Cryptographic Enforcement of Access Control Policies in the Cloud: Implementation and Experimental Assessment

Stefano Berlato[1 2], Roberto Carbone[2] and Silvio Ranise[2 3]
[1] *DIBRIS, University of Genoa, Genoa, Italy*
[2] *Security and Trust Research Unit, Fondazione Bruno Kessler, Trento, Italy*
[3] *Department of Mathematics, University of Trento, Trento, Italy*
{*sberlato, carbone, ranise*}*@fbk.eu, silvio.ranise@unitn.it*

Keywords: Cryptographic Access Control, Experimental Assessment, Honest but Curious Cloud Service Provider.

Abstract: While organisations move their infrastructure to the cloud, honest but curious Cloud Service Providers (CSPs) threaten the confidentiality of cloud-hosted data. In this context, many researchers proposed Cryptographic Access Control (CAC) schemes to support data sharing among users while preventing CSPs from accessing sensitive data. However, the majority of these schemes focuses on high-level features only and cannot adapt to the multiple requirements arising in different scenarios. Moreover, (almost) no CAC scheme implementation is available for enforcement of authorisation policies in the cloud, and performance evaluation is often overlooked. To fill this gap, we propose the toolchain COERCIVE, short for CryptOgraphy killEd (the honest but) cuRious Cloud servIce proVidEr, which is composed of two tools: TradeOffBoard and CryptoAC. TradeOffBoard assists organisations in identifying the optimal CAC architecture for their scenario. CryptoAC enforces authorisation policies in the cloud by deploying the architecture selected with TradeOffBoard. In this paper, we describe the implementation of CryptoAC and conduct a thorough performance evaluation to demonstrate its scalability and efficiency with synthetic benchmarks.

1 Introduction

Cryptographic Access Control (CAC) allows organisations to enforce Access Control (AC) policies over cloud-hosted sensitive data while guaranteeing strong confidentiality, i.e., neither external attackers nor Cloud Service Providers (CSPs) can read the data. Several researchers proposed CAC schemes using different cryptographic primitives, like Attribute-Based Encryption (ABE) (Jang-Jaccard, 2018), hybrid cryptography (Garrison et al., 2016), proxy re-encryption (Rezaeibagha and Mu, 2016) and onion encryption (Qi and Zheng, 2019).

While attaining remarkable qualities on paper, the majority of these CAC schemes seldom considers a concrete use in a specific scenario. As the focus is usually on high-level features only, little space is left for those aspects (e.g., flexibility, portability) related to CAC scheme deployments. In particular, the architecture (i.e., the definition of the entities that compose a scheme along with their logical or physical locations) is usually not provided, or it is fixed and cannot accommodate the requirements (e.g., enhance architecture scalability or minimize the vendor lock-in effect) of different scenarios. Moreover, few researchers provided even a prototype implementing their scheme, often just for measuring the performance of read and write operations on data. Finally, such prototypes usually can interact with one CSP only (e.g., Amazon Web Services (AWS), Google Cloud Platform (GCP) or Azure). In other words, there is a gap between CAC schemes in the abstract and a concrete approach for real-world deployment.

To fill this gap, we propose the toolchain COERCIVE—open-source and available online[1]—which is composed of two tools for achieving optimal enforcement of role-based CAC policies in the cloud. The first tool, TradeOffBoard, is a dashboard intended to help organisations in finding the best architecture for deployment of CAC schemes. We highlight that TradeOffBoard is already extensively described in (Berlato et al., 2020) as a "Web Dashboard" and it is not a novel contribution of this paper. Instead, we only illustrate its use in the context of COERCIVE. Our main contribution lies instead in the implementation and thorough performance evaluation of the second tool, CryptoAC. CryptoAC is a Swiss

[1] https://github.com/stfbk/CryptoAC

army knife tool for flexible and portable enforcement of role-based CAC policies in the cloud. CryptoAC implements the CAC scheme proposed by (Garrison et al., 2016). Moreover, CryptoAC supports dozens of different architectures seamlessly and it is cloud-independent, i.e., it can be deployed in any CSP without needing extra configuration.

