-anonymous timed release cryptography," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005, pp. 504–513.

[36] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of Proof-of-Elapsed-Time (PoET)," in *Stabilization, Safety, and Security of Distributed Systems*, P. Spirakis and P. Tsigas, Eds., 2017, pp. 282–297.

[37] J. H. Cheon, N. Hopper, Y. Kim, and I. Osipkov, "Timed-release and key-insulated public key encryption," in *Financial Cryptography and Data Security*, 2006, pp. 191–205.

[38] B. Cohen and K. Pietrzak, "Simple proofs of sequential work," in *Advances in Cryptology – EUROCRYPT 2018*, 2018, pp. 451–467.

[39] A. Conner-Simons, "Programmers solve MIT's 20-year-old cryptographic puzzle," https://www.csail.mit.edu/news/programmers-solve-mits-20-year-old-cryptographic-puzzle, 2019.

[40] M. Conti, E. S. Kumar, C. Lal, and S. Ruj, "A survey on security and privacy issues of Bitcoin," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3416–3452, 2018.

[41] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential benchmarking based on multiparty computation," in *Financial Cryptography and Data Security*, 2017, pp. 169–187.

[42] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits," in *ESORICS 2013*, 2013, pp. 1–18.

[43] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Over-encryption: Management of access control evolution on outsourced data," in *Proc. of VLDB*, Vienna, Austria, Sept. 2007.

[44] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Encryption policies for regulating access to outsourced data," *ACM TODS*, vol. 35, no. 2, pp. 12:1–12:46, April 2010.

[45] O. E. Dictionary, "Oxford English Dictionary," *Simpson, JA & Weiner, ESC*, 1989.

[46] S. Diesburg and A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computer Surveys*, vol. 43, no. 1, December 2010.

[47] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 211–227.

[48] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology – CRYPTO' 92*, 1993, pp. 139–147.

[49] M. Dworkin, "Recommendation for block cipher modes of operation, methods and techniques," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-38A, 2001. [Online]. Available: http://www.csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[50] H. Feistel, “Cryptography and computer privacy,” *Scientific american*, vol. 228, no. 5, pp. 15–23, 1973.

[51] P. Fletcher-Hill, “Kimono — trustless secret sharing using time-locks on Ethereum,” https://medium.com/@pfh/kimono-trustless-secret-sharing-using-time-locks-on-ethereum-8e7e696494d, 2018.

[52] K. Fu, S. Kamara, and Y. Kohno, “Key Regression: Enabling efficient key distribution for secure distributed storage,” in *Proc. of NDSS*, San Diego, CA, USA, February 2006.

[53] S. Garg, C. Gentry, A. Sahai, and B. Waters, “Witness encryption and its applications,” in *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, ser. STOC ’13, 2013, pp. 467–476.

[54] V. Goyal, O. Pandey, A. Sahai, and B. Waters, “Attribute-based encryption for fine-grained access control of encrypted data,” in *Proc. of CCS*, Alexandria, VA, USA, October-November 2006.

[55] A. Groce, J. Katz, A. Thiruvengadam, and V. Zikas, “Byzantine agreement with a rational adversary,” in *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II*, ser. ICALP’12, 2012, pp. 561–572.

[56] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, “Executing SQL over encrypted data in the database-service-provider model,” in *Proc. of ACM SIGMOD*, Madison, Wisconsin, June 2002.

[57] I. Hang, F. Kerschbaum, and E. Damiani, “ENKI: Access control for encrypted query processing,” in *Proc. of SIGMOD*, Melbourne, Australia, May 2015.

[58] B. Hauser, “Introducing Brownie: A python framework for testing, deploying and interacting with ethereum smart contracts,” https://medium.com/hyperlink-technology/introducing-brownie-a763859409ca, 2019.

[59] C. Hazay and Y. Lindell, “A note on the relation between the definitions of security for semi-honest and malicious adversaries,” 2010.

[60] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure coding in Windows Azure Storage,” in *Proc. of USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, USA, June 2012.

[61] J. Hur and D. Noh, “Attribute-based access control with efficient revocation in data outsourcing systems,” *IEEE TPDS*, vol. 22, no. 7, pp. 1214–1221, July 2011.

