Mix&Slice for Efficient Access Revocation on Outsourced Data

Enrico Bacis, Sabrina De Capitani di Vimercati, *Senior Member, IEEE,* Sara Foresti, *Senior Member, IEEE,* Stefano Paraboschi, *Member, IEEE,* Marco Rosa, Pierangela Samarati, *Fellow, IEEE* .

Abstract—A complex problem when outsourcing data to the cloud is access control management. Encryption, by wrapping data with a self-enforcing protection layer, provides access control enforcement by making resources intelligible only to users holding the necessary key. The real challenge becomes then the efficient revocation of access. We address this challenge and present an approach to effectively and efficiently 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. Our experimental results show the effectiveness of our approach, and confirm its efficiency, especially when managing large resources with dynamic access policy.

Index Terms—Access control; Access revocation; Resource encryption; Mix&Slice

1 Introduction

With the considerable advancements in Information and Communication Technologies 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. While trustworthy for providing storage and access functionality, the provider cannot typically be considered authorized to know the content of the resources it stores (*honest-but-curious* scenario [2], [3], [4]) and hence neither 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.

- *Enrico Bacis is with Google.*
  E-mail: enricobacis@google.com
- *Sabrina De Capitani di Vimercati, Sara Foresti, and Pierangela Samarati are with the Università degli Studi di Milano, Italy.*
  E-mail: firstname.lastname*@unimi.it*
- *Marco Rosa is with SAP Security Research, France.*
  E-mail: marco.rosa@sap.com
- *Stefano Paraboschi is with the Università degli Studi di Bergamo, Italy.*
  E-mail: parabosc@unibg.it

A preliminary version of this paper appeared under the title "Mix&slice: Efficient access revocation in the cloud," in Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS 2016) *[1]*

One of the complex aspects in using encryption to enforce access control concerns access revocation. If granting an authorization is easy (it is sufficient to give the newly authorized user access to the key for decrypting the resource), 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 supported by current technologies, 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 paper, 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, meaning that 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 classical *all-or-nothing transform* (AONT) [5] considers similar requirements, but the techniques proposed for it are not suited to our scenario. AONT approaches 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 (to enable authorized retrieval of portions of the resource). 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 a carefully designed bit-mixing approach that ensures, at the end of the process, that every individual bit in the input has had an impact on each of the bits in the encrypted output.
- *Slice:* the mixed macro-blocks are sliced into fragments so that each fragment includes bits from each macro-block of the resource. Fragments represent 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 service provider simply needs to provide storage functionality and is not required to play an active role in 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.

A preliminary version of this work appeared in [1]. In this paper, we extend our proposal with a more general approach to mixing by considering also the application of the Optimal Asymmetric Encryption Padding (OAEP), discussing two strategies for such a solution. We also extend the analysis of the effectiveness of our approach to cover erasure code attacks. Finally, we extend the experimental evaluation, where we evaluate the performance of our approach in terms of the throughput at the client-side for the application of our protection technique, and the efficiency in policy revocation.

Outline. The remainder of the paper is organized as follows. Section 2 illustrates the basic concepts of our approach, the working of mixing and slicing, and the properties they need to satisfy to enable effective revocation. Section 3 illustrates two main approaches to realize mixing, based on the iterative application of AES and on an extended 3-round OAEP, respectively. Section 4 illustrates the enforcement of access revocation. Section 5 discusses the effectiveness of our solution in providing revocation, considering its resilience against storage attacks by users who might maintain some local storage of previously accessed fragments or an erasure code on the resource. Section 6 illustrates our implementation and experimental evaluation, to measure the throughput of mixing as well as of access and update requests, confirming the advantages and applicability of our approach. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

2 Mix & Slice

We introduce the concepts of block, mini-block, and macro-block (Section 2.1) at the basis of our approach, and then illustrate the working of mixing (Section 2.2) and slicing (Section 2.3).

2.1 Blocks, mini-blocks, and macro-blocks

The basic building block of our approach is the application of a transformation that maps input data onto output data in a way that all bits of the output depend on all bits in the input. Such a transformation can be realized with either a cryptographic hash function or a symmetric block cipher. A cryptographic hash function is a non invertible function that transforms an input into an output such that every bit of the input has effect on every bit of the output. A symmetric block cipher 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 [6].

Clearly, the larger the number of bits that are missing in the encrypted version of a block, the larger the effort needed to perform a brute-force attack, which requires attempting 2^x possible combinations of values when x bits are missing. Such number of missing bits is the *security parameter* at the center of our approach and represents the atomic unit of protection. We then explicitly identify the following basic concepts.

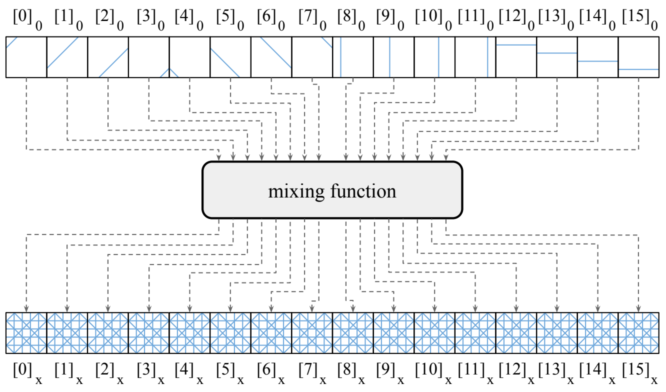


Fig. 1. An example of mixing of a macro-block with 16 mini-blocks $[0]\dots[15]$

- *Block*: a sequence of bits input to a block cipher or a cryptographic function (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 a block cipher on sequences of bits larger than individual blocks. In particular, our approach operates by *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. In the following, for concreteness and simplicity of the figures, we will illustrate our examples assuming blocks of 128 bits and mini-blocks of 32 bits, which corresponds to having 4 mini-blocks in every block. In the following, we will use *msize*, *bsize*, *Msize* to denote the size (in bits) of mini-blocks, blocks, 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 mixing process, a subscript associated with a mini-block/block denotes the round that produced it.

2.2 Mixing

Our mixing has the objective of producing an encrypted version of a resource in a way that each bit in the encrypted representation depends on every bit of the plaintext resource. As already noted, a block cipher provides mixing only at the level of block. For instance, given a sequence of 16 mini-blocks ($[0],\dots,[15]$), the application of a block cipher working on blocks of 4 mini-blocks each provides mixing among mini-blocks $[0]\dots[3]$, $[4]\dots[7]$, $[8]\dots[11]$, and $[12]\dots[15]$, 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 $[0]$ in the result of the block cipher applied over the first block $[0]\dots[3]$, will prevent reconstruction of such first block but will not prevent reconstruction of the other three blocks (mini-blocks $[4],\dots,[15]$). Protection at the block level is clearly not sufficient in our context, where we expect to manage resources of arbitrarily large size and aim to provide the guarantee that the lack of any individual mini-block in the encrypted representation of a resource implies the impossibility (apart from performing a brute-force attack) of reconstructing any other mini-block of the corresponding plaintext resource. The concept of macro-block allows us to provide mixing on an arbitrarily long sequence of bits (going much above the size of the block). An interpretation of mixing is that it extends the ability of protecting the correspondence between input and output of a block cipher to blocks of arbitrary size.

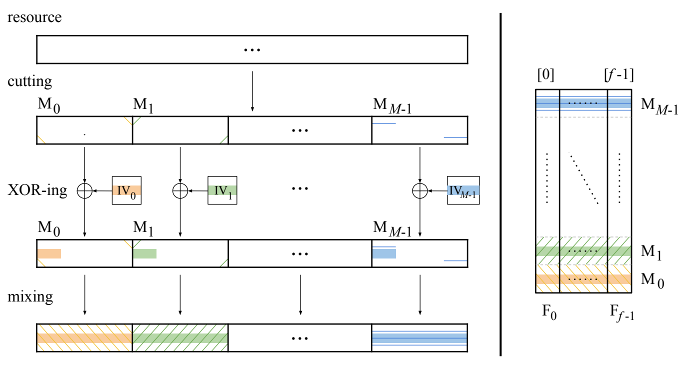
Figure 1 illustrates our mixing applied to a macro-block composed of 16 mini-blocks $[0]\dots[15]$. The pattern-coding in the figure shows that the 16 output mini-blocks depend on each of the 16 input mini-blocks.

To provide an effective and robust support to the enforcement of access revocation, *mixing* must satisfy the following properties.

- *Complete mixing:* every bit in the input macro-block of the mixing must cryptographically affect every bit of the output macro-block.
- *Arbitrary macro-block size:* mixing should operate on macro-blocks of arbitrarily large size.
- *No-shrinking effect:* the output of mixing, as well any of its intermediate results or observable state must not be smaller than the size of the input macro-block.

