Ref. Ares(2020)2818692 - 31/05/2020

Project title: Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control
Project acronym: MOSAICrOWN
Funding scheme: H2020-ICT-2018-2
Topic: ICT-13-2018-2019
Project duration: January 2019 – December 2021

D4.1

First Version of Encryption-based Protection Tools

Editors: Sara Foresti (UNIMI)
Giovanni Livraga (UNIMI)
Reviewers: Jonas Böhler (SAP SE)
Aidan O Mahony (EISI)

Abstract

This deliverable describes the first version of the implementation of encryption-based techniques developed in MOSAICrOWN. Two tools are presented. The first tool enables the dynamic enforcement of encryption to protect data in the execution of computations involving different parties. It enables the assignment of the different operations according to authorizations and economic considerations. The second tool implements an advanced, and efficient, AONT (All-Or-Nothing-Transform) encryption mode that offers strong protection guarantees, even in case of key leakage, and provides support for the efficient enforcement of access control policy updates.

Type	Identifier	Dissemination	Date
Deliverable	D4.1	Public	2020.05.31

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825333.

MOSAICrOWN Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	EMC Information Systems International	EISI	Ireland
3.	Mastercard Europe	MC	Belgium
4.	SAP SE	SAP SE	Germany
5.	Università degli Studi di Bergamo	UNIBG	Italy
6.	GEIE ERCIM (Host of the W3C)	W3C	France

Disclaimer: The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The below referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2020 by Università degli Studi di Bergamo and Università degli Studi di Milano.

Versions

Version	Date	Description
0.1	2020.05.05	Initial Release
0.2	2020.05.26	Second Release
1.0	2020.05.31	Final Release

List of Contributors

This document contains contributions from different MOSAICrOWN partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 1: Dynamic wrapping for protecting data in computations	Dario Facchinetti (UNIBG), Giovanni Livraga (UNIMI)
Chapter 2: Efficient AONT enforcement	Enrico Bacis (UNIBG), Sara Foresti (UNIMI)
Chapter 3: Conclusions	Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)

Contents

Executive Summary **7**

1 Dynamic wrapping for protecting data in computations **9**

1.1 Basic concepts and definitions 9
1.2 Rationale of the approach 12
1.3 Implementation 13
1.4 Tool installation and usage 18
1.5 Experimental results 20
1.6 Summary 21

2 Efficient AONT enforcement **22**

2.1 Mix&Slice 22
2.2 Aesmix library 24
2.3 Padding management 25
2.4 Fragment management and key regression 26
2.5 Tool installation and usage 27
2.5.1 Aesmix C library 28
2.5.2 Python wrapper 28
2.6 Experimental results 30
2.7 Summary 30

3 Conclusions **31**

Bibliography **32**

List of Figures

1.1 An example of a query plan enriched with relation profiles, assignees for the execution of its operations, and encryption/decryption operations (a), and of a set of authorizations (b) . . . 10
1.2 Rationale of the approach . . . 12
1.3 Execution flow of our tool . . . 13
1.4 Computing engine execution steps . . . 15
1.5 Execution flow of the tasks for assigning operations to providers . . . 16
1.6 Cost estimate varying the complexity of the UDF, considering a *Single-Provider* and a *Multi-Provider* scenario . . . 20
1.7 Average time required by each cost optimization step . . . 20

2.1 An example of mixing (a) and of slicing (b) . . . 23
2.2 An example of fragments evolution . . . 24
2.3 Overview of the key regression scheme . . . 27
2.4 Time required to encrypt a 1GiB resource . . . 30

Executive Summary

This document accompanies the release of the first version of advanced encryption-based tools developed in MOSAICrOWN for protecting confidentiality of data ingested, stored, and processed in a digital data market. The tools implement some of the technical solutions designed in the context of the project and described in deliverable D4.2 "Report on encryption-based techniques and policy enforcement" [FP20]. The tools provide protection of data by *wrapping* the data and making them intelligible only to users who have access to the decryption key. In particular, the wrapping protection layer realized by the tools described in this document provides guarantees of data confidentiality in storage and processing.

The document is organized in two chapters presenting two different tools, considering two aspects that are critical in the digital data market scenario, each implementing the advanced techniques leveraging encryption for wrapping data with a self-protection layer to guarantee their confidentiality against non authorized or non fully trusted parties.

The first tool, presented in Chapter 1, enables the identification of an approach for the management of queries when the data are associated with a security policy and the parties executing the queries may have partial access to the data. The idea is that a query plan compliant with the policy will have to apply adequate encryption techniques every time some data are passed to a party that does not have the privilege to access the data content in plaintext. The consideration of this security dimension extends the opportunity to operate over data in a distributed scenario, but it also increases the complexity of query optimization. The tool implements a strategy that aims at producing, in a short time, a query execution plan that at the same time is efficient and respects the constraints specified in the policy.

The second tool, presented in Chapter 2, considers the application of an enhanced Mix&Slice encryption technique to protect data in storage. The tool extends and improves the *Aesmix* library, making it usable in several scenarios. The core implementation of the mixing has been extended with a flexible support for the invocation of the OpenSSL EVP APIs, leveraging the hardware-accelerated AES-NI primitives when available, to efficiently wrap resources of arbitrary size. The tool also integrates a key regression scheme to improve the support of key management for secure deletion and robust management of access control policy updates.

1. Dynamic wrapping for protecting data in computations

The combination of data owned by different parties and stored in the data market requires proper protection and controlled sharing of sensitive information. This chapter describes the work done toward the implementation of a tool supporting collaborative computations using dynamic, on-the-fly wrapping as will be discussed in Deliverable D4.2 [FP20]. This represents a significant technical challenge, as the combined consideration of collaboration and security aspects (regulating release of data for the computation itself) requires a delicate extension of the architecture of existing approaches for query optimization. Our approach also considers economical aspects, with the goal of identifying an execution plan for collaborative query evaluation that satisfies security constraints (permitting to share data with authorized subjects only) and is more economical than local/centralized execution. We developed a standalone preliminary version of the tool to demonstrate the effectiveness of the proposed approach[1]. The developed tool aims at providing data owners with a wrapping technique enabling the controlled combination of data possibly owned by different parties. Our tool maintains owners in control of their data, permitting them to easily specify who can access them (in plaintext or encrypted form). This chapter first illustrates some basic concepts of the security model assumed by our tool (Section 1.1). It then gives a high-level overview of the approach adopted for the development of our tool (Section 1.2) and discusses the details of the tool implementation (Section 1.3). Finally, the chapter discusses the details of the installation and use (Section 1.4) of the proposed tool, and the experimental results obtained with its usage (Section 1.5).

1.1 Basic concepts and definitions

Collaborative computations can benefit from the presence of subjects offering computational resources at competitive prices. However, data could be sensitive and should undergo access restrictions that can affect the possibility of relying on external subjects for their management and processing. We address this problem by leveraging an approach that enables collaborative and distributed query execution with the controlled involvement of subjects that might not be fully trusted to access the data content [DFJ$^+$17]. For concreteness, but without loss of generality, the approach is framed in the context of relational database systems. The proposed approach is based on the definition of three levels of visibility:

- *plaintext visibility*: the subject can access the plaintext values of the attribute of a relation;
- *encrypted visibility*: the subject cannot access the plaintext values of the attribute of a relation, but can view an encrypted version of the same;

[1] The code is publicly available at `https://github.com/mosaicrown/query-opt`

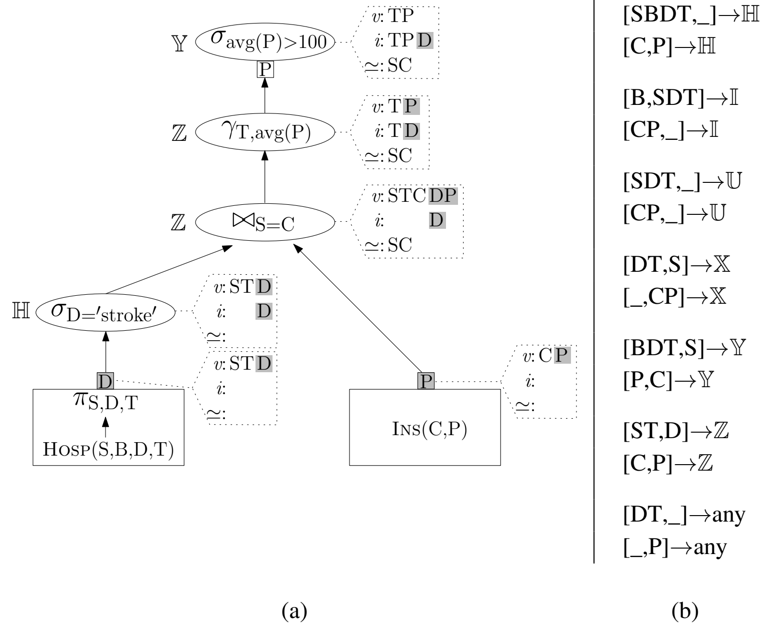


Figure 1.1: An example of a query plan enriched with relation profiles, assignees for the execution of its operations, and encryption/decryption operations (a), and of a set of authorizations (b)

- *no visibility*: the subject cannot access the values of the attributes of a relation, neither plaintext nor encrypted.