[62] “Hyperledger Sawtooth,” https://sawtooth.hyperledger.org, Hyperledger Foundation, 2018.

[63] D. Irvine, “Distributed file system,” MaidSafe, Tech. Rep., 2010.

[64] W. E. Jarvis, *Time capsules: a cultural history*. McFarland, 2015.

[65] N. Jefferies, C. Mitchell, and M. Walker, “A proposed architecture for trusted third party services,” in *Cryptography: Policy and Algorithms*, 1996, pp. 98–104.

[66] K. Kapusta, G. Memmi, and H. Noura, “An efficient keyless fragmentation algorithm for data protection,” *CoRR*, vol. abs/1705.09872, 2017. [Online]. Available: http://arxiv.org/abs/1705.09872

[67] G. O. Karame, C. Soriente, K. Lichota, and S. Capkun, “Securing cloud data under key exposure,” *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 838–849, July 2019.

[68] M. Keller, E. Orsini, and P. Scholl, “MASCOT: Faster malicious arithmetic secure computation with oblivious transfer,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, 2016, pp. 830–842.

[69] M. Keller, V. Pastro, and D. Rotaru, “Overdrive: making SPDZ great again,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2018, pp. 158–189.

[70] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016, pp. 839–858.

[71] N. Lambert and B. Bollen, "The SAFE network - a new, decentralised internet," http://docs.maidsafe.net/Whitepapers/pdf/TheSafeNetwork.pdf, MaidSafe, Tech. Rep., 2014.

[72] K. Leyton-Brown and Y. Shoham, "Essentials of game theory: A concise multidisciplinary introduction," *Synthesis lectures on artificial intelligence and machine learning*, vol. 2, pp. 1–88, 2008.

[73] C. Li and B. Palanisamy, "Timed-release of self-emerging data using distributed hash tables," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2344–2351.

[74] C. Li and B. Palanisamy, "Decentralized release of self-emerging data using smart contracts," in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018, pp. 213–220.

[75] M. Li, C. Qin, and P. P. C. Lee, "CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal," in *Proc. of USENIX ATC*, Santa Clara, CA, USA, July 2015.

[76] M. Li, C. Qin, P. P. C. Lee, and J. Li, "Convergent Dispersal: Toward storage-efficient security in a cloud-of-clouds," in *Proc. of HotStorage*, Philadelphia, PA, USA, June 2014.

[77] J. Liu, T. Jager, S. A. Kakvi, and B. Warinschi, "How to build time-lock encryption," *Designs, Codes and Cryptography*, vol. 86, pp. 2549–2586, 2018.

[78] G. Loukas and G. Öke, "Protection against denial of service attacks: A survey," *The Computer Journal*, vol. 53, pp. 1020–1037, 2010.

[79] M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM J. Comp.*, vol. 17, no. 2, pp. 373–386, Apr. 1988.

[80] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ forward error correction scheme for object delivery – RFC 6330," *IETF Request For Comments*, 2011.

[81] M. Luongo and C. Pon, "The Keep network: A privacy layer for public blockchains," https://keep.network/whitepaper, 2019.

[82] D. MacKay, "Fountain codes," *IEE Proceedings - Communications*, vol. 152, pp. 1062–1068(6), December 2005.

[83] M. Mahmoody, T. Moran, and S. Vadhan, "Time-lock puzzles in the random oracle model," in *Advances in Cryptology – CRYPTO 2011*, 2011, pp. 39–50.

[84] M. Mahmoody, T. Moran, and S. Vadhan, "Publicly verifiable proofs of sequential work," in *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, 2013, pp. 373–388.

[85] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz, "Mitigating distributed denial of service attacks with dynamic resource pricing," in *Seventeenth Annual Computer Security Applications Conference*, 2001, pp. 411–421.

[86] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," *ArXiv*, vol. abs/1711.09327, 2017.

[87] T. May, "Timed-release crypto," http://www.hks.net/cpunks/cpunks-0/1560.html, 1993.

[88] F. Mert Celebi, P. Fletcher-Hill, G. Kaemmer, and D. Que, "Kimono time capsule," https://kimono.network, 2018.