Complete mixing guarantees the complete dependency of each output bit from each input bit, and hence the impossibility of reconstructing the plaintext macro-block, or parts of it, when even a small portion (mini-block) of its mixed version is missing. As previously discussed, in this case the only possibility is the application of a brute-force attack. Since such a property is guaranteed at the level of macro-block, the second property requires no limitation on the size of the macro-block (e.g., trivially a solution operating complete mixing but with a macro-block of the size of a block would not be acceptable). This property permits the management of an efficient access revocation on resources of arbitrarily large size (Section 6). The last property imposes the size of the output and intermediate results of the process to be not smaller than the size of the input. The motivation for this is that an output or an intermediate result of smaller size would be advantageous to users, who could be able to store such compact information to reconstruct (part of) the input macro-block of a resource to which they have been revoked access. Ensuring the size of intermediate results does not decrease at any step of the process counteracts this threat (Section 5).

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. Indeed, mixing the whole resource as a single macro-block



(a) mix (b) slice

Fig. 2. Mix&Slice: from resource to fragments

Mix&Slice

```
1: cut R in $M$ macro-blocks $M_0, \ldots, M_{M-1}$
2: apply padding to the last macro-block $M_{M-1}$
3: $IV$ := randomly choose an initialization vector
4: for $i = 0, \ldots, M-1$ do                                   /* encrypt macro-blocks */
5:     $M_i[[1]] := M_i[[1]] \oplus IV$                       /* XOR the first block with the IV */
6:     Mix($M_i$)                                  /* one of: AES-Mix, OAEP-Mix, ROAEP-Mix */
7:     $IV := IV + 1$                            /* initialization vector for the next macro-block */
8:     for $j = 0, \ldots, f-1$ do                                         /* slicing */
9:         $F_j[i] := M_i[j]$
```

Fig. 3. Mix&Slice: from resource R to fragments

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 (i.e., enabling authorized access of portions of the resource). This requirement can be 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 6).

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 XOR-ed with an *initialization vector* (*IV*) before starting the mixing process. Since mixing 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 encryption mode [7]. Figure 2(a) illustrates such process.

2.3 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 reconstruction of the resource). Resource protection can be achieved by removing a mini-block for each macro-block of which the resource is composed. This observation brings us to the second concept giving the name to our approach, which is *slicing*. Slicing the encrypted resource consists in defining different *fragments* such that: *i)* every fragment contains a mini-block for each macro-block of the resource, *ii)* no two fragments contain the same mini-block, and *iii)* 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. Figure 2(b) illustrates the slicing process. Slicing and fragments are defined as follows.

Definition 2.1 (Slicing and fragments). Let R be a resource and $M_0, \ldots, M_{M-1}$ be its (individually mixed) macro-blocks, each composed of f mini-blocks. Slicing produces f fragments for R where $F_i = \langle M_0[i], \ldots, M_{M-1}[i] \rangle$, with $i = 0, \ldots, f-1$.

Figure 3 illustrates the Mix&Slice procedure for encrypting a resource R. R is first cut into M macro-blocks, a padding is then applied to the last macro-block, 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 (Section 3). Encrypted macro-blocks are finally sliced into fragments. In the next section, we elaborate on the mixing step (line 6) of such a process.

3 Mixing

We present two strategies to produce a mixing that satisfies the properties discussed in Section 2.2, that is: complete mixing, support for macro-blocks of arbitrary size, and no-shrinking effect. The first strategy (Section 3.1) leverages the AES block cipher and aims at providing efficient mixing of large macro-blocks, also thanks to the wide availability of the hardware implementation of AES. The second strategy (Section 3.2) is based on OAEP, to the aim of supporting large mini-blocks (beyond the size that the more efficient AES-based approach can support), hence increasing the size of the atomic unit of protection provided by mixing.

3.1 AES Mixing

Our first proposal for enforcing complete mixing of the bits in a macro-block extends the mixing provided at the

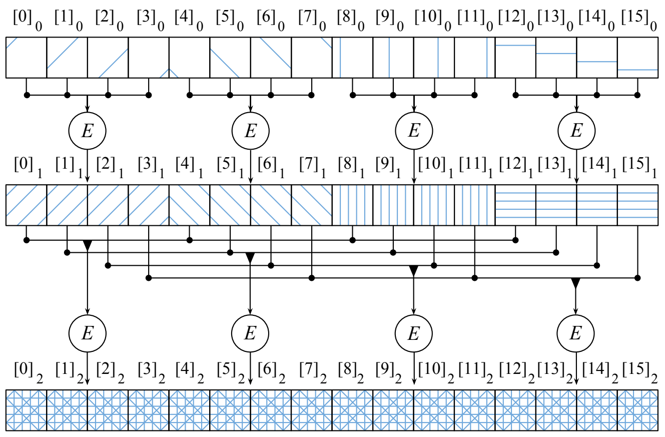


Fig. 4. An example of AES mixing (encrypt mode) of 16 mini-blocks assuming $m = 4$

block-level by AES to the macro-block granularity. Our strategy provides mixing by iteratively applying AES on different portions of the macro-block to guarantee complete dependency of the bits of the output from all the bits in the input. The basic step of our approach (on which we will iteratively build to provide complete mixing within a macro-block) is then the application of encryption at the block level. The idea is to iteratively apply 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 4, where the first row reports a sequence of 16 mini-blocks ($[0], \ldots, [15]$) composing 4 blocks (i.e., each block is composed of m=4 mini-blocks) and the second row reports the mini-blocks ($[0]_1, \ldots, [15]_1$) resulting from the first round. As visible from the pattern-coding in the figure, encryption provides mixing within each block so that each mini-block in the result depends on every mini-block in the same input block. In other words, each $[i]_1$ depends on every $[j]_0$ with (i div 4) = (j div 4). The second round applies 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 (i.e., the number m of mini-blocks in each block) in the sequence, hence considering as input a mini-block from each of all the different encryption operations in the previous round. The blocks considered for encryption are then $\langle[0]_1[4]_1[8]_1[12]_1\rangle$, $\langle[1]_1[5]_1[9]_1[13]_1\rangle$,$\langle[2]_1[6]_1[10]_1[14]_1\rangle$,$\langle[3]_1[7]_1[11]_1[15]_1\rangle$. The result is 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. 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 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). Providing mixing of a macro-block

```
AES-Mix(M)
1: for $i := 1, \ldots, x$ do                                      /* at each round $i$ */
2:   $span := m^i$                                   /* number of mini-blocks in a mixing */
3:   $distance := m^{i-1}$                /* leg of mini-blocks input to an encryption */
4:   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 $distance$ */
5:     let $block$ be the concatenation of all mini-blocks $[l]$
6:        s.t. ($l$ mod $distance$) = $j$ and ($j \cdot m$) div $span$ = $l$ div $span$
7:     $[[j]]_i := \mathrm{E}(k, block)$       /* write the result as the $j$-th block in output */
```

Fig. 5. AES mixing of a macro-block M

composed of b blocks with b=m^{x-1} requires x rounds of encryption each composed of b encryptions. Each round 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}).

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 4). 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. Figure 5 illustrates the mixing procedure. To illustrate, consider the example in Figure 4, 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, 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. 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 6 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. Note that, while for simplicity the figure pictures only propagation of the content of two mini-blocks, every mini-block actually carries along the content of all the mini-blocks with which

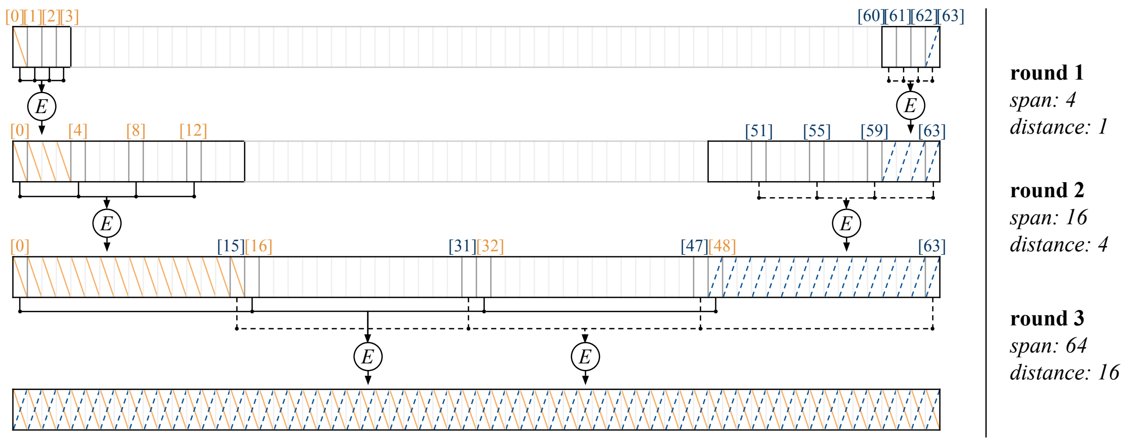


Fig. 6. Propagation of the content of mini-blocks [0] and [63] in the AES mix process