To enable the owners of relations (authorities) to formulate authorizations independently (i.e., without the need to coordinate with each other), authorizations are defined in such a way that each authorization regulates the release of a single relation (the one owned by the authority). Formally, given a relation R and a set of subjects $\mathscr{S}$, an authorization is a rule of the form $[P,E] \rightarrow S$, where $P \subseteq R$ and $E \subseteq R$ are subsets of attributes in R such that $P \cap E = \emptyset$, and $S \in \mathscr{S} \cup \{\text{any}\}$.

Authorization $[P,E] \rightarrow S$ states that subject S can view attributes P in plaintext and attributes E encrypted, while she cannot see the attributes that belong to neither P nor E. Note that each subject can have only one authorization for each relation. A default authorization, specified using keyword ‘any’ as subject of the rule, applies when no authorization has been defined for the subject.

Figure 1.1(b) illustrates a set of authorizations over relations Hosp(SSN, BirthDate, Disease, Treatment) and Ins(Customer, Premium) for a set of six subjects ℍ (owner of relation Hosp), 𝕀 (owner of relation Ins), 𝕌 (the user posing the query), 𝕏, 𝕐, and ℤ (subjects offering computational capabilities), along with the default authorization for ‘any’. In the figure, attributes are denoted using their initials and, for the sake of readability, in the authorizations we denote a set of attributes simply with the sequence of attributes composing it, omitting the curly brackets and commas (e.g., SBDT stands for {S,B,D,T}). Clearly, each owner is authorized to access all the attributes in its relation in plaintext (e.g., ℍ can access in plaintext all the attributes of relation Hosp), and possibly also attributes of other relations in plaintext or encrypted (e.g., ℍ can access attribute C of relation Ins in plaintext and attribute P encrypted). External subjects offering computational capabilities can access a subset of the attributes of the relations in plaintext or encrypted.

To verify if a subject is authorized to see a relation (base or resulting from the evaluation of a sub-query) it is necessary to capture its information content. To this purpose, the idea is to enrich each relation with a *relation profile* that depends on the explicit and implicit information leaked

by the relation. Implicit attributes are attributes that do not belong to the relation schema but have left a trace in the computation of the relation itself (e.g., a selection condition 'A=50' leaks the fact that all the tuples in the result have value of A equal to 50, disclosing A even if it is not explicitly visible in the relation). Equivalent attributes are instead sets of attributes that have been connected in the computation of the relation itself (e.g., condition 'A=B' implies precise leakage of the values of B from the visibility of A). To take into consideration both plaintext and encrypted visibility of attributes, the relation profile is defined as a quintuple $[R^{vp}, R^{ve}, R^{ip}, R^{ie}, R^{\simeq}]$ where: R^{vp} and R^{ve} are the *visible* attributes appearing in R's schema in plaintext (R^{vp}) or encrypted (R^{ve}) form; R^{ip} and R^{ie} are the *implicit* attributes conveyed by R, in plaintext (R^{ip}) or encrypted (R^{ie}) form; and $R^{\simeq}$ is a disjoint-set data structure representing the closure of the equivalence relationship implied by attributes connected in R's computation.

Consider the two relations HOSP (for hospital data) and INS (for insurance data) illustrated above and query "SELECT T, avg(P) FROM HOSP JOIN INS ON S=C WHERE D='stroke' GROUP BY T HAVING avg(P)>100" retrieving, for each treatment given to patients hospitalized for stroke, the average insurance premium (if greater than USD100). Figure 1.1(a) illustrates an example of query plan, extended with encryption and decryption operations, for the evaluation of this query. In the figure, each node is complemented with a tag (with dotted fence) representing the profile of the relation resulting from the evaluation of the node itself. In the figure, we use v for visible attributes, i for implicit attributes, and $\simeq$ for equivalence relationships, and represent encrypted attributes on gray background. Consider the join operation: the profile of its result contains attributes STC in the visible plaintext component, and attributes DP in the visible encrypted component, reflecting the fact that attributes D and P had been encrypted beforehand (represented by the gray boxes over D and P). The implicit component includes D in encrypted form, since it keeps track of the evaluation of selection condition D='stroke'. Finally, the equivalence component includes attributes S and C, which have been compared by the join condition.

Given a (base or derived) relation with its profile, a subject is authorized to access the relation if her authorizations enable her to access the information explicitly and implicitly conveyed by the relation itself. More precisely, a subject is authorized for a relation if the following conditions are satisfied.

- The subject is authorized to access in plaintext all the attributes, visible and implicit, represented in plaintext.
- The subject is authorized to access in plaintext or in encrypted form all the attributes, visible and implicit, represented in encrypted form (plaintext visibility naturally implies encrypted visibility since the encrypted representation of attribute values conveys less information than the corresponding plaintext values).
- The subject is authorized to access in the same form, be it plaintext or encrypted, all the equivalent attributes (i.e., attributes appearing in the same equivalence set in the relation). The idea is that the release of any of the attributes in an equivalence set indirectly leaks information also on the values of the other attributes in the same equivalence set. Uniform visibility then prevents unintended information leakage of attribute values due to comparisons in query evaluation. For instance, the evaluation of condition S=C should not leak the values of S to a subject authorized for accessing S in plaintext and C encrypted.

Considering a query plan, each operation should be assigned to a subject for its execution in respect of the authorization policy. Intuitively, an operation can be assigned to any subject that

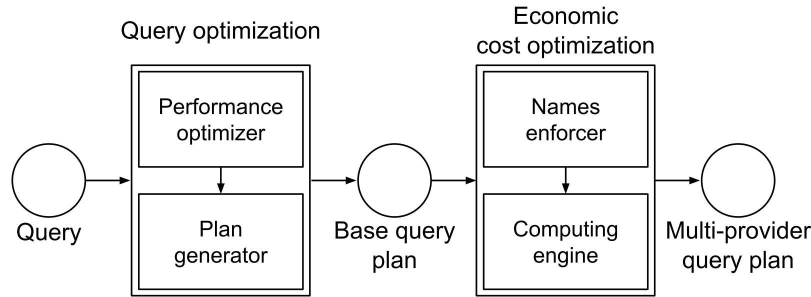


Figure 1.2: Rationale of the approach

is authorized to view: *i)* the operands of the operation, taking into consideration the fact that all the attributes in the relation schema that are not needed for the evaluation of the operation can be encrypted on-the-fly for query evaluation; and *ii)* the operation result. The choice, among all the potential candidates, of the subject in charge of the execution of each operation is then based on economic and/or performance parameters. Encryption and decryption operations can be inserted on-the-fly to adjust visibility of attributes to satisfy authorizations and to enable the evaluation of operations. Encryption can be used to protect attributes so to permit the assignment of operations to subjects that could not be considered otherwise. Decryption permits accessing plaintext values of encrypted attributes when needed for the computation. For instance, for each node in the query plan, Figure 1.1 reports its assignment. Note that attribute D is encrypted before the selection operation since server $\mathbb{Z}$, in charge of the evaluation of the join operation, is not authorized to access it in plaintext.

1.2 Rationale of the approach

Our approach for solving the problem illustrated above operates in two steps (see Figure 1.2). The first step produces an optimized plan, which we call *base query plan*, considering a single provider authorized for plaintext visibility over all base relations, and in charge of query execution (for brevity, when clear from the context, we will refer to a (base) query plan as a (base) plan, omitting the word 'query'). The second step tries to improve (i.e., reduce) the cost of the base query plan obtained in the first step. This goal is achieved by trying to assign operations to other providers, introducing encryption/decryption operations when needed to guarantee authorization enforcement and operation execution.