The paper is structured as follows. In Section 2 we report the background. Then, in Section 3 we give an overview of COERCIVE and motivate its need by showing how it can be used, together with TradeOffBoard, in an eHealth scenario often considered in cloud-related literature. Afterwards, we describe design and implementation concerns of CryptoAC in Section 4. To demonstrate the feasibility of our approach, we provide a thorough performance evaluation for CryptoAC to study its scalability and asymptotic behaviour with synthetic benchmarks in Section 5. Finally, we report related work in Section 6 and conclude the paper in Section 7.

2 Background

In this section, we introduce the concepts of AC (Section 2.1) and CAC (Section 2.2).

2.1 Access Control

In (Samarati and de Capitani di Vimercati, 2000), AC is defined as "the process of mediating every request to resources maintained by a system and determining whether the request should be granted or denied". Resources usually consist of data such as files and documents. Role-Based Access Control (RBAC) is one of the most widely adopted AC models for cloud computing (Cai et al., 2019). In RBAC, *users* are assigned to one or more *roles*. In an organization, a *role* reflects an internal qualification (e.g., employee). A *permission* $p = \langle res, op \rangle$ is defined as an operation *op* (e.g., read) over a resource *res* (e.g., a file). *Permissions* are assigned to one or more *roles* by administrators of the policy. *Users* activate a *role* to access the *permissions* needed to finalize their operations on resources (e.g., read a file). Formally, the state of an RBAC policy can be described by the set of users U, roles R, permissions P and the assignments *users-roles* $UR \subseteq U \times R$ and *roles-permissions* $PA \subseteq R \times P$. A user u can use a permission p if $\exists r : (\langle u, r \rangle \in UR) \wedge (\langle r, p \rangle \in PA)$. Role hierarchies can always be compiled away by adding suitable pairs to *UR*.

Table 1: CAC schemes entities.

Entity	Description
Proxy	Performs cryptographic computations, allowing users to access files and the administrator to manage the AC policy
RM	Mediates users' requests to add and write files by ensuring compliance with the AC policy
MS	Stores metadata
DS	Stores encrypted data

2.2 Cryptographic Access Control

Partially trusted environments (e.g., the cloud) preserve the integrity and availability of data but not their confidentiality (Garrison et al., 2016). In this case, cryptography is often used to enforce AC while ensuring the confidentiality of sensitive data. These data are encrypted and the permission to read the encrypted data is embodied by the secret decrypting keys. While implying a further computational burden (i.e., the cryptographic computations) and additional metadata (e.g., public keys), CAC allows encrypted data to be stored in partially trusted environments. Usually, CAC schemes involve four entities (Berlato et al., 2020), i.e., a proxy, a Reference Monitor (RM), a Metadata Storage (MS) and a Data Storage (DS), which we describe in Table 1