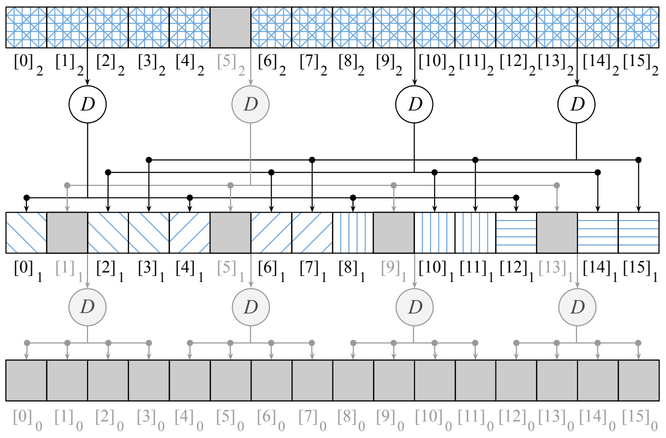


Fig. 7. An example of AES mixing (decrypt mode) of 16 mini-blocks assuming m=4 and the absence of mini-block $[5]_2$

it mixed in previous rounds. Given a macro-block 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 $\log_m(m{\cdot}b)$.

The AES mixing illustrated above satisfies all the properties discussed in Section 2.2. In particular, *complete mixing* is guaranteed since 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 with $\log_m(m{\cdot}b)$ rounds, that is, the rounds requested by our technique, a complete mixing of the macro-block is achieved. Consequently, the absence of a mini-block from the resulting encrypted macro-block will prevent reconstruction of the whole plaintext macro-block. As an example, consider Figure 7, where the first row reports the 16 encrypted mini-blocks obtained from the mixing in Figure 4, and suppose that mini-block $[5]_2$ (i.e., the mini-block represented with a uniform gray background) is missing. The figure shows that the absence of this mini-block prevents the decryption of block $\langle [4]_2[5]_2[6]_2[7]_2 \rangle$, which in turns prevents the reconstruction of block $\langle [1]_1[5]_1[9]_1[13]_1 \rangle$, which finally prevents the reconstruction of all the plaintext blocks (i.e., all mini-blocks $[0]_0, \ldots, [15]_0$). Such complete mixing clearly applies to macro-blocks of *arbitrary size* (the condition that the number mini-blocks be a power of the number m of mini-blocks in a block can be easily achieved through padding). Finally, AES mixing also guarantees *no-shrinking effect*, in fact the representation after each round has the same size as the original macro-block. Users aiming at reconstructing a resource they cannot access would then have no benefit in attacking one round compared to another (see Section 5).

3.2 OAEP Mixing

Our second proposal for enforcing complete mixing of a macro-block is based on an extended 3-round version of the Optimal Asymmetric Encryption Padding (OAEP) [8]. OAEP is a standard padding for RSA that operates on plaintext data and adds randomness to encryption, thus providing higher security guarantees. Figure 8 illustrates the structure that characterizes the classical OAEP. Given two cryptographic hash functions G and H (which can also be identical), and two plaintext input L and R (i.e., a padded message and a random seed), OAEP computes its output as $(G(\mathrm{R}) \oplus \mathrm{L}) \| H((G(\mathrm{R}) \oplus \mathrm{L}) \oplus \mathrm{R})$, where $\oplus$ is the XOR operator and $\|$ is the concatenation of two bit-strings. Although OAEP can be used to build an All-Or-Nothing Transform resistant to chosen-plaintext attacks [9], this basic OAEP structure with 2 rounds does not guarantee every bit of the input to cryptographically affect each bit of the resulting output (i.e., it does not guarantee complete mixing as discussed in Section 2.2). For instance, as visible in Figure 8, the change of a bit in L (i.e., the left input in the figure) impacts all the bits in the right part of the OAEP output, but it only impacts the bit in the same position in the left part of the OAEP output, while not affecting the other bits in the left part. To perform complete mixing, we apply an extended 3-round OAEP as illustrated in Figure 9, where *mix* denotes

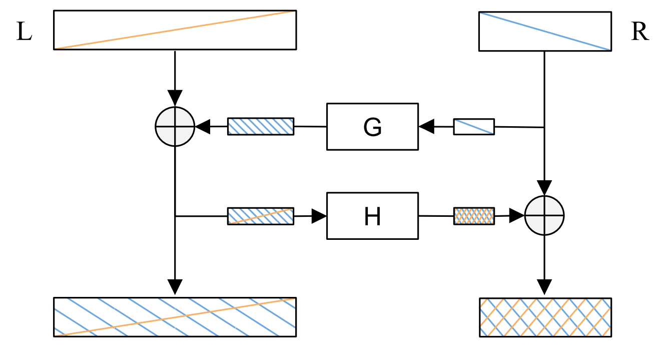


Fig. 8. Classical OAEP structure

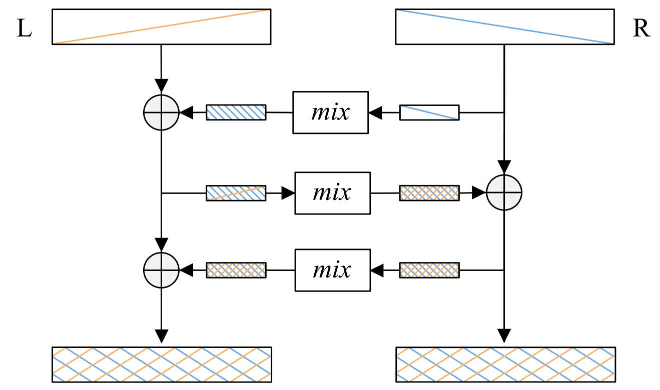


Fig. 9. OAEP with complete mixing

the transformation (cryptographic hash functions G and H in the original OAEP), on which we elaborate next, which is applied at each round. Indeed, 3 rounds are necessary, but also sufficient, to create a pseudorandom permutation, ensuring complete mixing.[1]

With 3-round OAEP ensuring *complete mixing*, we need then to consider the other two properties that must be ensured by mixing, that is, support for macro-blocks of arbitrary size and *no-shrinking effect*. Before discussing the mixing function to guarantee these properties we make a note on the input and final step to produce output for the application of OAEP mixing. As for input, plaintext L and R to be considered as input are obtained by splitting the plaintext input in two parts of the same size, so to ensure the no-shrinking effect and maximize protection guarantees. As for output, we note that being a padding scheme based on cryptographic hash functions, OAEP does not provide data confidentiality (which is instead provided by AES-Mix already during the mixing process). Therefore, the OAEP output must be encrypted (e.g., for this last step our implementation uses AES in counter mode [11]). As for the mixing function, since the functions (i.e., G and H) at the different rounds can be identical, we simply assume to use the same function (*mix* in Figure 9) in all the three rounds of our OAEP mixing.

An easy way to use a 3-round OAEP mixing that provides complete mixing while ensuring no-shrinking effect

1. A 4-round OAEP produces a super pseudorandom permutation [10] resistant to an adversary with oracle access to its inverse permutation. This is not needed to create an AONT and would impact the efficiency of mixing.