The estimation of the economic cost takes into consideration the cost of executing relational operations and user defined functions (UDFs), as well as the costs of data encryption and of data transfer when data are to be transferred among different providers. To this end, the costs of operation execution (which take into consideration also the execution performance) are enriched in the second step with estimations for the costs of UDFs. The costs of data encryption and transfer (only relevant to the second step, since the first assumes a single provider only and no encryption) are estimated in the second step when trying to allocate operations to the different providers, based on their price lists. The main advantage of our two-steps approach, compared to solutions aimed at integrating authorization enforcement and cost optimization in existing query optimizers (e.g., [TLL19, SKS+19]), is that it can be easily integrated with existing DBMSs without the need

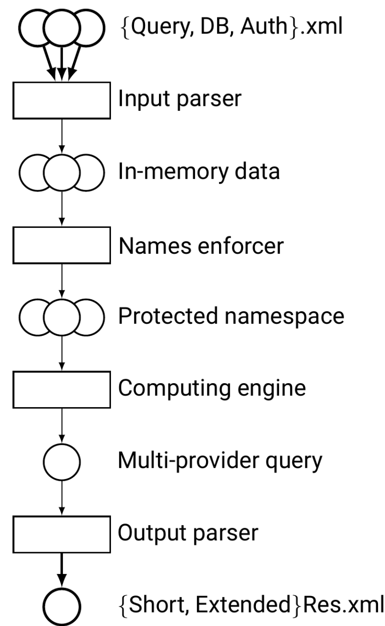


Figure 1.3: Execution flow of our tool

to redesign their internal modules and architecture. This is due to the fact that our solution keeps operations allocation to providers and subsequent cost optimization (second step) separated from the traditional single-provider plan optimization (first step).

The definition of an optimal allocation of operations to providers, to improve the costs of the base query plan, is done through the execution of a set of tasks, carried out by specific software modules (the core modules of our tool are highlighted in the inner boxes in Figure 1.2) that will be described in the following section.

1.3 Implementation

We now illustrate the implementation of our tool for determining an optimal allocation of operations to providers. Figure 1.3 provides a high-level overview of the execution flow of our tool. Boxes in the figure represent the main software modules of the tool. Our tool takes as input three .xml files containing information on: *i)* the query to be executed; *ii)* the relational tables on which the query operates; and *iii)* the authorization policy and economic costs of the providers (more details on such files will be discussed in the remainder of this chapter). It then parses the input files (*input parser* in Figure 1.3) and loads all information of interest in memory. The tool then enforces some preliminary checks on attribute names to avoid ambiguity when different attributes in different relations have the same name (*names enforcer* in Figure 1.3), obtaining a consolidated protected namespace. The tool then performs the core operations for determining an optimal allocation of operations to providers (*computing engine* in Figure 1.3), producing a query plan where operations are assigned to providers and possibly enriched with encryption/decryption operations. It finally parses the output (*output parser* in Figure 1.3) and produces as output two .xml files.

We now illustrate in more details, following their execution flow, the main software modules of our tool.

Input Parser. The input parser is the module in charge of retrieving and loading in memory

all information needed as input for the computation. It is composed of a compact XML parser, receiving as input XML representations of: the query plan (file `Query.xml`), the data over which the query operates (file `DB.xml`), the authorizations and economic costs of providers (file `Auths.xml`). An example of such files is reported in Section 1.4. The parser oversees the building of an in-memory representation of such information, needed for subsequent computations. `DB.xml` contains a representation of the relational schemas involved in the computation and of their attribute sizes. `Auths.xml` contains a representation of the authorization policy (detailing for each provider the sets of attributes that can be accessed plaintext and those that can be accessed only in encrypted form), as well as of the economic costs of the providers. `Query.xml` contains a representation of the base query plan, including statistics and metrics (e.g., cost) of the operations involved, as well as *constraints* on how attributes can be encrypted. We consider single attribute constraints, specifying the kind of encryption that can be applied to each attribute (e.g., a selection on an attribute can be performed if the attribute is encrypted with deterministic symmetric encryption, but not if it is wrapped with randomized symmetric encryption), as well as constraints involving sets of attributes that have to be encrypted with the same encryption, using the same key, to perform a particular operation (e.g., a join).

The parser is based on the JDOM2[2] library, a Java-based solution that provides the utilities for accessing and manipulating XML data from Java code. We leverage the JDOM2 `SAXBuilder`, which is built upon SAX[3] (Simple API for XML), a widely-used and "de-facto" specification describing how XML parsers can efficiently pass information from XML documents to software applications. A peculiarity of our approach in this regard is that we structured the parser in such a way to completely automate the in-memory reconstruction of the query tree plan starting from its XML representation. To this purpose, we implemented a Java class hierarchy that models relational algebra operators, custom UDFs, and encryption/decryption operators. At runtime, for each operator in the .xml file containing the query plan, we first retrieve the `Class` object related to the operator itself (which is referenced by an XML string), then we leverage the Java reflection[4] (a feature of the language that enables program introspection) to get the operators' class constructor. We finally execute the constructor, building a new operator instance. This feature of our parser increases the flexibility of our tool (bypassing type check at compile time, this permits achieving fully dynamic allocation).

Names Enforcer. The Names Enforcer is the module in charge of detecting and resolving *name conflicts*, that is, the use of the same textual string to refer to two different elements inside the query plan. An example of that is the use of the same name for two different attributes, for instance belonging to the schemas of two different relations. This conflict can complicate and/or invalidate the enforcement of the authorization policy as well as the definition of the needed encryption/decryption operations, since attribute names are not unique.

This module operates incrementally, determining the origin of each name. To this end, starting from the leaves of the query plan, it performs the required substitutions and propagates the changes up to the root of the tree. More precisely, the module performs a post-order traversal of the query plan, hence starting the naming corrections from the leafs of the tree. From the evaluation of the leaf, the module determines a set of name bindings: name repetitions are erased and, for each repetition, a pair (*old_name, new_name*) is created. The set of bindings is then passed to the parent

[2]http://www.jdom.org/index.html
[3]http://www.saxproject.org/
[4]https://www.oracle.com/technical-resources/articles/java/javareflection.html

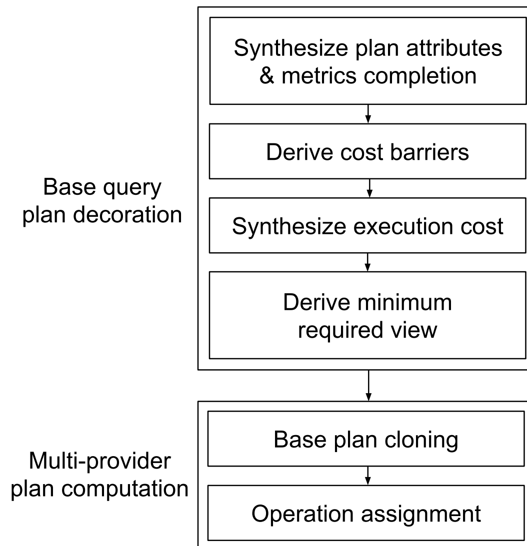


Figure 1.4: Computing engine execution steps

node, so to propagate the corrections to the entire query plan.

When the computations performed by the Names Enforcer terminate, the entire namespace is made consistent, hence permitting to correctly and uniquely address attributes in the query plan, authorizations, and encryption/decryption constraints.

Computing Engine. The Computing Engine represents the core module of our tool, in charge of: *i)* decorating the base query plan with information needed for the allocation of operations to providers; and then *ii)* identifying a multi-provider query plan, with operations assigned to different providers, so to reduce the economic cost of query execution. A high-level graphical representation of the main operations involved in those steps is illustrated in Figure 1.4. We now illustrate these two main steps, together with the tasks executed in each of them.