[89] M. Naor and O. Reingold, "On the construction of pseudorandom permutations: Luby—rackoff revisited," *Journal of Cryptology*, vol. 12, no. 1, pp. 29–66, Jan 1999. [Online]. Available: https://doi.org/10.1007/PL00003817

[90] M. Nojoumian, A. Golchubian, L. Njilla, K. Kwiat, and C. Kamhoua, "Incentivizing blockchain miners to avoid dishonest mining strategies by a reputation-based paradigm," in *Intelligent Computing*, 2019, pp. 1118–1134.

[91] D. Nuñez, I. Agudo, and J. Lopez, "Delegated access for Hadoop clusters in the cloud," in *Proc. of IEEE CloudCom*, 2014.

[92] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe, "Practical secrecy-preserving, verifiably correct and trustworthy auctions," *Electronic Commerce Research and Applications*, vol. 7, pp. 294 – 312, 2008.

[93] C. Patterson, "Distributed content delivery and cloud storage," www.smithandcrown.com/distributed-content-delivery-cloud-storage, Smith and Crown, Tech. Rep., 2017.

[94] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD Records*, vol. 17, no. 3, pp. 109–116, Jun. 1988.

[95] G. Paul, F. Hutchison, and J. Irvine, "Security of the MaidSafe Vault Network," in *Wireless World Research Forum Meeting 32*, Marrakesh, Morocco, May 2014.

[96] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin, "Secure deletion for a versioning file system," in *Proc. of FAST*, San Francisco, CA, USA, Dec. 2005.

[97] J. Poon and T. Dryja, "The Bitcoin lightning network: Scalable off-chain instant payments," https://www.bitcoinlightning.com/wp-content/uploads/2018/03/lightning-network-paper.pdf, 2016.

[98] M. O. Rabin, "How to exchange secrets with oblivious transfer," *IACR Cryptology ePrint Archive*, vol. 2005, p. 187, 2005.

[99] M. O. Rabin and C. Thorpe, "Time-lapse cryptography," http://nrs.harvard.edu/urn-3:HUL.InstRepos:26506434, Harvard Computer Science Group, Tech. Rep. TR-22-06, 2006.

[100] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, June 1960.

[101] J. K. Resch and J. S. Plank, “AONT-RS: Blending security and performance in dispersed storage systems,” in *Proc of FAST*, San Jose, CA, USA, February 2011.

[102] R. Rivest, “All-or-Nothing encryption and the package transform,” in *Proceedings of the 4th Internation Workshop on Fast Software Encryption (FSE)*, Haifa, Israel, January 1997.

[103] R. L. Rivest, “Description of the LCS35 time capsule crypto-puzzle,” https://people.csail.mit.edu/rivest/lcs35-puzzle-description, 1999.

[104] R. L. Rivest, A. Shamir, and D. A. Wagner, “Time-lock puzzles and timed-release crypto,” MIT, Tech. Rep., 1996.

[105] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[106] A. Shokrollahi, “Raptor codes,” *IEEE/ACM TON*, vol. 6, no. 3-4, pp. 213–322, May 2011.

[107] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, vol. 2, no. 9, 1997.

[108] “TCG storage security subsystem class: Opal,” www.trustedcomputinggroup.org/wp-content/uploads/TCG_Storage-Opal_SSC_v2.01_rev1.00.pdf, Aug. 2015.

[109] M. Theoharidou, N. Papanikolaou, S. Pearson, and D. Gritzalis, “Privacy risk, security, accountability in the cloud,” in *Proc. of IEEE CloudCom*, Bristol, UK, December 2013.

[110] “Ganache – personal blockchain for ethereum development,” https://github.com/trufflesuite/ganache, Truffle Blockchain Group, 2019.

[111] M. von Maltitz and G. Carle, “A performance and resource consumption assessment of secret sharing based secure multiparty computation,” in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2018, pp. 357–372.

[112] D. Vorick and L. Champine, “Sia: Simple decentralized storage,” https://www.sia.tech/whitepaper.pdf, Nebulous Inc., Tech. Rep., 2014.