Below, we give an intuition of the functioning of the CAC scheme proposed in (Garrison et al., 2016) for enforcing role-based CAC policies. Each user u and role r is provided with a pair of secret and public keys $(\mathbf{k}^{\mathbf{s}}_{u}, \mathbf{k}^{\mathbf{p}}_{u})$ and $(\mathbf{k}^{\mathbf{s}}_{r}, \mathbf{k}^{\mathbf{p}}_{r})$, respectively. The content f of each file *fn* is encrypted with an individual (i.e., different for each file) symmetric key $\mathbf{k}^{\mathbf{sym}}_{fn'}$. To assign u to r, r's secret key $\mathbf{k}^{\mathbf{s}}_{r}$ is encrypted with $\mathbf{k}^{\mathbf{p}}_{u}$, resulting in $\{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}}$. To give permission over a file *fn*, the symmetric key $\mathbf{k}^{\mathbf{sym}}_{fn}$ related to the file is encrypted with $\mathbf{k}^{\mathbf{p}}_{r}$, resulting in $\{\mathbf{k}^{\mathbf{sym}}\}_{\mathbf{k}^{\mathbf{p}}_{r}}$. The use of both asymmetric and symmetric cryptography is usually called "Hybrid Encryption" (Garrison et al., 2016). An operation *op* can be either read (R) or read-write (RW). The CAC policy is represented by decorated versions of the relations *UR* and *PA*, i.e. $\{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}}$ is attached to the pair $\langle u, r \rangle$ in *UR* whereas $\{\mathbf{k}^{\mathbf{sym}}\}_{\mathbf{k}^{\mathbf{p}}_{r}}$ is attached to the pair $\langle r, p \rangle$ in *PA*; by abusing notation, we write $\langle u, r, \{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}} \rangle$ and $\langle r, p, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle$, respectively and, together with public cryptographic keys, we refer to these extended tuples as metadata. In (Garrison et al., 2016), metadata contain additional information (e.g., digital signatures for authenticity and integrity) that we omit here for the sake of simplicity.

Besides read and write operations, the CAC scheme in (Garrison et al., 2016) specifies also administrative operations for managing the AC policy such as adding and deleting users, roles, files and assign-

Table 2: CAC scheme operations.

Operation	Description
addU(u):	add u in U
deleteU(u):	delete u from U; $\forall r \in R : \exists\langle u, r, \{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}} \rangle \in UR$, *revokeU(u, r)*
addR(r):	add r in R
deleteR(r):	delete r from R; $\forall fn : \exists\langle r, \langle fn, *\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle \in PA$, *revokeP$_{RW}$(r,fn)*
addF(fn, f):	add $\langle fn, *\rangle$ in P and f in DS
deleteF(fn):	delete $\langle fn, *\rangle$ from P and f from DS; delete all $\langle *, \langle fn, *\rangle, *\rangle$ from PA
assignU(u, r):	add $\langle u, r, \{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}} \rangle$ in UR
revokeU(u, r):	delete $\langle u, r, \{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}} \rangle$ from UR; update r's keys in all $\langle *, r, *\rangle \in UR$; update fn's keys in all $\langle *, \langle fn, *\rangle, *\rangle \in PA : \exists\langle r, \langle fn, *\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle \in PA$
assignP(r,fn, op):	add $\langle r, \langle fn, op\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle$ in PA
revokeP$_W$(r,fn):	change $\langle r, \langle fn, RW\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle$ to $\langle r, \langle fn, R\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle$
revokeP$_{RW}$(r,fn):	delete all $\langle r, \langle fn, *\rangle, \{\mathbf{k}^{\mathbf{sym}}_{fn}\}_{\mathbf{k}^{\mathbf{p}}_{r}} \rangle$ from PA; update fn's keys in all $\langle *, \langle fn, *\rangle, *\rangle \in PA$
readF(fn):	decrypt $\{\mathbf{k}^{\mathbf{s}}_{r}\}_{\mathbf{k}^{\mathbf{p}}_{u}}$ with $\mathbf{k}^{\mathbf{s}}_{u}$, then $\{\mathbf{k}^{\mathbf{sym}}\}_{\mathbf{k}^{\mathbf{p}}_{r}}$ with $\mathbf{k}^{\mathbf{s}}_{r}$ and finally $\{f\}_{\mathbf{k}^{\mathbf{sym}}}$ with $\mathbf{k}^{\mathbf{sym}}$
writeF(fn, f′):	encrypt f' with $\mathbf{k}^{\mathbf{sym}}$