- **Base query plan decoration.** The main goal of this step is to decorate (i.e., enrich) the base query plan with estimates of economic costs, relation profiles, and minimum required views (i.e., constraints that specify the minimum visibility, be it plaintext or encrypted, over the attributes for performing an operation), that is, with the information necessary for identifying the allocation of operations to providers. The first task in this regard involves *attributes synthesizing and metrics completion*: the input plan is traversed in post-order and operation execution is simulated to obtain the profiles of all relations involved in the computation (*attributes synthesizing* in Figure 1.4). Also, each node is enriched in the traversal with information on the metrics (e.g., costs) characterizing the node's operator (*metrics completion* in Figure 1.4). Metrics completion is performed by the `CostEngine`, a small utility in charge of considering the memory and I/O profiles of the node's operator for producing an estimate of the economic cost.
  After these preliminary tasks, the so-called *cost barriers* of each node are estimated. Within this task, nodes in the plan are classified based on whether their operations' cost is dominated by *CPU usage* or *data storage*. In case a node cost is dominated by CPU usage, such information is propagated down to the leaves, to mark the fact that resorting to less costly providers would be recommended for this path (with the rationale being that usage of CPU power is typically the most expensive resource according to the providers' price lists).

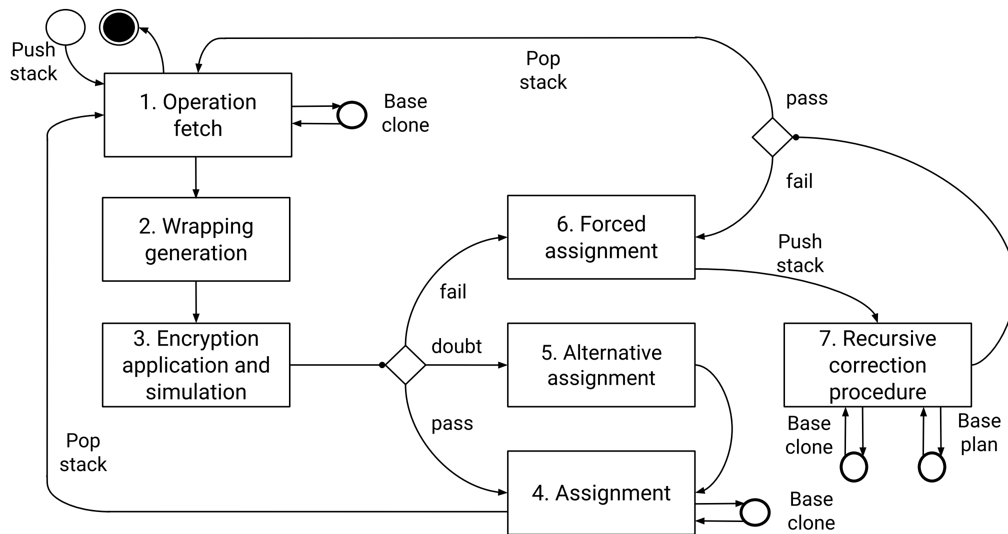


Figure 1.5: Execution flow of the tasks for assigning operations to providers

Based on the information computed up to this point, the overall cost of executing the query is estimated (*synthesize execution cost* in Figure 1.4) during a post-order tree traversal, and the *minimum required views*, necessary for the allocation of operations to providers, are computed during a tree traversal in pre-order.

- **Multi-provider plan computation.** After having decorated the base query plan with all necessary information, the actual process for allocating operations to providers starts. The decorated plan is cloned (*base plan cloning* in Figure 1.4), obtaining a working copy on which the allocations and modifications introduced by the computing engine can be written without losing the actual information of the original base plan, which is kept as a reference. For the sake of readability, we refer to the working copy of the decorated base plan with the term *clone*. Assignment choices are evaluated comparing the economic cost of execution of the base plan and the clone (*Operation assignment* in Figure 1.4). To this end, we adopt a greedy approach choosing, for each operation, the candidate that minimizes the evaluation cost of the considered operation. Such an approach identifies a good, even if not always optimal, assignment of operations to providers. We now discuss the tasks executed for allocating operations to providers, graphically illustrated in Figure 1.5.

  Step 1: an operation is fetched from the clone. Note that operation retrieval is performed through a post-order traversal of the query plan. Steps 2–7 are then executed for each operation fetched.

  Step 2: an attempt is made to outsource to the available providers the operation fetched in the previous step, together with the relation over which the operation works. This operation is performed by the `PolicyMovesGenerator` sub-module of our tool that, considering one provider at a time, determines the possible encryption that can be applied to the single attributes. This is based on the authorizations holding for the specific provider, as well as on the possible constraints regulating the use of encryption (such as on adopting the same key for the attributes involved in the computation of a join, or on keeping certain attributes plaintext for operations that cannot operate on encrypted data). Note that our tool can operate with randomized symmetric encryption, deterministic symmetric encryption, Paillier

encryption (a semi homomorphic scheme useful to perform sums) and order preserving encryption (OPE). Based on the operation to be performed, this step then determines which attributes need to be encrypted, and with which scheme. The attributes involved in the computation are classified in two sets, based on whether they need to stay plaintext (for operation execution) or they need to be encrypted (for authorization enforcement). The module stops (and hence does not generate any encryption operation), if any of the following conditions is satisfied: *i)* at least one attribute belongs to both the attributes that need to stay plaintext and the attributes that need to be encrypted; *ii)* the provider in charge of the operand relation cannot (for its authorizations) decrypt the attributes that need to be plaintext; and *iii)* attributes that need to have the same visibility level cannot have it.

Step 3: in case encryption operations have been determined in the previous step, such operations are applied and evaluated, adjusting accordingly the metrics associated with the node (e.g., increasing the size of the attributes due to the adoption of encryption).
After encryption has been evaluated, the module tries to allocate the operation to the available providers, selecting the one that entails the minimum cost. The module evaluates two costs: the actual cost *op_cost* of node evaluation, and the cost *subtree_cost* associated to the execution of the sub-tree rooted at the node. This evaluation is performed comparing these costs of the clone (i.e., the ones just estimated), which is updated step by step, and the related costs in the original decorated base plan. Now three cases can occur: *i)* both *op_cost* and *subtree_cost* are lower than those in the original plan (leading to step 4); *ii)* either *op_cost* or *subtree_cost* is lower, while the other one is higher (leading to step 5); and *iii)* both *op_cost* and *subtree_cost* are higher than those in the original plan (leading to step 6).

Steps 4-5-6: here the operation is assigned to a provider. If the simulated costs are lower than those in the original decorated plan (case *i)* above), the assignment is consolidated (step 4) and the procedure returns to step 1, fetching another operation. If one cost is higher and the other cost is lower (case *ii)* above), the module tries a temporary allocation calling a binary search explorer (a structure keeping track of assignment attempts with a binary prefix tree), suggesting an alternative assignment (step 5). The procedure then returns to step 1, fetching another operation. If the costs simulated for allocating the operation to the cheapest provider are worse than the costs in the original decorated plan (case *iii)* above), then the operation is not assigned to any external provider but is forced to remain at its original owner (step 6). If this is the case, the procedure moves to step 7.

Step 7: a recursive correction procedure is executed for each child of the node that leads to step 6, that is, for which allocation to an external provider failed. For each of those child nodes, the execution target is set to the base provider, encryption is removed, and the node's operation is once again simulated producing new cost evaluations. If the total cost of the sub-query rooted at the considered child is now lower than the corresponding sub-query in the base plan, the correction procedure terminates and the procedure returns to step 1, fetching a new node. Otherwise, such correction procedure is recursively triggered on all the descendants.

Output Parser. The Output Parser starts its operation when the Computing Engine terminates, that is, when a multi-provider plan has been defined. This module is in charge of producing the final output of our tool, in terms of two XML representations of the multi-provider plan: one includes

the allocation of the different operations to the providers together with costs, and one including also information on the kind of encryption needed for each attribute (an example of such files is reported in Section 1.4). For the output parser, similarly to the input parser, we leverage the JDOM2 `SAXBuilder`, built upon SAX.

1.4 Tool installation and usage

In this section, we discuss the installation and use of our tool for computing a multi-provider query plan.

Installation

Installation requires to clone the publicly available repository.

```
git clone https://github.com/mosaicrown/query-opt.git
```

After cloning the repository, dependencies must be installed, and the project can be compiled.

```
mvn package
```

The command line to launch the demo is the following.

```
cd ./src/main/java/Launcher; ./demo.sh
```

Input files and usage

The required input files to run our tool (see Section 1.3) are:

- `Query.xml`: the base query plan;
- `DB.xml`: the database schema and related attributes;
- `Auth.xml`: the set of authorizations and the economic costs of providers.