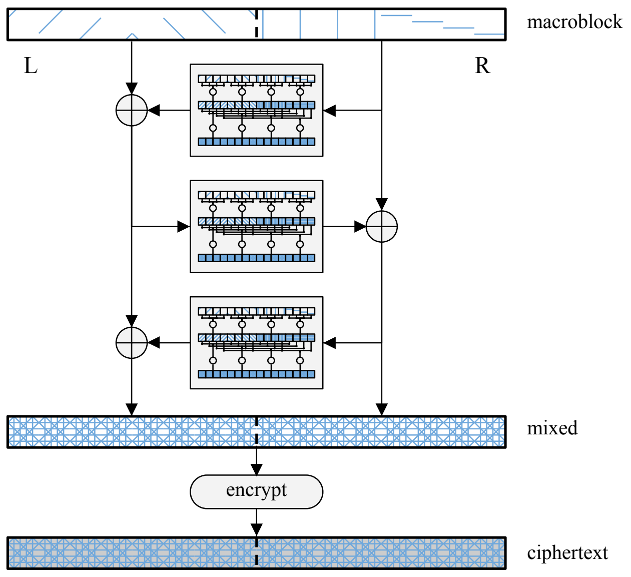


Fig. 10. OAEP mixing with internal layered structure

would be to use, as *mix* function, a cryptographic hash function with an input of size equal to the size of the output. In fact, this constraint would maintain the size of the input and output constant throughout all the process. The adoption of a cryptographic hash function to implement our *mix* function, however, limits the size of the input macro-block to be (at most) twice the size of the output of the selected cryptographic hash function, hence not supporting macro-blocks of *arbitrary size*. The support of macro-blocks of arbitrarily large size could be achieved by using a stream cipher with initial seeds as *mix* function. In fact, the stream cipher can produce any required number of bits needed for the XOR operation with the half of a macro-block, meaning that we can accommodate input and output of arbitrary size. However, the seeds used in the computation are compact in size, and hence their use would violate the *no-shrinking effect* property that the mix transformation needs to ensure. In the remainder of this section, we illustrate two possible approaches for the *mix* function that enforce complete mixing, while guaranteeing both support for macro-blocks of arbitrary size and no-shrinking effect.

OAEP-Mix. Our first approach uses our AES-Mix structure discussed in Section 3.1 as *mix* function, with the only difference of using a cryptographic hash function instead of AES for mixing mini-blocks coming from different blocks at each round (i.e., the circled E in Figure 4 and line 7 in Figure 5). The resulting process is illustrated in Figure 10, where the *mix* function of Figure 9 has been instantiated to be the layered process in Figure 4, and the last step of encryption has been included. The layered structure of AES-Mix guarantees complete mixing, supports macro-blocks of arbitrary size, and satisfies the no-shrinking effect. Figure 11 shows the pseudo-code of OAEP-Mix. Note that, while in the pseudo-code, for simplicity we have maintained the internal call to AES-Mix (Figure 5), as noted the last step at each round of AES-mix is in this case the application of a cryptographic hash function, instead of AES encryption. The reason for this is that the mix function needs not be invertible, since in

```
OAEP-Mix(M)
1: Let M=LR s.t. size(L)=size(R)          /* left (L) and right (R) half of M */
2: $L_1 := L \oplus$ AES-Mix(R)                              /* first round */
3: $R_1 :=$ AES-Mix($L_1$) $\oplus$ R                          /* second round */
4: $L_2 := L_1 \oplus$ AES-Mix($R_1$)                          /* third round */
5: $M := L_2 || R_1$                                   /* mixed macroblock */
6: $E(k, M)$                             /* mixed and encrypted macroblock */
```

Fig. 11. OAEP mixing of a macro-block M

the OAEP approach encryption is applied at the end of the mixing process. Hence, there is no need to encrypt at each step as AES does and the application of a cryptographic hash function (e.g., BLAKE2 [12] in our implementation) in a way that guarantees no-shrinking effect is sufficient. The advantage of using a cryptographic hash function instead of AES is efficiency of the computation and the possibility of using larger blocks and setting larger values for the mini-block size (e.g., 512 bits for *bsize* and up to 256 bits for *msize* when using BLAKE2, in contrast to 128 bits for *bsize* and up to 64 bits for *msize* when using AES), thus possibly reducing the number of rounds needed for performing a complete mix of the input. By using the layered structure of AES-Mix as *mix* function, the input macro-block must include $b = 2 \cdot (bsize/msize)^{x-1}$ blocks, with *bsize* the size of the output of the adopted cryptographic hash function, thus requiring x rounds for the internal AES-Mix. For instance, assuming a macro-block of size *Msize*=16.384 bits, blocks of size *bsize*=512 bits, and mini-blocks of size *msize*=32 bits, the mix function based on BLAKE2 needs 2 rounds only (i.e., $b = 2 \cdot 16^1$) in contrast to 4 rounds necessary when adopting AES with then *bsize*=128 bits (i.e., $b = 2 \cdot 4^3$).

ROAEP-Mix. Our second approach uses the 3-round OAEP structure as *mix* function, thus building a recursive OAEP structure. This is illustrated in Figure 12, where the *mix* function of Figure 9 has been instantiated to be again the process in Figure 9 itself, and as before the last step of encryption has been included. Figure 13 shows the pseudo-code of ROAEP-Mix. By using recursive OAEP mixing, the input macro-block must include $b = 2^x = (Msize/bsize)$ blocks, with *bsize* the size of the output of the adopted cryptographic hash function, thus requiring x recursive applications of the 3-round OAEP structure. The main advantage of the ROAEP-Mix with respect to AES-Mix and OAEP-Mix is the flexibility in the definition of the mini-block size (*msize*). For AES-Mix and OAEP-Mix, the largest *msize* is 64 bits (half the size of the AES block) and half the size of the output of the selected hash function (e.g., 256 bits in our implementation with BLAKE2), respectively. For ROAEP-Mix, the largest mini-block size corresponds to the size of the output of the selected cryptographic hash function (e.g., 512 bits in our implementation). This provides stronger protection against brute-force attacks. While in most application scenarios the use of AES-Mix (and 32 or 64 bits for the mini-block size) can be appropriate, some applications may want more flexibility, especially if having larger mini-blocks does not have a significant impact on performance. We note that block ciphers (e.g., AES) are often implemented by dedicated circuits within many modern CPUs and the throughput exhibited by their hardware-