[113] M. Waldman and D. Mazieres, “Tangler: a censorship-resistant publishing system based on document entanglements,” in *Proc. of ACM CCS*, Philadelphia, PA, USA, November 2001.

[114] A. Webster and S. E. Tavares, “On the design of S-boxes,” in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1985, pp. 523–534.

[115] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, C. Pollard, and V. Buterin, “Storj: a peer-to-peer cloud storage network (v2.0),” https://storj.io/storj.pdf, Storj Labs Inc., Tech. Rep., 2016.

[116] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[117] A. C. Yao, “Protocols for secure computations,” in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, ser. SFCS ’82, 1982, pp. 160–164.

[118] S. Yu, C. Wang, K. Ren, and W. Lou, “Attribute based data sharing with attribute revocation,” in *Proc. of ASIACCS*, April 2010.

[119] G. Zyskind, O. Nathan, and A. Pentland, “Enigma: Decentralized computation platform with guaranteed privacy,” *arXiv:1506.03471*, 2015.

A

Mix&Slice Implementation

This appendix describes the open-source implementation of the *Mix&Slice* encryption mode described in Chapter 2.

The implementation is done in C and consists of single-threaded and a multi-threaded encryption/decryption functions that make use of AES as base symmetric encryption primitives. The use of OpenSSL EVP APIs leverages hardware-accelerated AES-NI primitives when available.

A.1 APIs

The file `includes/aes_mix.h` contains the following three definitions:

- `BLOCK_SIZE`: number of bytes in a cipher block (16 bytes for AES).
- `MINI_SIZE`: number of bytes in a mini-block.
- `MINI_PER_MACRO`: number of mini-blocks in a macro-block.

These parameters can be modified at compile time to experiment with different sizes.

A.1.1 Single-thread APIs

The file `includes/aes_mix.h` contains the prototype of the only two methods that are necessary to use Mix&Slice:

```
void mixencrypt(
        const unsigned char* data, unsigned char* out,
        const unsigned long size, const unsigned char* key,
        const unsigned char* iv);

void mixdecrypt(
        const unsigned char* data, unsigned char* out,
        const unsigned long size, const unsigned char* key,
        const unsigned char* iv);
```

The parameters are as follows.

- data: pointer to the source buffer (plaintext in case of mixencrypt and ciphertext in case of mixdecrypt).
- out: pointer to the destination buffer.
- size: number of bytes in source (and destination) buffers.
- key: symmetric key (string) used for the AES functions.
- iv: initialization vector for the AES functions.

A.1.2 Multi-thread APIs

The file includes/aes_mix_multi.h contains the prototypes of the only two methods that are necessary to use Mix&Slice in multi-threaded mode:

```
void t_mixencrypt(
        unsigned int thr, const unsigned char* data,
        unsigned char* out, const unsigned long size,
        const unsigned char* key, const unsigned char* iv);

void t_mixdecrypt(
        unsigned int thr, const unsigned char* data,
        unsigned char* out, const unsigned long size,
        const unsigned char* key, const unsigned char* iv);
```

The only additional parameter is thr, the number of threads to use.

A.1.3 Slicing phase

The mixing phase is the real encryption phase. The slicing phase strongly depends on the file management and should be implemented according to the ratio of policy updates with respect to decryption processes and can be easily sped up with ad-hoc file management. Because of this, the performance of the mixing phase is a good proxy of the performance of the whole Mix&Slice technique.

The version implemented here keeps the fragments together. This benefits the policy update process, whereas the decryption process has to pay the overhead for rearranging the bytes before performing the unmixing phase.

The file `includes/aes_mixslice.h` contains the prototypes of the two methods that perform the whole Mix&Slice encryption:

```
void mixslice(
        unsigned int thr, const unsigned char* data,
        unsigned char* fragdata, const unsigned long size,
        const unsigned char* key, const unsigned char* iv);

void unsliceunmix(
        unsigned int thr, const unsigned char* fragdata,
        unsigned char* out, const unsigned long size,
        const unsigned char* key, const unsigned char* iv);
```

The `mixslice` method first uses `t_mixencrypt` to perform the mixing phase. The slicing phase rearranges the output of the mixing phase in slices. The user is responsible for creating the buffer that will contain the `fragdata`. The slices are concatenated and written to the `fragdata` buffer. The user of the function can read the fragments directly from there as follows:

- each fragment consists of `fragsize = size / MINI_PER_MACRO` bytes;
- the first fragment spans the `fragdata` bytes in range `[0, fragsize)`;
- the second fragment spans the `fragdata` bytes in range `[fragsize, fragsize*2)`;

- and so on until `[size - fragsize, size)`.