An example of `Query.xml` file is the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<operation>
        <type>projection</type>
        <name>project10</name>
        <basicmetrics>
                <inputsize>7462</inputsize>
                <inputtuplesize>0.06</inputtuplesize>
                <outputsize>2488</outputsize>
                <outputtuplesize>0.02</outputtuplesize>
                <cputime>0.028</cputime>
                <iotime>0.00001</iotime>
        </basicmetrics>
        <projectedset>
                <name>t2.b</name>
                <name>t2.c</name>
        </projectedset>
        <inputattributes>
                <name>t2.b</name>
                <name>t2.c</name>
        </inputattributes>
        <sonoperations>
                <operation>
                        ...
```

The tool understands relational algebra and custom UDF operators. It supports the following two operation modes: (a) debug_mode, which produces a full stack report of execution; and (b) depth_mode, which performs an exhaustive search over the space of alternatives by setting up the maximum height of a binary prefix tree.

The following command, executed within the directory src/main/java/Launcher, performs the assignment of custom input queries.

```
./soqd.sh Query.xml DB.xml Providers.xml [no_uvr] [debug_mode] [depth=X]
```

Output files

The result produced by our tool is a multi-provider plan that minimizes the cost according to our greedy strategy illustrated in Section 1.3, while satisfying authorizations. The result also specifies the encryption and decryption operations extending the base plan.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<operation>
        <type>UDF</type>
        <name>createstablerelations</name>
        <executor>EC2</executor>
        <cost>0.49727904552380947</cost>
        <outRelProf>
                <rvp>
                        <attribute>t1.a1</attribute>
                        <attribute>t2.d4</attribute>
                </rvp>
                <rve/>
                <rip>
                        <attribute>t1.b2</attribute>
                        <attribute>t2.e5</attribute>
                        <attribute>t2.f6</attribute>
                        <attribute>t1.c3</attribute>
                </rip>
                <rie>
                        <attribute>t1.b2</attribute>
                        <attribute>t2.e5</attribute>
                </rie>
                <ces>
                        <set>
                                <attribute>t1.b2</attribute>
                                <attribute>t2.e5</attribute>
                        </set>
                </ces>
        </outRelProf>
        <postEncryptionMoves/>
        <sonoperations>
                ...
```

Debug tracer

Our tool offers a debugging mode (debug_mode option), for which we implemented a report generation utility illustrating the whole process of query plan decoration, as well as of the operation assignment process. For each step performed by the tool, a logger saves a compact string representing the modification. The reports can be saved in a file leveraging Guava (Google Core Libraries for Java[5]), a set of Java language extensions to increase efficiency and

[5] https://github.com/google/guava

	Single-Provider	Multi-Provider
linear	0.041$	0.041$ (0.0%)
pseudo-linear	0.047$	0.019$ (40.4%)
quadratic	3.465$	3.465$ (0.0%)

Figure 1.6: Cost estimate varying the complexity of the UDF, considering a *Single-Provider* and a *Multi-Provider* scenario

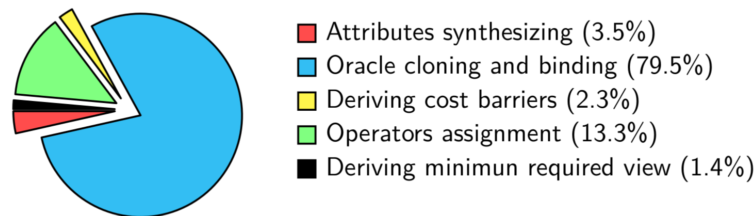


Figure 1.7: Average time required by each cost optimization step

permit safe usage of filesystem resources. In particular, we serialized in-memory reports using a `BufferedWriter` that writes text to a character-output stream, buffering characters to efficiently write single characters, arrays, and strings. A `google.common.io.Closer` class object is used to ensure safe usage of filesystem resources.

1.5 Experimental results

We tested our tool on a set of ad-hoc demo plans to stimulate the features of our tool (e.g., the support for UDFs, not included in the suite of TPC benchmarks proposed by the Transaction Processing Performance Council[6]).

The demo set includes queries with three types of UDFs, having linear, pseudo-linear, and quadratic complexity, respectively. Figure 1.6 reports the economic costs for query evaluation. The cost includes computational costs and data transfer costs. In the table, the *Single-Provider* column shows the monetary cost under the assumption that only the most expensive and most trusted provider is used for the execution of the query. The *Multi-Provider* column assumes the availability of several providers, with less expensive providers associated with a more restrictive access policy and limitations on the visibility of the attributes involved in the query.

We close this section with some notes on the performance of our tool. Experiments have been executed on an Intel i5 server with 16 GB memory and SSD drive running Ubuntu 18.04 LTS. To analyze the performance of our tool, we run a set of experiments to measure the time necessary for building the multi-provider plan (i.e., for optimizing the cost of query execution). The time has been measured using `System.nanoTime` method, which measures time at the granularity of nano seconds. The average time necessary for building the multi-provider plan in our experiments is 26.9ms for query plans with approximately 10 operations, which seems a promising result compatible with the needs of most applications. We also measured the average time required to perform each cost optimization step. The results of these experiments are shown in Figure 1.7

[6] http://www.tpc.org

The experimental evaluation of our approach has shown its effectiveness, especially for computations of high complexity. This work, therefore, shows how to support efficient large-scale analysis by taking advantage of economically convenient providers.

1.6 Summary

In this chapter, we presented our tool for enabling collaborative computations over data stored in the market and possibly owned by different parties. The proposed tool enforces authorizations established by data owners for the selective release of their data (in plaintext or in encrypted form) in the identification of the providers that minimize query evaluation costs. Our tool supports, besides traditional relational operations, also the evaluation of UDFs, today widely used for data analysis. Part of the results presented in this chapter have been published in [BDF+19a].

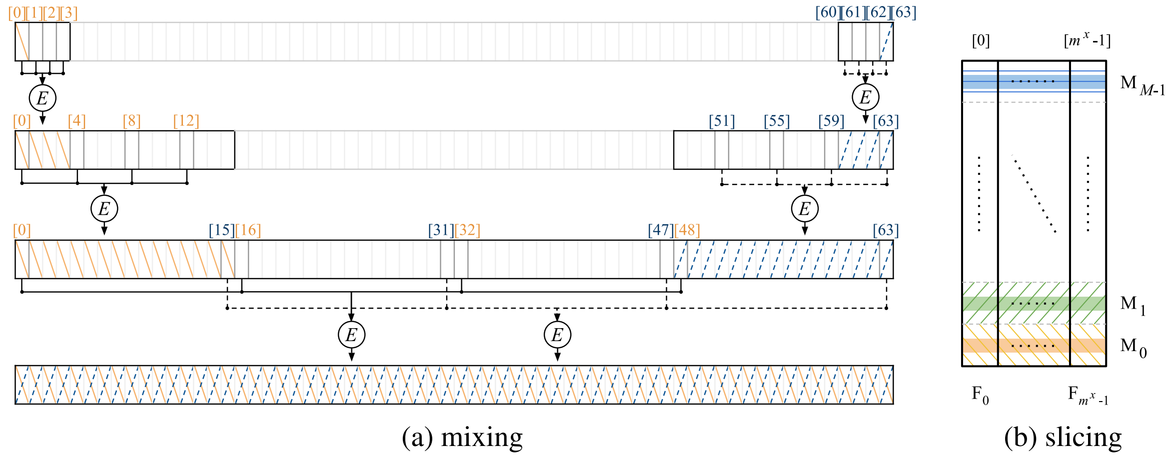
2. Efficient AONT enforcement

The content of resources stored in the data market can be sensitive and might need to be protected to the data market itself. Hence, resource owners often rely on encryption to protect the confidentiality of their sensitive resources. The problem of protecting resource has been addressed by MOSAICrOWN and a solution based on the Mix&Slice encryption mode will be presented in Deliverable D4.2 [FP20]. Mix&Slice is an *All-Or-Nothing Transform* encryption scheme that guarantees complete interdependence among the bits of a ciphertext. In this chapter, we present *Aesmix* library, which implements Mix&Slice. The library, implemented in C, is also integrated in a tool realized in Python to enable wider adoption of the Mix&Slice functionalities. The developed tool aims at providing data owners with a wrapping technique that guarantees protection of data in storage and in transit. Also, our tool permits the efficient management of updates to the access policy by the data owner and enforces secure deletion of resources. This chapter first illustrates the Mix&Slice encryption mode (Section 2.1) and then presents the technical details of Aesmix library (Section 2.2), including the issues related to padding management (Section 2.3). The chapter then illustrates the evolution of a resource stored in the data market and protected using the Mix&Slice encryption mode, to enforce updates to the policy regulating accesses to its content and to permanently delete it, even in case of exposure of the encryption key (Section 2.4). Finally, the chapter discusses the details of installation and use (Section 2.5) of the proposed tool and the experimental results obtained with its usage (Section 2.6).