ing and revoking users and permissions to roles. We briefly describe all operations in Table 2. We use the character "*" as a wildcard in tuples for *UR* and *PA* assignments. For instance, $\{\langle *, r, *\rangle \in UR\}$ represents the set of *UR* assignments related with r (i.e., the set of users assigned to r). We highlight that, whenever *RW* permission over a file *fn* is revoked from a role r (*revokeP$_{RW}$(r,fn)* operation), the related key $\mathbf{k}^{\mathbf{sym}}_{fn}$ is discarded and a new key $\mathbf{k}^{\mathbf{sym'}}_{fn}$ is generated. This mechanism prevents r from still being able to decrypt the content f of *fn*. The same behaviour applies whenever a user u is revoked from a role r (*revokeU(u, r)* operation), i.e., r's keys are renewed to prevent u from still having access to r's permissions. As u may have cached keys for future access, also the keys of all files r has permission over are replaced. For better performance, the content f of a file *fn* is not immediately re-encrypted (i.e., active re-encryption) but instead it is re-encrypted with the new key $\mathbf{k}^{\mathbf{sym'}}_{fn}$ by the next user to write to the file (i.e., lazy encryption). For more details about the construction of the CAC scheme, please refer to (Garrison et al., 2016).

3 COERCIVE Overview

In this section, we give an overview of the toolchain COERCIVE (see Figure 1). We first motivate its need by describing an eHealth scenario often considered in the cloud-related literature (Section 3.1). Then, we show how to apply TradeOffBoard in such a scenario (Section 3.2). We highlight that CryptoAC is instead extensively described in Section 4.

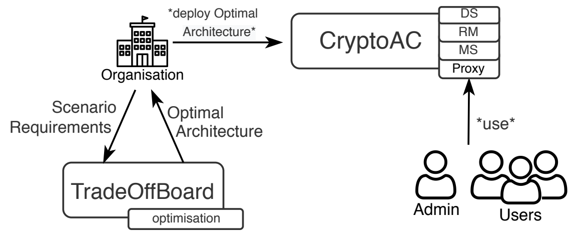


Figure 1: COERCIVE overview.

3.1 The eHealth Scenario

We start from the consideration that different scenarios may have specific requirements to satisfy. At a high-level, COERCIVE allows organisations to evaluate CAC scheme architectures against these requirements (TradeOffBoard box in Figure 1) and then deploy the optimal architecture for the scenario under consideration (CryptoAC box in Figure 1).

To understand why these are crucial activities, we consider the following scenario adapted from the literature on outsourcing medical data to the cloud (Horandner et al., 2016; Sato and Fugkeaw, 2015; Premarathne et al., 2016). Suppose a person with a mental disorder is hospitalized in a specialized clinic. The clinic is storing in the cloud the patients' data encrypted under a CAC scheme. However, the CAC scheme expects the patient's name to be part of the metadata (e.g., in the AC policy or in the files name); the CSP may then infer that a specific person is a customer of the clinic and share this information for targeted advertisement. This is an instance of a well-known requirement (see, e.g., (Domingo-Ferrer et al., 2019)) which states to hide not only medical data but also metadata from CSPs. Moreover, this may not be the only requirement to consider; others may include the use of hardened devices only for processing medical data, the prioritization of redundancy to avoid data loss, the maximization of CSP (monetary) savings and the minimization of the vendor lock-in.

However, different architectures satisfy these requirements at different levels. Even worse, these requirements may conflict with each other, leading to a stalemate in which we need to carefully evaluate the benefits and drawbacks of different architectures. For instance, while enabling ubiquitous access, allowing employees to use personal computers is against the requirement of using hardened devices only. Then, the setup of a data-centre on-premise may be unappeal-