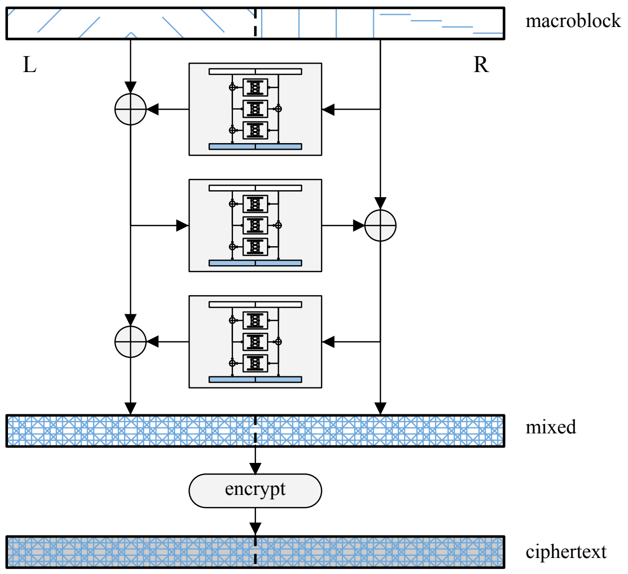


Fig. 12. Recursive OAEP mixing

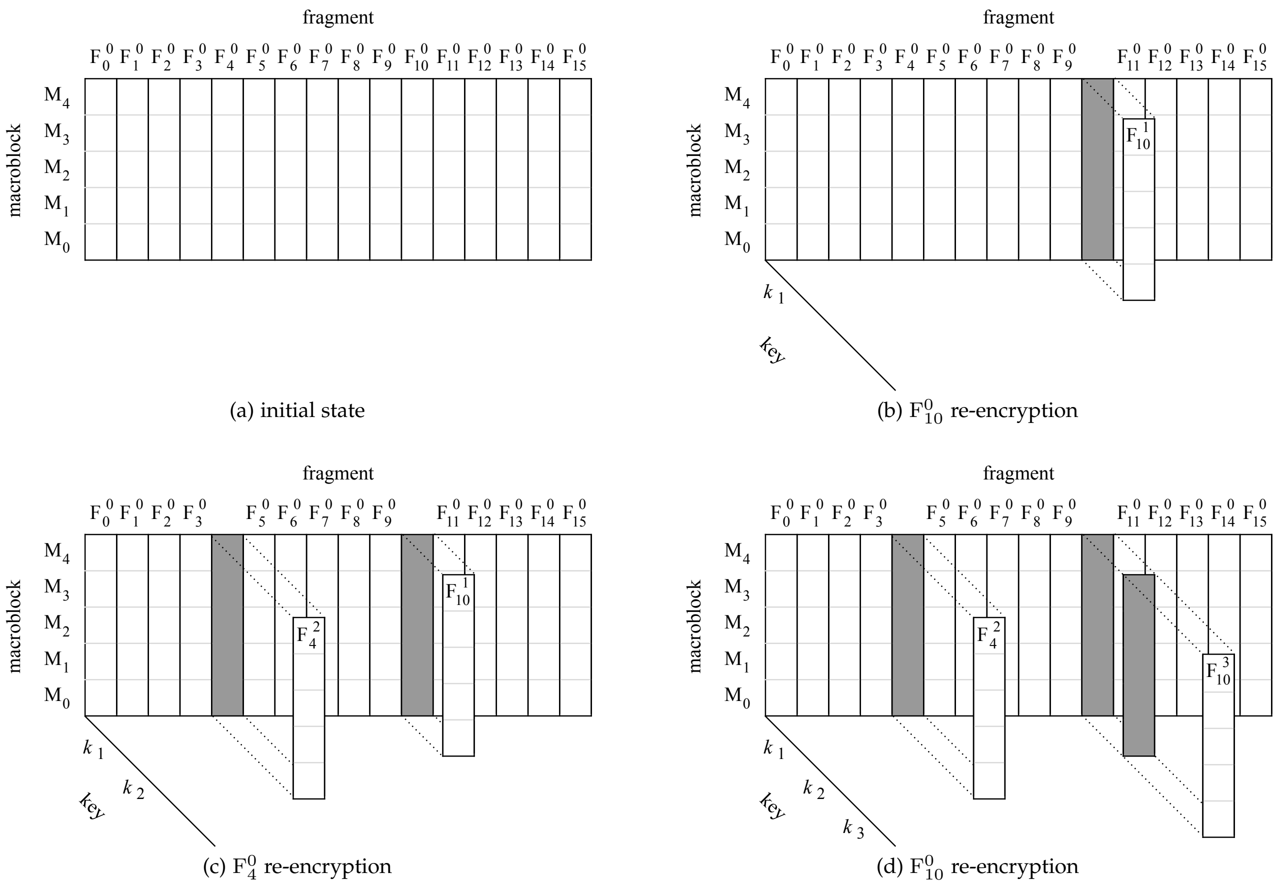
```
ROAEP-Mix(M)
1: Let M=LR s.t. size(L)=size(R)      /* left (L) and right (R) half of M */
2: if $Msize/2 = bsize$        /* L and R can be input of the mix function */
3: then $L_1 := L \oplus mix(R)$                           /* first round */
4:      $R_1 := mix(L_1) \oplus R$                         /* second round */
5:      $L_2 := L_1 \oplus mix(R_1)$                       /* third round */
6: else $L_1 := L \oplus$ ROAEP-Mix(R)                      /* first round */
7:      $R_1 :=$ ROAEP-Mix($L_1$) $\oplus$ R                  /* second round */
8:      $L_2 := L_1 \oplus$ ROAEP-Mix($R_1$)                  /* third round */
9: if $Msize = b \cdot bsize$
10: then return($E(k, L_2 || R_1)$)      /* mixed and encrypted macroblock */
11: else return($L_2 || R_1$)                          /* mixed macroblock */
```

Fig. 13. Recursive OAEP mixing of a macro-block M

accelerated computation is better than the software computation of cryptographic hash functions, as confirmed by our experiments (Section 6). As CPU architectures evolve with hardware acceleration of hash functions, similar advantages will also be enabled for the recursive application of OAEP.

4 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 unavailable 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



fragment
macroblock
key
M_0 M_1 M_2 M_3 M_4
F_0^0 … F_{15}^0
F_{10}^1
F_4^2
F_{10}^3
k_1 k_2 k_3
(a) initial state
(b) F_{10}^0 re-encryption
(c) F_4^0 re-encryption
(d) F_{10}^0 re-encryption

Fig. 14. An example of fragments evolution

users, to reconstruct the plaintext resource or any portion of it. In other words, it equates to revocation.

Hence, access revocation can be 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 6). 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 14 illustrates an example of fragments evolution due to the enforcement of a sequence of revoke operations. Figure 14(a) is the starting configuration with the original fragments computed as illustrated in Section 2. Figure 14(b-d) is the sequence of rewriting to enforce revocations, which involve, respectively: fragment F_{10}, 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 resulting from the application of the Mix&Slice 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 shadow in correspondence to the previous versions of the fragments denotes 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 [13]. 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 $z \geq i$). By contrast, not knowing the private key, other users cannot perform forward derivation. To note that the cost that users must pay for backward 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,

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

Fig. 15. Revoke on resource R

```
Access
1: download R's descriptor and all its fragments
2: retrieve seed $s_l$ used for the last encryption
3: compute keys $k_0, \ldots, k_l$
4: for each downloaded fragment $F_i^x$ do
5:     if $x > 0$ then
6:         $F_i^0 := D(k_x, F_i^x)$                          /* retrieve the original version of fragments */
7: for $j = 0, \ldots, M-1$ do                               /* reconstruct and decrypt macro-blocks */
8:     $M_j$ := concatenation of mini-blocks $F_i^0[j]$, $i = 0, \ldots, f-1$
9:     UnMix($M_j$)  /* one of: AES-Mix, OAEP-Mix, ROAEP-Mix in decrypt mode */
```

Fig. 16. Access to resource R

s_3 in the example of Figure 14(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) [14], [15]. 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 15 illustrates the pseudocode for the revocation process.

To access a resource, users need first to download the resource descriptor, to retrieve the most recent seed s_l, and all the fragments. With the seed, users can compute the keys necessary to decrypt fragments that have been overwritten, to retrieve their version encrypted with k_0. Then, they can combine the mini-blocks in fragments to reconstruct the macro-blocks in the resource. Finally, they apply mixing in decrypt mode to macro-blocks to retrieve the plaintext resource. Figure 16 illustrates the pseudo-code for the process to access a resource. In the pseudocode, the call to UnMix corresponds to a call to one of our mixing strategies in decrypt mode.

Note that the size of macro-blocks influences the performance of both revoke and access operations. Larger macro-blocks naturally provide better policy update performance as they decrease policy update cost linearly, with limited impact on the efficiency of decryption, since its cost increases logarithmically (Section 6).

5 Effectiveness of the Approach

In this section, we elaborate on the effectiveness of our approach for enforcing revocation, meaning its resilience against possible attacks by users trying to maintain access to resources even after 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, *M* is the number of macro-blocks, and *f* is the number of fragments, with *f*=*m*·*b*.

We consider the threat coming from a user whose access to a resource has been revoked, and who can still download the resource from the server. As a matter of fact, with access policy enforced by encryption, a revoked user can still be able to download the resource after revocation, since it is the encryption itself (and hence the re-writing of fragments in case of revocation) that prevents the 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 the user was authorized for the access. In other words, we assume the user to be able to decrypt the fragments that are in their original state or have been overwritten before the user has been revoked access. Since keys and seeds are compact, such a threat 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 three cases, depending on whether the user stores: complete fragments, portions of them across the whole resource, or an erasure code computed on the fragments.

Local storage of fragments. Suppose a user locally stores (when authorized) some fragments of the resource. Even if one of these fragments is later overwritten for revoking access to the user, and then its most recent version stored at the server is unintelligible to the user, the user would have it 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 among the fragments that the user has stored locally, which has probability (f_{loc}/f) to occur, where f_{loc} is the number of fragments stored by the user. After more policy changes have been enforced, and hence more fragments have been overwritten, such a probability becomes $P_A = (f_{loc}/f)^{f_{miss}}$, where f_{miss} is the number of fragments that have been overwritten since the user has been revoked access. Probability P_A 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 the user 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. Hence, 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 a regular structure for the fragments is sufficient for protection and is then preferable.

Local storage of 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 bits the user does not have, but such an effort will be lower. 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 unmixing rounds until the plaintext is reconstructed, with a great cost. We note that the user can cut down on such cost by locally maintaining, in addition to the portions of the original mini-blocks, also a partial representation of the intermediate results of the computation. This partial representation would allow the user to test correctness of a guess without performing all the mixing rounds. In fact, 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 policy updates for revocations, the advantage of the user keeping partial results of the computation will become ineffective. In addition to this, we note that, also in this case, the storage and computational efforts required to the user do not seem to make this attack much preferable for the user with respect to the choice of locally storing the whole plaintext resource itself in the first place.

Local storage of erasure codes. Instead of storing fragments, or portions of them, a user could compute and locally maintain an erasure code on fragments. Erasure codes [16] support the construction of additional sequences of bits that can be used to compensate the loss of some bits in a message. The simplest erasure code is the parity bit, which permits to verify the integrity of a bit sequence and to recover the value of a bit that may have been lost. In general, an erasure code (e.g., Reed-Solomon codes [17]) of size n bits permits the recovery (i.e., compensates the loss) of up to n bits in a message. Major cloud providers extensively use erasure codes in their storage architectures to improve reliability, as a more efficient alternative to complete replication. In our scenario, the user could compute and locally store an erasure code that permits to mitigate the loss of one or more fragments, or - more precisely - the intelligibility of fragments that have been overwritten for policy revocation. The user can then locally store a code of size equal to t fragments (with $t < f$, where f is the total number of fragments composing the resource) to be able to recover the plaintext of the resource as long as no more than t different fragments have been overwritten and not accessible to the user.