2.1 Mix&Slice

Mix&Slice [BDF+16] is an *All-Or-Nothing Transform* (AONT) encryption scheme that guarantees complete interdependence (*mixing*) among the bits of the encrypted content. Intuitively, Mix&Slice guarantees 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. Mix&Slice makes each portion of the ciphertext needed, in terms of information theory, to reconstruct any of the portions of the plaintext resource content. Hence, it can be profitably used to both permanently delete a resource by making a portion of it unavailable, and to modify the access control policy by re-encrypting a small portion of the resource with a key known only to authorized users.

We note that traditional AONT approaches consider similar requirements, but they assume that the encryption keys are not known to final users. On the contrary, in a digital data market scenario, final users are expected to know the key used to protect the confidentiality of the content of the resources from the data market itself. Users can locally store encryption keys used to protect



(a) mixing (b) slicing

Figure 2.1: An example of mixing (a) and of slicing (b)

resources, possibly also after the resource has been deleted. Hence, traditional AONT approaches cannot be adopted.

Mix&Slice is based on two concepts, *mixing* and *slicing*, as summarized in the following.

Mixing. Mixing operates by iteratively applying different encryption rounds on the resource content. The first round of encryption is applied at the level of each block composing the resource and provides mixing guarantees at the level of each single block. To extend mixing, each block is considered as composed of a finite number of mini-blocks of fixed length *msize*. At the second round, encryption is applied over blocks including mini-blocks representative of different blocks resulting from the previous round of encryption (e.g., assuming each block is composed of 4 mini-blocks, the first round encrypts contiguous sequences of 4 mini-blocks, the second round encrypts sequences of 4 mini-blocks at distance 4, the third encrypts sequences of 4 mini-blocks at distance 16, etc.). The number of rounds necessary for complete mixing of a macro-block (i.e., sequence of contiguous blocks in the resource) composing a resource depends on the number of blocks in the macro-block. Figure 2.1(a) illustrates an example of mixing of 64 mini-blocks assuming each block to include 4 mini-blocks. In the figure, we use notation $[i]$ to denote the i-th mini-block in a macro-block M.

Slicing. Slicing consists in defining fragments for the resource. Each fragment includes a mini-block for each macro-block, and each mini-block is represented in exactly one fragment. Figure 2.1(b) illustrates an example of slicing of a resource composed of M macro-blocks, each including m^x mini-blocks, denoted $[0],\ldots,[m^x-1]$. In the figure, we use notation F_i to denote the i-th fragment (a column in the figure) and notation M_j to denote the j-th macro-block (a row in the figure).

The availability of all the fragments and of the encryption key is a necessary condition for decrypting a macro-block, and hence access the resource content. Permanent deletion of a resource can then be achieved by making one of its fragments unavailable (e.g., encrypting it with a key known to the resource owner only). Similarly, policy updates can be enforced by re-encrypting a randomly chosen fragment with a new key, known only to authorized users. Indeed, even if all the users initially authorized for the resource know the key used by Mix&Slice to initially encrypt the resource, only the users authorized after the policy update can decrypt the re-encrypted fragment and hence decrypt the resource. Clearly, operating on a single fragment considerably reduces the overhead for the resource owner, compared to operating on the whole resource. Figure 2.2

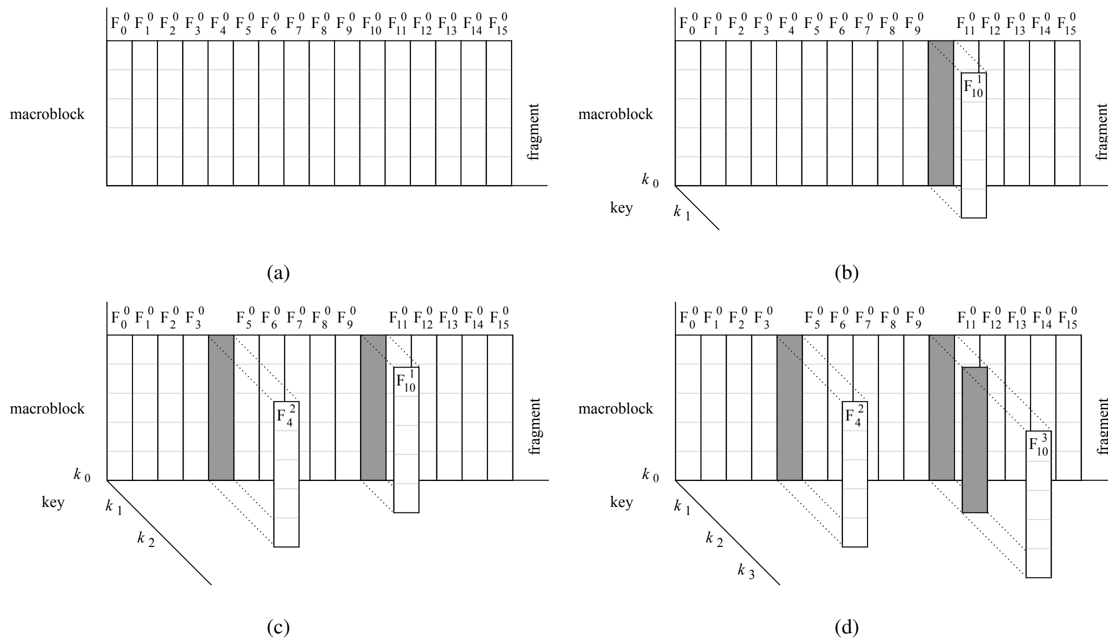


Figure 2.2: An example of fragments evolution

illustrates an example of fragment evolution because of policy updates. In the figure, shadowing represents the fact that the original fragment version is not available, as it has been substituted by a new version.

2.2 Aesmix library

Aesmix is a library, written in C, that implements the Mix&Slice encryption mode. The implementation of Aesmix leverages AES [Ann01] as its building block. Indeed, Mix&Slice uses a symmetric cryptographic function to 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.

Since most current x86 processor architecture, including the architectural framework for both Intel and AMD processors, offers the support for a hardware implementation of AES (e.g., AES-NI [Gae12] for Intel architectures), we upgraded the Aesmix library to take advantage of such a hardware accelerated instruction set to improve the performance and flexibility of the Mix&Slice encryption mode. In fact, the hardware implementation of AES is expected to provide a 3x to 10x performance improvement over existing software solutions (for example Crypto++ [Cry]), depending on the mode of operation[1]. To the aim of leveraging the hardware implementation of AES in Aesmix library, we used EVP [EVP] high level interface offered by the OpenSSL cryptographic library [Ope].

To offer a more widespread access of the Mix&Slice capabilities and since MOSAICrOWN policy engine has been implemented in Python, we provide a Python wrapper to Aesmix library. Such a wrapper will ease the integration of Mix&Slice with other MOSAICrOWN tools. Our implementation of the wrapper is based on Python *cffi* (C Foreign Function Interface) library.

[1] https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/

Indeed, the use of *cffi* library permits Python code to invoke C functions available in a compiled shared object (*.dll* for Windows, *.so* for Linux). We opted for this solution since the *cffi* library directly integrates with C *ABI* (Application Binary Interface), and then limits the performance overhead caused when invoking a C function from Python. The developed Python wrapper checks the number of CPU cores available in the system, to limit the computational time of encryption and decryption operations by parallelizing the Mix&Slice operations, performing them in multi-threading.

As illustrated in the following sections, our Python wrapper enhances Aesmix C-library by managing resources of any size, implementing a padding technique that well fits the requirements of Mix&Slice, and fragment segmentation.

2.3 Padding management

While Aesmix C library implementing Mix&Slice implicitly assumes that the size of plaintext resources is a multiple of the size of macro-blocks, usually the size of resources does not perfectly fit the size of macro-blocks. To enable the encryption of resources of arbitrary size, our Python wrapper implements a padding suitable for Mix&Slice, which is applied immediately after the resource is acquired (before invoking the C library implementing mixing encryption phase).