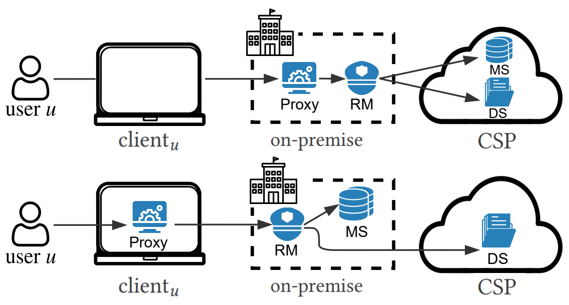
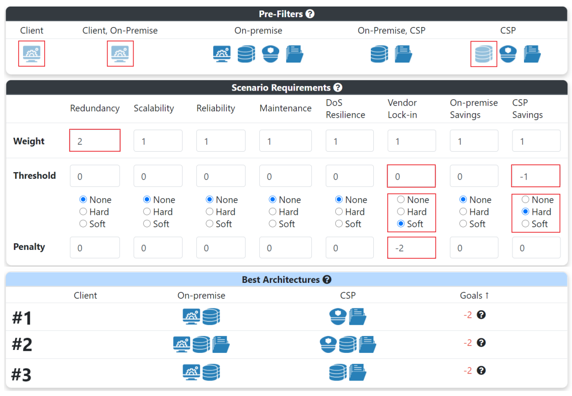


Figure 2: Two architectures for CAC schemes in the cloud.

ing for small clinics, for they may lack the skills and resources to efficiently manage it. Although moving some services to the cloud may enhance redundancy and scalability, it would threaten (sensitive) metadata and suffer from high CSP-related monetary costs and the vendor lock-in effect. In this context, COERCIVE supports the investigation of alternative CAC architectures (by using TradeOffBoard) and then deploy the selected one (by using CryptoAC).

3.2 TradeOffBoard Optimisation

To accomplish its tasks, a CAC entity (i.e., proxy, RM, MS or DS) must be deployed in a location or "domain" (e.g., software needs to run on a machine). For CAC schemes targeting cloud-hosted data, there exist three possible domains (Berlato et al., 2020), i.e., client$_u$ (e.g., the user u's computer), on-premise (e.g., a data-centre within the organisation) and the CSP. It naturally follows that there are multiple ways to assign entities to domains, i.e., there are several architectures for CAC schemes (e.g., see Figure 2 for two examples). An architecture may have multiple instances of the same entity under different domains. Moreover, an architecture does not define which CSP to use (e.g., AWS, Azure, GCP).



Pre-Filters

Client	Client, On-Premise	On-premise	On-Premise, CSP	CSP

Scenario Requirements

	Redundancy	Scalability	Reliability	Maintenance	DoS Resilience	Vendor Lock-in	On-premise Savings	CSP Savings
Weight	2	1	1	1	1	1	1	1
Threshold	0	0	0	0	0	0	0	-1
	[✓ None] Hard Soft	[✓ None] Hard Soft	[✓ None] Hard Soft	[✓ None] Hard Soft	[✓ None] Hard Soft	None Hard [✓ Soft]	[✓ None] Hard Soft	None [✓ Hard] Soft
Penalty	0	0	0	0	0	-2	0	0

Best Architectures

	Client	On-premise	CSP	Goals ↑
#1				-2
#2				-2
#3				-2

Figure 3: TradeOffBoard screenshot.

As an illustrative example, we input in TradeOffBoard the requirements of the eHealth scenario described in Section 3.1 (see Figure 3). TradeOffBoard presents a first block "Pre-Filters" in which an organisation can exclude the presence of an entity in a domain. For the eHealth scenario, we set the Pre-Filters so to avoid the deployment of the proxy in the client$_u$ domain (to avoid personal devices) and the MS in the CSP domain only (to avoid the disclosure of sensitive metadata to the CSP). Then, the "Scenario Requirements" block allows prioritizing a set of relevant architectural requirements (e.g., reliability, maintenance) identified in (Berlato et al., 2020). This set is not fixed, and requirements can be added and removed according to the scenario under investigation. Following the description of the eHealth scenario in Section 3.1, we prioritize redundancy by assigning a higher weight with respect to other requirements. Then, we set a (soft) constraint on the vendor lock-in requirement (i.e., to allow architectures suffering from the vendor lock-in effect at the cost of a penalty). Finally, we set a (hard) constraint on CSP (monetary) savings (i.e., to exclude architectures leading to significant CSP related expenses).