We note that erasure codes represent a more efficient attack strategy, with respect to the two previously discussed, for users aiming to maintain access to resources after revocation. In fact, erasure codes are more compact in size with respect to fragments or portions of them. It is interesting here to evaluate the protection against a user using an erasure code with a comparison between an AONT like the one proposed by Rivest [5] and our proposal. Rivest's scheme relies on the use of a compact key, whose application produces in the process observable states of size smaller than the input, hence violating the *no-shrinking property*. Consequently, a user without an erasure code could store the AONT key and thus be able to invert the transformation even if a fragment has been updated. The missing fragment prevents reconstructing some portions of the plaintext, however, the majority of it would still be accessible. To illustrate, assume the user maintains an erasure code as big as $a\%$ of the resource size and that the owner has overwritten $r\%$ of the resource size. When $r \leq a$, the user would succeed in reconstructing the plaintext resource in both Rivest's scheme and our approach. However, when $r > a$, the Rivest's scheme and our approach differ significantly. With Rivest's AONT, the user can apply the erasure code and then use the AONT key to invert the function and re-gain access to a portion of the resource potentially as big as $100 - (r - a)\%$ of the resource size. By contrast, in our approach, even after applying the erasure code, there is no key that can be used to invert the function, and therefore no part of the resource can be reconstructed. This property is graphically illustrated in Figure 17, showing the percentage of the resource that can be recovered by a user who has maintained an erasure code of size $a = 3\%$ of the resource size, as the percentage of resource that is overwritten by the owner increases. This analysis suggests as a possible approach for the owner to counteract erasure code attacks to overwrite more than a single fragment for policy revocation. Moreover, we note that to build the erasure code, the user had to have access to the whole resource. In scenarios where the major cost for the user is the time and network traffic needed to download the resource, rather than the local cost of storage, erasure codes are not a concern, as the user could have easily built

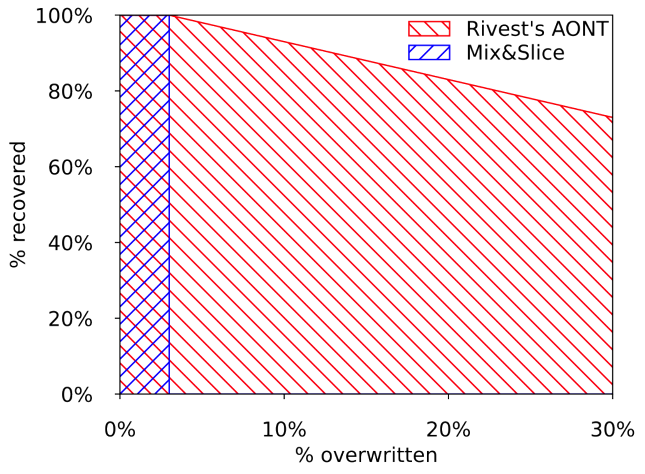


Fig. 17. Comparison of percentage of resource recovered between Rivest's AONT and Mix&Slice when the revoked user has kept an erasure code whose size is 3% of the resource size

a complete local copy of the resource after its download. For scenarios where the crucial parameter is the storage cost at the user-side, revocation should consider the possibility for users to build local erasure codes and therefore the overwriting of more than one fragment for enforcing policy revocation.

A note on collusion. Collusion can happen when two users join their effort to gain access to a resource that neither of them can access (we do not consider collusion with the server, which is assumed trustworthy to enforce the re-writing requested by the owner). Collusion is then represented by users who join their effort in maintaining portions of the resource (e.g., fragments, parts of mini-blocks, or erasure codes 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.

6 Implementation and Experiments

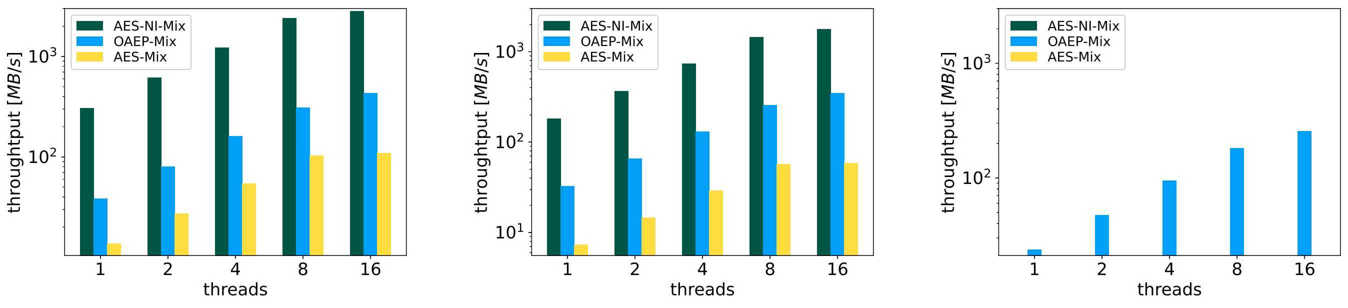
To verify the applicability and the benefits of our proposal, we implemented and tested a client application (Section 6.1) for the encryption/decryption of resources to be accessed. We also implemented the interaction protocol (Section 6.2) between the client and the server for the storage, retrieval, and policy update of resources. For the server, we used Swift as object storage service which is available open source, and is a good representative of what is offered by a modern object storage service for the cloud. For the client, we built a Swift client application that implements the `get` method to retrieve a resource from the server, and the `put_fragments` method to store/overwrite a fragment of a resource at the server. The client application also implements our mix strategies in encrypt and decrypt mode. The client is written in Python with a C component responsible for the invocation of mixing in encrypt or decrypt mode. The experiments have been performed using, for the client, a machine with Linux Ubuntu 22.04 LTS, Intel Xeon CPU E5-2620 v4, 8 cores. For the server, we used an Amazon EC2 m4.xlarge instance, with 4 CPUs and 16 GB of RAM. The client was connected to the Internet by a symmetric 100 Mbps connection. Our experiments evaluated the throughput for reconstructing the plaintext of the resources (i.e., the execution of the mix process in decrypt mode, Section 6.1) and for accessing resources at the server as well as managing policy updates (i.e., the access to a resource and the upload/overwrite of fragments, Section 6.2).

6.1 Mixing throughput

We evaluated the cost, in terms of throughput, of the client for reconstructing the plaintext representation of a resources protected with our approach, which entails the execution of the mixing process in decrypt mode. We considered then the troughput for mixing a macro-block with the different strategies. In particular, we considered the AES-Mix (*bsize*=128 bits) with both the software implementation of AES and its hardware implementation AES-NI, supported by most of the current Intel x86 CPUs, and the OAEP-Mix (*bsize*=512 bits) with BLAKE2 as cryptographic hash function for the internal OAEP layer. The experiments have been performed over a macro-block of 256KiB (512KiB for the OAEP-Mix where the *mix* function operates on half of the input).

Figure 18 compares the throughput obtained by the application of our mixing strategies (i.e., AES-Mix, AES-NI-Mix, and OAEP-Mix), varying the number of threads activated by the client application (1, 2, 4, 8, or 16 threads), and the size *msize* of mini-blocks (32 bits, 64 bits, or 128 bits). Note that AES-Mix requires 8 rounds (i.e., 256KiB=$32 \cdot 4^8$) and 15 rounds (i.e., 256KiB=$64 \cdot 2^{15}$) when *msize*=32 bits and *msize*=64 bits, respectively, compared to the 4 rounds (i.e., 256KiB=$32 \cdot 16^4$) and 5 rounds (i.e., 256KiB=$64 \cdot 8^5$), respectively, required by OAEP-Mix. Also, OAEP-Mix is the only mix strategy that supports mini-blocks of size *msize*=128 bits for which 7 rounds (i.e., 256KiB=$128 \cdot 4^7$) are needed. As visible from Figure 18, OAEP-Mix performs better than AES-Mix, as cryptographic hash functions are more efficient than AES encryption. However, AES-NI-Mix, leveraging hardware implementation, has the best performance. For instance, the AES-NI multi-threaded implementation with *msize*=32 reaches a throughput of 2.5 GB/s. The figure also shows that, increasing the number of threads, we reach a performance level that is 8 times the one obtained by the single-threaded implementation. This is consistent with the presence of 8 physical cores in the CPU we used, each with a dedicated AES-NI circuitry.