A.1.4 Installation

Before proceeding please install the `openssl/crypto` library source and the `libtool` binary. In *ubuntu*, you can proceed as follows:

```
$ sudo apt install libtool-bin libssl-dev
```

To compile and install the dynamic library in your system you can:

```
$ make
$ sudo make install
```

To remove the library simply do:

```
$ sudo make uninstall
```

A.1.5 Test

There are three test suites:

- *main*: main test suite that verifies that Mix&Slice principles are enforced.
- *blackbox*: test suite that verifies the Mix&Slice principles in an *abstract* sense (without knowledge about the code).
- *multithread*: test suite that verifies that the Mix&Slice principles are enforced in the multi-threaded implementation.

make is used for compilation and testing purposes. A basic *compile-and-test* setup is made by the steps:

```
$ make
$ make test
```

See the `Makefile` for all the compile and test targets.

A.2 Python Wrapper

The python implementation wraps both the phases and offers a CLI tool that wraps the `libaesmix` library.

The C implementation has been built with performance in mind, whereas the python wrapper and the CLI tool has been implemented to offer widespread access to the Mix&Slice capabilities. The mixing and slicing phases use the C implementation, but the python conversion adds a big overhead since it has to materialize all the buffers in memory.

Since the tool materializes all the buffers in memory and has to perform both the mixing and the slicing phases, you should only use the CLI tool on files that are at maximum as large as a third of your available memory.

Please check the file `example.py` to understand how to use the library.

A.2.1 Requirements

Before proceeding please install the `openssl/crypto` library source. In ubuntu, you can proceed as follows:

```
$ sudo apt install libssl-dev
```

A.2.2 Installation

The package has been uploaded to PyPI at `https://pypi.org/project/aesmix` so, after installing the requirements, you can install the latest released version using pip:

```
$ pip install aesmix
```

To install the version from this repository, you can use the commands:

```
$ make build
$ sudo make install
```

To install the package in a virtual environment, use:

```
python setup.py install
```

The python wrapper will also compile the `libaesmix` library.

A.2.3 Command Line Interface

This package also installs the `mixslice` tool that can be used as follows.

To encrypt a file:

```
$ mixslice encrypt sample.txt

INFO: [*] Encrypting file sample.txt ...
INFO: Output fragdir:   sample.txt.enc
INFO: Public key file:  sample.txt.public
INFO: Private key file: sample.txt.private
```

To perform a policy update:

```
$ mixslice update sample.txt.enc

INFO: [*] Performing policy update on sample.txt.enc ...
INFO: Encrypting fragment #68
INFO: Done
```

To decrypt a file:

```
$ mixslice decrypt sample.txt.enc

INFO: [*] Decrypting fragdir sample.txt.enc
          using key sample.txt.public ...
INFO: Decrypting fragment #68
INFO: Decrypted file: sample.txt.enc.dec

$ sha1sum sample.txt sample.txt.enc.dec

d3e92d3c3bf278e533f75818ee94d472347fa32a  sample.txt
d3e92d3c3bf278e533f75818ee94d472347fa32a  sample.txt.enc.dec
```

A.3 Key regression mechanism

The key regression implementation is based on “Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage” [52].