Existing padding schemes are not suited for Mix&Slice. Indeed, padding techniques operate under the assumption that the number of bytes to be added for padding is less than 256. Since the size of macro-blocks is arbitrary, 256 bytes could not be sufficient. We therefore propose a novel padding scheme that nicely fits the requirements of Mix&Slice. Our padding scheme presents similarities with well-known *ANSI X9.23* and *PKCS#7* padding techniques.

ANSI X9.23 [ANS88, Sch15]. This padding technique works with blocks of 8 bytes each and aims at padding blocks in such a way that the resulting number of bytes in a message is always a multiple of 8. Hence, the number of bytes added by ANSI X9.23 for padding is at most 8. The number of padding bytes can then always be represented in one byte. ANSI X9.23 uses the last byte to keep track of the number of padding bytes, while the remaining padding bytes are set to all 0's. Since the number of padding bytes is always represented in the last byte of the block, the removal of padding is deterministic. Indeed, the algorithm first reads the last byte to identify padding bytes that need to be ignored.

PKCS#7 [Kal98]. This padding technique, described in *RFC5652*, aims at padding messages that require the addition of at most 255 padding bytes. Indeed, independently from the number of bytes composing a block, if x bytes need to be added for padding purposes, all the padding bytes are set to value x. If no padding byte is needed, since the message content has a size that is a multiple of the block size, PKCS#7 requires the insertion of an additional block, to guarantee that the last byte always represents the number of padding bytes.

Since Mix&Slice does not impose any limit or constraint to the size *Msize* of macro-blocks, neither ANSI X9.23 nor PKCS#7 can be directly adopted. As a matter of fact, the number of padding bytes necessary for a resource encrypted with Mix&Slice can be as large as $Msize-1$ in the worst case and such a number of bytes might exceed 256, which is the maximum number that can be represented in one byte. Given the number *Msize* of bytes in a macro-block, we denote with *psize* the number of bytes necessary to represent the maximum number of padding bytes for a resource encrypted using Mix&Slice. Assuming to reserve *psize* bytes to the representation of the number of padding bytes, the maximum number of padding bytes in the macro-block is

$Msize + (psize - 1)$. The adopted padding scheme should, as a first step, compute the minimum number *psize* of bytes sufficient to represent value $Msize + (psize - 1)$.

If a resource occupies, in the last macro-block storing its content, at most $Msize - psize$ bytes, the number of padding bytes can be represented in the last *psize* bytes, while padding the remaining (empty) bytes. On the contrary, if a resource occupies, in the last macro-block storing its content, more than $Msize - psize$ bytes, the macro-block cannot accommodate the *psize* bytes necessary to the padding scheme. In such a case, our padding technique requires the use of an additional macro-block for storing the resource, to properly manage padding. Apart for the last *psize* bytes, the remaining padding bytes are set to all 0's. Note that even if, for a resource, the number of padding bytes could be represented in less than *psize* bytes, the last *psize* bytes in the last macro-block are always devoted to the representation of the number of padding bytes. This guarantees that the value of *psize* depends only on *Msize*, and hence that padding removal is deterministic and operates in the same manner for all the resources.

As a final note, since *psize* could be (and normally is) greater than 1, it is important to standardize the *endianness* of the padding value. We opted for a *big-endian* representation (i.e., most significant digits appear first).

2.4 Fragment management and key regression

When using Mix&Slice for resource encryption, secure resource deletion and policy updates are enforced by re-encrypting a randomly chosen fragment using an encryption key known only to the resource owner and to users authorized to read the resource content. To support secure deletion and policy updates, our tool implements fragment-level re-encryption using hardware accelerated AES-256 in counter mode.

The re-encryption of a fragment for resource deletion or policy update through our tool can be performed using the following command.

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

Each secure deletion and each policy update operation requires the use of a fresh new encryption key for fragment re-encryption. Therefore, due to policy changes, fragments of a resource might be encrypted with different keys. To limit the overhead of key management and distribution, our proposal is complemented with a convenient approach based on a *key regression* technique [FKK05]. Key regression is an RSA-based cryptographically strong technique (i.e., the generated keys appear as pseudorandom) allowing a resource 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 resource 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.

Let (n, e) be the public key known to all users and (n, d) be the private key known to the resource owner only. To generate a key sequence, the resource owner selects a seed s_0 in $\mathbb{Z}_N$. The byte representation of s_0 is then used as the base key k_0 for resource encryption. To generate a key k_{y+1} from a key k_y (with $y \geq 0$), the owner first computes $s_{y+1} = ((s_y)^d \mod n)$ by encrypting s_y with her own private key. Then k_{y+1} is the byte representation of s_{y+1}. Any user knowing a seed s_i

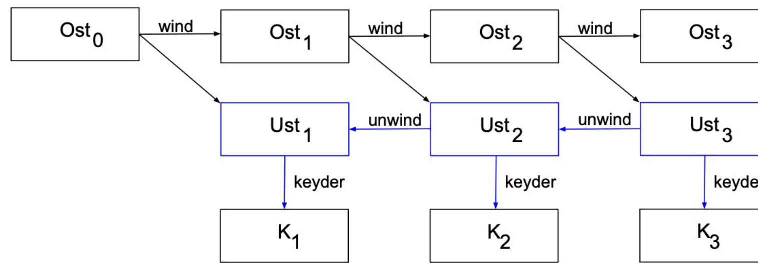


Figure 2.3: Overview of the key regression scheme

also knows key k_i. Also, she can derive any key k_j with $j < i$ by backward computing s_{i-1}=$((s_i)^e$ mod $n)$, until $i-1 = j$ (i.e., decrypting s_i with the public key of the resource owner).

Figure 2.3 illustrates key regression technique, where O_{st_i} and U_{st_i} represent the status information (including the seed and the RSA encryption key) for the owner and for the user, respectively. In the figure, `wind` and `unwind` operations correspond to the generation of a new status and the retrieval of a previously generated one, respectively. Note that only the resource owner can perform a `wind` operation, while both the resource owner and users knowing a status can perform an `unwind` operation to obtain a previous state from a more recent one. In the figure, `keyder` operation corresponds to the derivation of a key starting from the corresponding seed (i.e., its byte interpretation). The cost that users pay for key regression is small. On a single core, the computer we used for the experiments can process several hundred thousand key derivations per second.

Since, due to policy evolution, different fragments of a same resource might be re-encrypted with different keys, our implementation of Mix&Slice associates a meta-data structure with each resource. Such a structure keeps track, for each resource, of: the encryption key, the initialization vector, the list of fragments that have been re-encrypted, and the RSA key used for key regression. Hence the meta-data structure associated with a resource should be considered as confidential as the key itself and needs to be properly protected. The meta-data structure is stored in a file that is locally stored at the client side (i.e., on the resource owner and final user premises). The availability of such a file is necessary for encryption, decryption, deletion, and policy update operations.

To facilitate the enforcement of policy updates and resource deletion, our implementation of Mix&Slice generates a directory for each encrypted resource and stores each fragment as a separate file in the directory. Fragment deletion and/or re-encryption operates on a single file.

Given a resource that needs to be encrypted, our Python wrapper first pads it using the technique illustrated in Section 2.3. Then, it creates a new directory and uses the *cffi* library to invoke our C library implementing Mix&Slice to perform encryption. Similarly, to decrypt a resource our Python wrapper first reads the files representing the different fragments composing the resource from the corresponding directory and, using the meta-data file associated with the resource, retrieves the original representation of each encrypted fragment. Then, it invokes our C library to perform decryption, removes padding bytes, and finally reassembles the original plaintext resource.

2.5 Tool installation and usage

In this section, we discuss the installation and use of our Aesmix C library and of our Python wrapper.

2.5.1 Aesmix C library

Our Aesmix C-library is publicly available at `https://github.com/mosaicrown/aesmix`. In the following, we will describe the main commands for its installation and use.

Installation

Aesmix library requires, for its installation, the preliminary installation of `openssl/crypto` library and the `libtool` binary. In Ubuntu, this can be done using the following command.

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

To compile sources and install our Aesmix library (`libaesmix.so`) in your system, it is necessary to invoke the following command.

```
make
sudo make install
```

The command to remove the library is instead the following.

```
sudo make uninstall
```

The following commands lunch the test suite in single and multithreaded mode, respectively.

```
make test
make multitest
```

Usage

The availability of AES-NI hardware acceleration is automatically verified by Aesmix library. If AES-NI is available, Aesmix library relies on its functionalities to encrypt and decrypt resources, using the `includes/aes_mix.h` and `includes/aes_mix_multi.h` APIs.