The results of our experiments also show that, even if our (AES and OAEP) mixing 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 performance is orders of magnitude better than



(a) *msize*=32 bits (b) *msize*=64 bits (c) *msize*=128 bits

Fig. 18. Mixing throughput comparison, varying the size (*msize*) of mini-blocks and the number of client-side threads

the bandwidth of current network connections, even for a large number of fragments. As shown in Figure 18, the encryption and decryption speed of Mix&Slice can reach up to 2.5 GB/s (or 20 Gbps) when 16 threads are used. This is 1000 times faster than the bandwidth required to stream a 4K video (20 Mbps), and 20 times faster than commonly available home broadband connections (1 Gbps). For reference, the AWS machine configuration that was used in the tests supported a network bandwidth of 750 Mbps, so the network was always the performance bottleneck in our client-server tests. High-end cloud configurations in AWS offer network connections as fast as 50 Gbps, however these configurations also come with up to 192 cores, enabling even faster encryption speed.

6.2 Access and update throughput

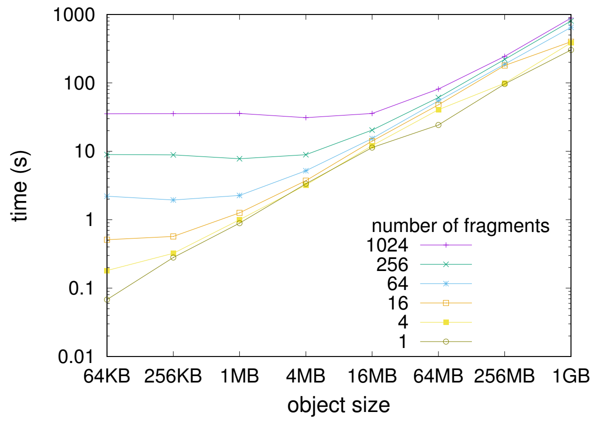
We evaluated the cost, in terms of time, of `get` requests and the cost, in terms of throughput, of `get` requests and policy updates. In our implementation, the interaction protocol operates on top of a Swift server instance installed on the Amazon EC2 platform. For the management of the fragments composing an object (we will use the term object instead of resource to be consistent with the Swift terminology), we considered two options: *1)* fragments are managed as separate objects; *2)* fragments are managed as sub-objects of a single object through the Dynamic Large Objects (DLO)[2] service of Swift, which can split large objects into a number of sub-objects all downloadable through a single request. The first option has the advantage of simplifying policy updates, since the re-encryption of a fragment can be mapped to a single update of the object storing the corresponding fragment. Also, this option is available with any object storage service. In this case, however, the client has the additional overhead of being responsible for opening a large number of connections with the server (one for each fragment) to concurrently access all the fragments of the object and guarantee high performance. With the second option (the use of DLO), which is specific to Swift, the client instead generates a single `get` request for the object, independently from the number of fragments; the Swift server is responsible for mapping such a request into a number of independent requests for downloading all the fragments composing the object. An approach similar to the use of DLO can be realized when the object storage service offers the flexibility to operate with get and put only on a portion of the object.

Our experiments consider objects of different size (1MB, 4MB, 16MB, 64MB, 256MB, or 1GB) composed of a variable number of fragments (i.e., 1, 4, 16, 64, 256, and 1024). The configurations with 1 fragment per object represents our baseline, since they correspond to the case where the object is stored in encrypted form without adopting our approach.

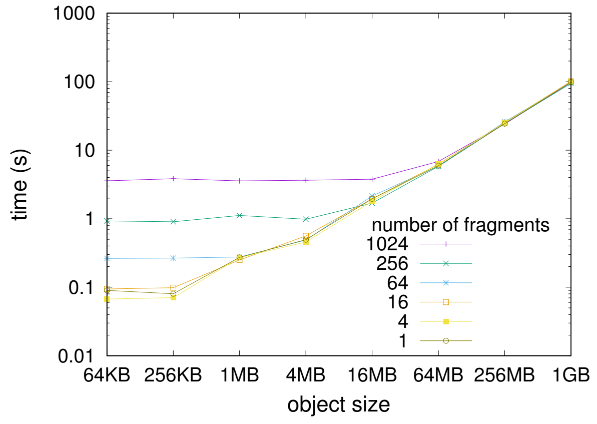
Figure 19 reports the time required for managing a `get` request varying the object size and the number of fragments, when managing each fragment as a single object (Figure 19(a)) and when relying on DLO service (Figure 19(b)). In both scenarios, the graphs clearly show that for medium/large-size objects (i.e., with size greater than 4MB) the overhead introduced by our approach is limited compared with the baseline, especially when adopting the DLO service (Figure 19(b)). For medium/large-size objects, the parameter with the major impact on performance is therefore the network bandwidth. As expected, splitting the resource in a larger number of fragments implies a higher overhead for the execution of the `get` request, especially when the resource is small. The impact of the overhead due to fragmentation is one order of magnitude higher when each fragment is managed as a separate object (Figure 19(a)). This is mainly due to the need of opening a different connection for the download of each fragment.

To evaluate the throughput of our approach when managing policy updates, we considered a workload characterized by one `put_fragment` request after 50 `get` requests on objects in a collection of 1000 objects all of the same size. Figure 20 reports the throughput obtained varying the object size and the number of fragments, when managing each fragment as a single object (Figure 20(a)) and when relying on the DLO service (Figure 20(b)). The figures show that for medium/large-size objects the benefits of our approach in the management of policy updates is significant. In fact, for objects with size greater than 4MB the throughput of the configurations using fragments is always higher than the throughput of the baseline. Indeed, the baseline implies a complete overwriting of the resource at each `put_fragment` request, while our approach manages the same operation overwriting a single fragment having size $1/f$ the size of the resource (with a saving of $(f-1)/f$ the size of the resource). Our approach provides therefore

2. https://docs.openstack.org/swift/latest/overview_large_objects

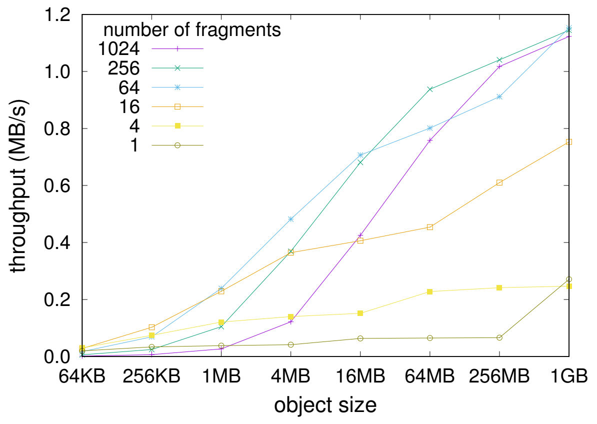


(a) Swift

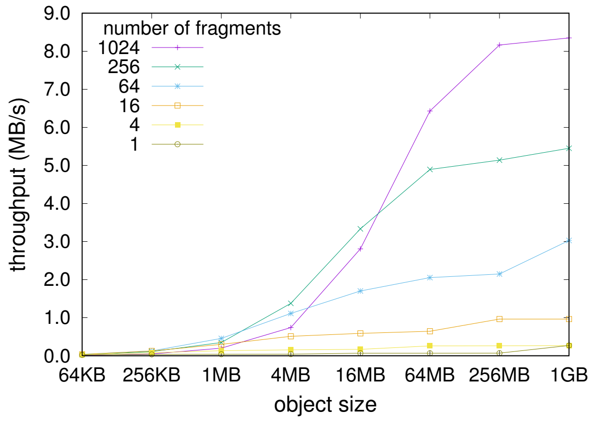


(b) Swift DLO

Fig. 19. Time for the execution of `get` requests



(a) Swift



(b) Swift DLO

Fig. 20. Throughput for a workload combining `get` and `put_fragment` requests

higher scalability compared to the baseline, with a higher advantage if the access policy changes frequently. Comparing Figure 20(a) and Figure 20(b) we can see a significant benefit deriving from the use of the DLO service, thanks to the lower times for the execution of the `get` requests (see Figure 19). We note that the number of fragments has an impact on throughput also in case of policy updates. Indeed, splitting a resource in a higher number of fragments implies a higher cost of `get` operations but a lower cost of `put_fragment` operations, since they overwrite a smaller portion of the revoked resource. Therefore, the number of fragments that better balances the performance of `get` and of `put_fragment` operations can be different depending, for example, on the size of the object and on the frequency of policy updates.

Concluding, our experimental results demonstrate that the size *msize* of mini-blocks and the number of fragments f have an impact on the performance of our solution. While the size of mini-blocks represents our security parameter and must be chosen by the data owners based on their security requirements, the number of fragments (and hence the size of macro-blocks) can be chosen considering performance only. The identification of the best value for the number of fragments, however, has to take into consideration different factors with an impact on performance, including: the size of the objects, the frequency of policy updates, the frequency and average size of `get` requests, and the network bandwidth.