Based on how each architecture satisfies these requirements, TradeOffBoard solves the optimisation problem and proposes a ranking of the architectures in the "Best Architectures" block. For the eHealth scenario, TradeOffBoard proposes three best architectures. These avoid the use of the client$_u$ domain as expected by the Pre-Filters. Moreover, the MS is either deployed in the on-premise domain only (architecture #1) or in a hybrid fashion between the on-premise and the CSP domains (architectures #2 and #3) so to always protect sensitive metadata. To enhance redundancy, the DS is always deployed in the CSP domain. Nonetheless, to limit the vendor lock-in effect and CSP monetary expenses, entities deployment is balanced between the on-premise and the CSP domains. As a final remark, we highlight that TradeOffBoard solves the optimisation problem in few milliseconds. For more details on TradeOffBoard and its performance evaluation, we refer the interested reader to (Berlato et al., 2020).

4 CryptoAC

As highlighted in Section 1, the (few) prototypes implementing CAC schemes available in the literature can usually interact with one CSP only and have a fixed (or unspecified) architecture; this makes them unsuitable for real-world deployment, as they lack in particular portability (across different CSPs) and (architectural) flexibility.

To fill this gap, we propose CryptoAC, a fully working and usable implementation of the CAC scheme proposed by (Garrison et al., 2016). Concretely, CryptoAC is composed of four software modules (see Table 3), one for each CAC scheme entity presented in Section 2.2 (i.e., proxy, RM, MS and DS). We highlight that we followed development best practices (e.g., Test-Driven Development, Continuous Integration pipelines) for producing quality software.

In this section, we first explain the use of container technology to achieve cloud-independency (Section 4.1). Then, we describe the implementation of the proxy (Section 4.2), RM (Section 4.3), MS (Section 4.4) and DS (Section 4.5) modules. Finally, we discuss CryptoAC multiple architectures support and deployment (Section 4.6). For the sake of brevity, here we do not discuss security aspects not relevant research-wise such as the use of TLS for establishing secure channels or of cryptographic certificates for mutual authentication between CryptoAC modules.

4.1 Cloud-Independency

We choose to use the microservice architecture,[2] that consists of logically splitting an application into loosely coupled services (usually called containers), where each service is self-contained and implements a single functionality with clear interfaces.

To enable the microservice paradigm in our context, we ship our modules in Docker images.[3], a read-only templates containing instructions to create containers, which run on the Docker platform in an isolated environment. All major CSPs provide support for running containers (e.g., AWS,[4] Azure[5] and GCP[6]). Using Docker, CryptoAC achieves cloud-independency and can therefore be used in any (combination) of these CSPs.

As it usually happens, CryptoAC Docker images are built on other base images, with some additional customization. The Docker image of the MS module is built on top of the official MySQL database image over which we provide an `.sql` file for creating the database (e.g., tables, views, triggers) of metadata. The remaining three entities (i.e., proxy, RM and DS) cannot be implemented just as easily since they need to support particular functionalities. To avoid code duplication and enhance maintainability, we implement all three entities in a single Java program. Its Docker image is built on top of the official OpenJDK

[2] https://developer.ibm.com/articles/why-should-we-use-microservices-and-containers

[3] https://www.docker.com/ [4] https://aws.amazon.com/ecs/?nc1=h_ls

[5] https://azure.microsoft.com/en-gb/product-categories/containers/

[6] https://cloud.google.com/sdk/gcloud/reference/container

Table 3: CryptoAC software modules.