In the following, we comment on the prototype of the multithreaded encryption method in Aesmix library and its parameters, which is one of the methods exposed by the library.

```
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);
```

The input parameters of the method represent:

- `thr`: the number of threads to use for encryption;
- `data`: pointer to the input buffer, storing the plaintext resource on which the method operates;
- `out`: pointer to the output buffer, where the encrypted result will be stored;
- `size`: number of bytes of the input and output buffers;
- `key`: symmetric key (string) used for the AES functions;
- `iv`: initialization vector for the AES functions.

2.5.2 Python wrapper

Our Python wrapper, providing command line access to Aesmix library, is available in the Python Package Index (PyPi[2]), a repository of software for Python programming language.

[2] https://pypi.org/

Installation

Similarly to Aesmix C-library, a preliminary step requires the installation of `openssl/crypto` library. The latest released version of such a library can be installed using `pip` through the following command.

```
pip install aesmix
```

Alternatively, to install the package in a virtual environment, it is necessary to enter the directory `aesmix/python/` and run the following command.

```
python setup.py install
```

Usage

After installation, Aesmix is available as a CLI tool under the `mixslice` alias. The help message associated with `mixslice` command provides a brief overview of the functionalities and usage of our tool, as illustrated in the following.

```
usage: mixslice [-h] {encrypt,update,decrypt} ...

Mix&Slice tool for encrypting/decrypting files

positional arguments: {encrypt,update,decrypt}

sub-command help
    encrypt             encrypt a file
    update              perform policy update
    decrypt             decrypt a file

optional arguments:
-h, --help            show this help message and exit
```

The following command is used to encrypt file `sample.txt`.

```
$ 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
```

The default configuration produces 1024 `sample.txt` fragments, which will be stored in `fragdir` directory. Also, the command produces two additional files, `.public` and `.private`, containing: the AES key, the initialization vector IV, and asymmetric RSA certificates necessary for key regression.

The following command is instead used to decrypt encrypted file `sample.txt.enc`.

```
$ 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
```

The command produces a decrypted file `sample.txt.enc.dec`, whose content can be hashed and compared with the original copy of the file `sample.txt`, as illustrated in the following example.

```
$ sha1sum sample.txt sample.txt.enc.dec
d3e92d3c3bf278e533f75818ee94d472347fa32a  sample.txt
d3e92d3c3bf278e533f75818ee94d472347fa32a  sample.txt.enc.dec
```

	Number of threads					
Hardware support	**1**	**2**	**4**	**8**	**16**	**32**
Yes	2.439567	1.416715	0.890562	0.466210	0.358425	0.281926
No	56.655915	31.374009	17.031515	8.652336	5.790404	5.004548

(a) mini-blocks of 32 bits

	Number of threads					
Hardware support	**1**	**2**	**4**	**8**	**16**	**32**
Yes	4.027704	2.368083	1.390318	0.751499	0.599889	0.432854
No	101.946789	62.814606	30.724053	16.609045	10.790716	8.550898

(b) mini-blocks of 64 bits

Figure 2.4: Time required to encrypt a 1GiB resource

2.6 Experimental results

Figure 2.4 illustrates the results of our experimental evaluation, aimed at comparing the performance of our Aesmix tool varying the number of concurrent threads (from 1 to 32) and the size of mini-blocks (32 or 64 bits). The experiments have been run in presence and absence of the hardware implementation of AES (e.g., AES-NI) to assess the performance advantage of the availability of the circuit. For our experiments, we considered macro-blocks of 16KB and a resource of 1GiB. The experiments have been performed over a client with Ubuntu 16.04, Dual Xeon and 256 GB RAM.

As visible from the figure, the availability of an architecture that provides a native hardware implementation for AES provides considerably better performance. Also, a higher number of threads permits to reduce the time necessary for encryption and decryption operations, reducing computational time of one order of magnitude when moving from 1 to 32 threads. We note that also the size of mini-blocks has an impact on computational times, with smaller mini-blocks implying reduced times.

2.7 Summary

In this chapter, we presented our implementation of the Mix&Slice encryption mode for resource protection in a data market environment. Our tool supports the encryption and decryption of resources of arbitrary size, while guaranteeing AONT property of encryption. Also, our implementation permits to efficiently enforce updates to the policy regulating access to the resources and to permanently delete resources from the store. Mix&Slice also represents the building block for relying on a DCS for storing resources in the data market. Part of the results presented in this chapter have been published in [BDF+20, BDF+19b].

3. Conclusions

This document reported on the implementation efforts associated with some of the novel techniques developed in WP4 for data wrapping. The developed tools provide the building blocks towards the realization of: *i)* a data processing framework able to enforce data protection requirements while considering performance, scalability, as well as economic costs, and *i)* an encryption layer of wrapping providing strong protection guarantees even in case of key leakage and enabling efficient policy update and secure resource deletion.

The tool described in Chapter 1 operates in a scenario where different providers can be involved in the storage and processing of possibly sensitive or company-confidential data. Authorizations associated with datasets regulate the visibility of data to different parties. Visibility can be either on plaintext data or on their encrypted representation. Each party is characterized by a distinct profile in terms of trust and cost. The tool enables then the consideration of the economical dimension in data processing, which has a clear role in the digital data market scenario. In general, we can expect that less expensive providers are also associated with lower trust and hence more restrictive authorization (e.g., restricted to operate on encrypted data as they are not trusted to know their plaintext value). The tool presented in this deliverable enables the identification of an optimal assignment of query plan operations to different parties, injecting encryption at query execution time to enable the involvement of less expensive providers while ensuring data confidentiality.

The tool described in Chapter 2 offers an efficient and adaptable implementation of the Mix&Slice approach, providing an *All-Or-Nothing-Transform* (AONT) encryption mode. The AONT implemented is robust against users who previously had access to resources and locally store, in addition to the cryptographic keys, some compact constant-size information. This information would allow the user to continue to have access to the resource even after a portion of the resource is made unavailable. Instead, with Mix&Slice, this threat is mitigated. The tool realizes a variant of Mix&Slice that enables the management of arbitrary size resources and leverages hardware implementation of AES, whenever available. Experimental results confirm the compatibility of the implemented approach with the requirements of large-scale data management architectures.

The tools presented in this deliverable demonstrate the applicability and working of the novel data wrapping techniques developed in MOSAICrOWN. The tools are available open source and are expected to become the foundation for further work.

Bibliography

[Ann01] Announcing the Advanced Encryption Standard. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, 2001.

[ANS88] ANSI X9.23, American National Standard for financial institution message encryption, 1988.

[BDF+16] 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 ACM CCS*, Vienna, Austria, October 2016.

[BDF+19a] 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 PerCom*, Kyoto, Japan, March 2019.

[BDF+19b] 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 IEEE GLOBECOM*, Waikoloa, HI, USA, December 2019.

[BDF+20] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Securing resources in decentralized cloud storage. *IEEE TIFS*, 15(1):286–298, December 2020.

[Cry] Crypto++. https://github.com/weidai11/cryptopp.

[DFJ+17] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. An authorization model for multi-provider queries. *Proc. of the VLDB Endowment*, 11(3):256–268, November 2017.

[EVP] EVP OpenSSL. https://wiki.openssl.org/index.php/EVP.

[FKK05] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. *IACR Cryptology ePrint Archive*, 2005:303, January 2005.

[FP20] S. Foresti and S. Paraboschi, editors. *D4.2 - Report on encryption-based techniques and policy enforcement*. Technical report of MOSAICrOWN, June 2020. (to appear).

[Gae12] H. Gael. Intel AES-NI. https://software.intel.com/en-us/node/256280#inpage-nav-3, 2012.

[Kal98] B. Kaliski. RFC2315: PKCS# 7: Cryptographic message syntax version 1.5, March 1998.

[Ope] OpenSSL wiki. `https://wiki.openssl.org`.

[Sch15] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C, 20th Anniversary Edition*. John Wiley & Sons, March 2015.

[SKS+19] G. Salvaneschi, M. Köhler, D. S., P. Haller, S. Erdweg, and M. Mezini. Language-integrated privacy-aware distributed queries. *PACMPL*, 3(OOPSLA):167:1–167:30, October 2019.

[TLL19] C. Thoma, A. Labrinidis, and A.J. Lee. Shoal: Query optimization and operator placement for access controlled stream processing systems. In *Proc. of DBSec*, Charleston, SC, USA, July 2019.