Example

The key regression mechanism can be used as follows.

```
from aesmix.keyreg import KeyRegRSA

iters = 5
stp = KeyRegRSA()

print("== WINDING ==")
for i in range(iters):
    stp, stm = stp.wind()
    print("k%i: %r" % (i, stm.keyder()))

print("\n== UNWINDING ==")
for i in range(iters - 1, -1, -1):
    print("k%i: %r" % (i, stm.keyder()))
    stm = stm.unwind()
```

B

Securing Resources in Decentralized Cloud Storage — Proofs of Theorems

Proof of Theorem 1

Let us assume, by contradiction, the existence of a (k, r)-allocation for a resource split into $s = k < k + 1$ slices. Given s different slices, no more than s nodes can be used to store one replica of the slices. Since s=k, the allocation function is storing the whole resource using at most k nodes. Therefore, it is not k-protected. Note that $k + 1$ slices are sufficient to define a (k, r)-allocation. The r-replication requirement is easily satisfied by replicating r times each of the $k + 1$ slices. The k-protection requirement is satisfied by storing each slice to a different node. Hence, for each replica of the resource, each coalition of k nodes misses one slice (the one stored at the $(k + 1)$-th node). □

Proof of Theorem 2

Since there are $r \cdot (k + 1)$ slices to be stored, a (k, r)-allocation cannot use more than $r \cdot (k+1)$ nodes. However, a (k, r)-allocation with $s = k+1$ cannot use less than $r \cdot (k + 1)$ nodes. Let us consider the case where slices are not *replicated (i.e., $r = 1$), since the same discussion applies to each replica of the resource. Assume, by contradiction, that a (k, r)-allocation stores more than one slice at one of the nodes. If the function adopts $n = k+1$ nodes, there will be at least one node that does not store any slice (which is equivalent to say that $n \leq k$) as the number of slices is $k + 1$. Then, k nodes store all the slices composing the resource, thus violating k-protection.* □

Proof of Theorem 3

To guarantee r-replication, each slice should be stored at (at least) r nodes. If allocation function φ is k-protected, for each coalition $\mathtt{N_i}$ of k nodes, there exists at least a slice $\mathtt{s_j}$ that is not stored at any of the nodes in $\mathtt{N_i}$. To guarantee that $\mathtt{s_j}$ has r copies, there must exist at least r additional nodes that store $\mathtt{s_j}$, and n should be at least equal to $k + r$. Note that $k + r$ nodes are sufficient to define a (k, r)-allocation. Consider, as an example, $s = \binom{k+r}{k}$ slices, the set $\mathtt{N_1}, \ldots, \mathtt{N_s}$ of possible coalitions of k nodes, and allocation function φ that assigns the i-th slice to all the nodes in $\mathcal{N}$, but the ones in the i-th coalition: $\varphi(\mathtt{s_i}) = \mathcal{N} \setminus \{\mathtt{N_i}\}$. Function φ is k-protected, since each coalition $\mathtt{N_i}$ cannot access slice $\mathtt{s_i}$, and r-replicated, since slice $\mathtt{s_i}$ is stored at each node $\mathtt{n_i} \in \mathcal{N} \setminus \{\mathtt{N_i}\}$, then at $n - k = r$ nodes. □

Proof of Theorem 4

An allocation function φ is k-protected if the set of slices stored at any coalition $\mathtt{N_i}$ of k nodes is not complete (i.e., at least one slice is missing). To guarantee that φ is an allocation function each slice should be stored at least on one node. If each coalition misses a different slice, we are minimizing the number of slices necessary to define an allocation function φ. Indeed, since $n > k$, the missing slice for each coalition $\mathtt{N_i}$ can be stored at the nodes in $\mathcal{N} \setminus \mathtt{N_i}$, as they will miss another slice. Since there are $\binom{k+r}{k}$ possible coalitions of k nodes in $\mathcal{N}$, s should be at least $\binom{k+r}{k}$ to guarantee that each coalition misses a different slice. Assume, by contradiction, that $r = 1$, $s = \binom{k+r}{k} - 1$, and that φ is k-protected. In this case, two coalitions $\mathtt{N_i}$ and $\mathtt{N_j}$ will miss the same slice $\mathtt{s_x}$. That is, there are at least $k + 1$ nodes (the ones *in $\mathtt{N_i} \cup \mathtt{N_j}$) missing $\mathtt{s_x}$. However, if $n = k + 1$ then φ cannot be an allocation function, since no node in $\mathcal{N}$ stores $\mathtt{s_x}$. (The same reasoning applies with larger values for r.)* □