Entity	Base Docker	Implementation	HTTP RESTful APIs
Proxy	OpenJDK	Java Program	22 (10 profiles,12 AC)
RM	OpenJDK	Java Program	3 (add/write file, configure)
MS	MySQL	`.sql` file	- (MySQL protocol)
DS	OpenJDK	Java Program	4 (CRUD Paradigm)

Docker image. At startup, each container is configured with an operation mode that specifies which entity (i.e., either proxy, RM or DS) the instance should support. Each container can support one entity at a time. Besides, it is possible to fine-tune several parameters (e.g., cryptographic algorithms, key size) directly from the command line.[7]

4.2 Proxy

The proxy module interfaces users with data (e.g., through read and write requests, as explained in Section 2.2) by performing cryptographic computations. Besides users, the administrator uses the proxy module to manage the AC policy. In particular, the administrator can create and delete users, roles, files and distribute assignments and permissions, i.e., the administrator can modify U, R, PUR and PA by sending an (authenticated and signed) request to the MS module. The proxy module has to be configured with relevant information to interact with the other modules (e.g., URLs, cryptographic certificates, credentials).

Each operation performed by a user u (e.g., write a file) or the administrator (e.g., assign a user to a role) is digitally signed with $\mathbf{k}_u^\mathbf{s}$ (or, since we are considering RBAC, with the private signing key of the role $\mathbf{k}_r^\mathbf{s}$ that was assumed by the user to perform the operation). Digital signatures protect encrypted data and metadata against accidental or malicious modifications by providing integrity and authenticity. Users, roles and files have an identifier (i.e., username, role name and file name) and a random token of 50 bytes acting as a pseudonym. When a user digitally signs an operation, he/she appends his/her (or the assumed role's) token instead of his/her (or the role's) identifier. In this way, users can fetch public signing keys to verify signatures without learning the identity of the user (or role) that created the signature. Finally, users and roles are assigned a status, either "incomplete" (i.e., created but not fully configured), "operational" (i.e., ready for use) or "deleted" (by the administrator). Public cryptographic keys of deleted users and roles are kept to verify digital signatures.

[7] https://github.com/stfbk/CryptoAC

Web Application. The proxy module offers a web User Interface (UI), developed with JQuery,[8] Bootstrap[9] and the Apache Template Engine.[10] The proxy module uses the Spark Java framework[11] to setup a web server and expose two sets of RESTful APIs. The first set is related to the management of users' profiles; a user's profile contains the user's secret cryptographic keys (generated within the proxy) along with other configurations data (set through the UI) like username, password and URLs of other modules. The second set is related to the management of the AC policy; these APIs have a 1-to-1 mapping with administrative operations, i.e., add and delete users, roles and files, read and write files and distribute and revoke assignments and permissions. These operations are available via a UI for administrators, which presents only add, read and write operations to normal users. All the inputs to the web UI are validated against OWASP-approved regular expressions[12] to avoid web-based attacks (e.g., injection, Cross-Site Scripting).

4.3 Reference Monitor

When CryptoAC is configured to act as a RM, it registers three APIs. The first one is reserved for the administrator and allows to configure the RM module (e.g., by providing URLs of other modules). The other two APIs are for adding and writing files, respectively. When a user wants to add a new file, the RM module checks whether the user is granting to the administrator all permissions over the file. To counter malicious users who keep adding useless files to exhaust resources (e.g., storage space), we can simply implement a mechanism to limit the number (or size) of files each user can upload. When a user u wants to write over a file, the RM queries the AC policy in the MS to ensure that u has the permission to do it. While it would provide two layers of security, we follow the scheme in (Garrison et al., 2016) and do not make the RM module mediate read requests too, as this control is cryptographically enforced (see Section 2.2). Again, to counter malicious users and Denial-of-Service attacks, we can implement a mechanism to limit the number of files a user can upload or send a warning to the administrator.

4.4 Metadata Storage

When starting the MS module, we initialise the MySQL database by creating tables "users" for U, "roles" for R, "roleTuples" for UR and "permissionTuples" for PA. Furthermore, we create two additional tables to contain file-related metadata like status flags and version numbers, i.e., "files" and "fileTuples".[13] To hide (potentially sensitive) identifiers and avoid the disclosure of the AC policy to the users, we limit users' `Select` privileges to grant access to tokens only and employ views and row-level permissions. In practice, we define views over the "roleTuples" and "fileTuples" tables which automatically filter assignments not associated with the user querying the database, whose username is available through the `USER()` MySQL function.[14] In this way, each user knows his portion of the AC policy only and we respect the need-to-know principle (i.e., each user has access only to the information strictly needed to accomplish his task).