7 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 [5], 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*, where the ciphertext is suffixed with the used random 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 hence be able 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. Different techniques have been proposed for the definition of AONT (e.g., [18], [19]). These approaches, however, are based on the assumption that the key has never been shared with the user.

Most approaches for efficient secure deletion [20], [21] 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 [22] and recent proposals have considered the integration of such approaches with flexible policies [20]. All these approaches are not applicable in our scenario, since the server does not have access to (and hence does not store) the key.

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., [23], [24], [25], [26], [27], [28]) 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 not always easily and efficiently support access revocation. Approaches based on selective encryption (e.g., [15], [29], [30]) 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 [15], [29] or updatable encryption [31]. Over-encryption consists in requesting the server to enforce an additional encryption layer on the encrypted resources so to enforce policy changes. Updatable encryption schemas instead directly support updates in the key used for encrypting data by sending to the server a short update token, without revealing encryption keys. Although over-encryption and updatable encryption enable enforcement of policy changes without requiring to download resources, they require the collaboration of the server for enforcing the changes. On the contrary, Mix&Slice can be used also if the server is completely unaware of its adoption. Similarly to Mix&Slice, *Knob* [32] enforces revocation through the re-encryption of a small portion of the resource and does not require trust assumptions on the server. It however relies on a trusted hardware component. While providing more efficiency compared to Mix&Slice, *Knob* does not provide protection against users who accessed the resource before being revoked access.

A related line of work addresses the problem of protecting the confidentiality of encrypted data stored in a distributed environment in case of key exposure. The approaches in [33] and [34] leverage Mix&Slice and a new secret sharing schema, respectively, combined with data fragmentation to prevent resource reconstruction by making a single fragment unavailable.

8 Conclusions

We presented an approach for efficiently enforcing access revocation on encrypted resources stored at external providers. Our solution includes a mixing phase followed by a slicing phase. The mixing phase transforms the original plaintext resource in an encrypted resource where the whole encrypted representation is needed to go back to the original plaintext resource. We showed different strategies for implementing this mixing that differ in the performance and security guarantee offered. The slicing phase splits the encrypted resource in fragments that represent the unit of revocation since the lack of a fragment makes it impossible for a user to reconstruct the plaintext resource. We showed that our approach 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.

Acknowledgments

This work was supported in part by the EC under grants MARSAL (101017171), GLACIATION (101070141), EdgeAI (101097300), by the Italian MUR under PRIN project HOPE, and by project SERICS (PE00000014) under the MUR NRRP funded by the EU - NextGenerationEU.

References

[1] 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 CCS*, Vienna, Austria, October 2016.

[2] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati, "Towards owner-controlled data sharing," in *Advances in Computing, Informatics, Networking and Cybersecurity*, P. Nicopolitidis, S. Misra, and L. Yang, Eds. Springer, 2022.

[3] S. Xie, M. Mohammady, H. Wang, L. Wang, J. Vaidya, and Y. Hong, "A generalized framework for preserving both privacy and utility in data outsourcing," *IEEE TKDE*, vol. 35, January 2023.

[4] S. Sharma, A. Burtsev, and S. Mehrotra, "Advances in cryptography and secure hardware for data outsourcing," in *Proc. of ICDE*, Dallas, TX, USA, April 2020.

[5] R. Rivest, "All-or-nothing encryption and the package transform," in *Proc of FSE*, Haifa, Israel, January 1997.

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

[7] 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

[8] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *Proc of EUROCRYPT*, Perugia, Italy, May 1994.

[9] V. Boyko, "On the security properties of OAEP as an all-or-nothing transform," in *Proc. of CRYPTO*, Santa Barbara, CA, USA, August 1999.

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

[11] M. J. Dworkin, "Sp 800-38c. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2004.

[12] M. J. Saarinen and J.-P. Aumasson, "The BLAKE2 cryptographic hash and message authentication code (MAC)," Internet Requests for Comments, RFC Editor, RFC 7693, 11 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7693

[13] 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.

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

[15] 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, September 2007.

[16] 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 ATC*, Boston, MA, USA, June 2012.

[17] 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.

[18] G. O. Karame, C. Soriente, K. Lichota, and S. Capkun, "Securing cloud data under key exposure," *IEEE TCC*, vol. 7, no. 3, pp. 838–849, February 2017.

[19] H. Qiu, K. Kapusta, Z. Lu, M. Qiu, and G. Memmi, "All-or-nothing data protection for ubiquitous communication: Challenges and perspectives," *Information Sciences*, vol. 502, pp. 434–445, 2019.

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

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

[22] "TCG storage security subsystem class: Opal," August 2015. [Online]. Available: www.trustedcomputinggroup.org/wp-content/uploads/TCG_Storage-Opal_SSC_v2.01_rev1.00.pdf

[23] Y. Zhang, R. H. Deng, S. Xu, J. Sun, Q. Li, and D. Zheng, "Attribute-based encryption for cloud computing access control: A survey," *ACM Computing Surveys*, vol. 53, no. 4, pp. 83:1–83:41, July 2021.

[24] 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.

[25] 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.

[26] S. Xu, J. Ning, X. Huang, Y. Li, and G. Xu, "Untouchable once revoking: A practical and secure dynamic EHR sharing system via cloud," *IEEE TDSC*, vol. 19, no. 6, pp. 3759–3772, November-December 2022.

[27] S. Yu, C. Wang, K. Ren, and W. Lou, "Attribute based data sharing with attribute revocation," in *Proc. of ASIACCS*, Beijing, China, April 2010.

[28] C. Ge, W. Susilo, J. Baek, Z. Liu, J. Xia, and L. Fang, "Revocable attribute-based encryption with data integrity in clouds," *IEEE TDSC*, vol. 19, no. 5, pp. 2864–2872, September-October 2022.

[29] 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.

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

[31] D. Boneh, S. Eskandarian, S. Kim, and M. Shih, "Improving speed and security in updatable encryption schemes," in *Proc. of ASIACRYPT*, December 2020, (virtual).

[32] S. Contiu, L. Réveillère, and E. Rivière, "Practical active revocation," in *Proc. of Middleware*, Delft, Netherlands, December 2020.

[33] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Securing resources in decentralized cloud storage," *IEEE TIFS*, vol. 15, no. 1, pp. 286–298, December 2020.

[34] K. Kapusta, M. Rambaud, and G. Memmi, "Revisiting shared data protection against key exposure," in *Proc. of ASIACCS*, Taipei, Taiwan, October 2020.

Enrico Bacis is a Senior Software Engineer in the Applied Privacy Research team at Google Zurich. His research interests are in improving the privacy of the web ecosystem. He contributed to more than 20 academic publications in the security and privacy field and, during his Ph.D., he spent several months in Google Munich, London, and Zurich.
https://enricobacis.com

Sabrina De Capitani di Vimercati is a professor at the Università degli Studi di Milano, Italy. Her research interests are in data security and privacy. She has published more than 210 papers in journals, conference proceedings, and books. She has been a visiting researcher at SRI International, CA, USA, and George Mason University, VA, USA.
https://decapitani.di.unimi.it

Sara Foresti is a professor at the Università degli Studi di Milano, Italy. Her research interests are in data security and privacy. She has published more than 100 papers in journals, conference proceedings, and books. She has been a visiting researcher at George Mason University, VA, USA. She chairs the IFIP WG 11.3 on Data and Application Security and Privacy.
https://foresti.di.unimi.it

Stefano Paraboschi is a professor at the Università degli Studi di Bergamo, Italy. His research focuses on information security and privacy, Web technology for data intensive applications, XML, information systems, and database technology. He has been a visiting researcher at Stanford University and IBM Almaden, CA, USA, and George Mason University, VA, USA.
https://cs.unibg.it/parabosc

Marco Rosa is a Security Researcher in the Threat Intelligence team at SAP Security Research. His research interests mainly focus in protecting cloud platforms from unauthorized accesses. He contributed to more than 15 academic publications in the security and privacy field, and during his Ph.D he spent 5 months in SAP Security Research as an intern.

Pierangela Samarati is a professor at the Università degli Studi di Milano, Italy. Her main research interests are in data protection, security, and privacy. She has published more than 290 papers in journals, conference proceedings, and books. She has been a visiting researcher at Stanford University, CA, USA, SRI International, CA, USA, and George Mason University, VA, USA. She is a Fellow of ACM, IEEE, and IFIP.
https://samarati.di.unimi.it