Proof of Theorem 5

1) *Since each coalition of k nodes should not be able to reconstruct the resource, it should miss at least one slice. The number of slices used by the allocation function is $s = \binom{k+r}{k}$, which is sufficient for each coalition to miss at least one slice. In fact, the number of possible coalitions of k nodes is $\binom{k+r}{k}$. Let us assume, by contradiction, that two coalitions $\mathtt{N_i}$ and $\mathtt{N_j}$ miss the same slice $\mathtt{s_x}$. Therefore, there are $k + 1$ nodes ($\mathtt{N_i} \cup \mathtt{N_j}$) that do not store $\mathtt{s_x}$. However, in this case $\mathtt{s_x}$ would be stored at $n-(k+1) = r-1$ nodes. Hence, the allocation function would not be r-replicated.*

2) *By definition of (k, r)-allocation with $n = k + r$ nodes and $s = \binom{n}{k}$ slices, each coalition of k nodes misses a different slice. Let us consider two coalitions $\mathtt{N_i}$ and $\mathtt{N_j}$ that differ in one node only. Coalition $\mathtt{N_j}$ misses one slice, $\mathtt{s_j}$, while $\mathtt{N_i}$ misses $\mathtt{s_i}$. Since $\mathtt{N_j}$ misses one slice only, it stores $\mathtt{s_i}$. Hence, $\mathtt{N_i} \cup \mathtt{N_j}$ includes $k + 1$ nodes and stores all the slices composing the resource.* □

Proof of Theorem 6

P_u) *The probability to obtain back the original plaintext resource corresponds to the probability that $k + 1$ nodes, each storing a different slice, do not fail. Since p_u is the probability that a node fails, the probability that at least one of the r replicas of a slice is available is $(1 - (p_u)^r)$, with $(p_u)^r$ the probability that all r replicas of the slice are unavailable. The probability to obtain back all the $k + 1$ slices, and then also the original plaintext resource, is then $(1 - (p_u)^r)^{k+1}$ and $(1 - (1 - (p_u)^r)^{k+1})$ is the probability that the resource cannot be decrypted.*

P_c) *The probability that the resource is exposed is the probability that all the slices are compromised, meaning that $k + 1$ nodes are malicious. Since $1 - p_c$ is the probability that a node is not compromised and each slice has r replicas, the probability that at least one replica is exposed is $(1 - (1 - p_c)^r)$,* *with* $(1-p_c)^r$ *the probability that all* r *replicas of a slice are not compromised. We can then conclude that the probability that all* $k+1$ *slices are exposed is* $(1-(1-p_c)^r)^{k+1}$. □

Proof of Theorem 7

P_u*) To obtain back the original plaintext resource, we need the slices stored on any combination of* $k+1$ *nodes, that is,* $k+1$ *nodes must not fail. A resource therefore becomes unavailable when any combination of* r *or more nodes fail. The probability that* i *nodes fail, with* $i = r, \ldots, k+r$*, and* $k+r-i$ *nodes do not fail is equal to* $(p_u)^i(1-p_u)^{k+r-i}$*. Since the number of combinations of* i *nodes out of* $k+r$ *is* $\binom{k+r}{i}$*, the probability that a resource is unavailable is* $P_u = \sum\limits_{i=r}^{k+r} \binom{k+r}{i}(p_u)^i(1-p_u)^{k+r-i}$.

P_c*) A coalition can compromise the confidentiality of the resource whenever it involves any combination of* $k+1$ *nodes, that is, at least* $k+1$ *nodes must be compromised. The probability that* i *nodes are compromised, with* $i = k+1, \ldots, k+r$*, and* $k+r-i$ *nodes are not compromised is equal to* $(p_c)^i(1-p_c)^{k+r-i}$*. Since the number of combinations of* i *nodes out of* $k+r$ *is the binomial coefficient* $\binom{k+r}{i}$*, we can conclude that the probability that any combination of at least* $k+1$ *nodes are compromised in a collection of* $r+k$ *nodes is* $P_c = \sum\limits_{i=k+1}^{k+r} \binom{k+r}{i}(p_c)^i(1-p_c)^{k+r-i}$. □