Finally, to avoid SQL injection and (stored) Cross-Site Scripting attacks, inputs to the database are sanitized with the `PreparedStatement` Java class,[15] while outputs from the database are sanitized using the OWASP Java Encoder.[16]

4.5 Data Storage

The DS module registers four APIs following the Create, Read, Update and Delete (CRUD) paradigm. Normal users can invoke the Read API, while Create, Update and Delete APIs can be invoked by the administrator and the RM module only. The CAC scheme in (Garrison et al., 2016) expects both new and overwritten (encrypted) files to be uploaded from the proxy to the RM. Then, the RM has to check the legitimacy of users' requests and, if everything is confirmed, send the files to the DS. However, we note that this two-hop approach may waste bandwidth and create a bottleneck in the RM module. As such, we modify this behaviour so that files are instead uploaded from the proxy module to the DS module and stored in a temporary upload directory. Then, the proxy asks the RM module to check the users' requests and, if everything is confirmed, the RM module asks the DS module to move the files in the actual storage directory from which users can download them. Otherwise, the files are simply discarded.

4.6 Multiple Architectures Support

CryptoAC initially inherits the ability to support multiple architectures from the CAC scheme we chose to

[8] https://jquery.com/
[9] https://getbootstrap.com/
[10] https://velocity.apache.org/
[11] http://sparkjava.com/
[12] https://owasp.org/www-community/OWASP_Validation_Regex_Repository
[13] https://github.com/stfbk/CryptoAC
[14] https://dev.mysql.com/doc/refman/8.0/en/information-functions.html
[15] https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
[16] https://owasp.org/owasp-java-encoder/

implement (Garrison et al., 2016). Indeed, the authors proved that in their scheme the proxy entity can be deployed both in the client$_u$ and on-premise domains. Moreover, they assumed the RM, DS and MS entities to stay in the CSP domain; it follows that deploying these entities on-premise instead is possible, as they would move from a partially trusted domain (i.e., CSP) to a fully trusted one (i.e., on-premise).

We preserve support for multiple architectures by making the proxy, RM and DS modules expose RESTful APIs (documented with Swagger OpenAPI) returning JSON-formatted responses to guarantee maximum flexibility. By defining clear interfaces for accessing inner functionalities, each module is self-contained and independent from the location of other modules. Besides, this approach allows CryptoAC to be easily integrated with other services (e.g., as a plugin for a text editor). Finally, we implement the proxy module so that one instance can serve multiple users (i.e., the proxy module can be deployed both in the client$_u$ and on-premise domains). By effectively decoupling the deployment from the implementation, CryptoAC results to be fully agnostic with respect to the chosen architecture, i.e., the code of CryptoAC requires no modification when supporting different architectures.

Deployment. An organisation can easily use available tools such as Cloudify,[17] Kubernetes[18] or Docker-compose[19] to deploy CryptoAC, based on the chosen architecture. As a proof-of-concept, we propose a simple docker-compose file for quick local deployment and make it available along with the source code of CryptoAC.[20] We note that the local deployment can automatically be ported to the cloud using new docker-compose features for integration with AWS and Azure. Finally, as we employ the Data Access Object (DAO) pattern for decoupling the logic of the CAC scheme from the management (e.g., storage, retrieval) of metadata and encrypted data, CryptoAC modules can be easily replaced by other services. For instance, the code of the RM module could be ported in an AWS Lambda service, while AWS RDS could be used as MS and AWS S3 as DS.

5 Performance Evaluation

In this section, we propose a thorough performance evaluation for CryptoAC. First, we measure the execution time of each operation described in Table 2 (Section 5.1). However, the performance of some operations is not constant, but instead it is influenced by the state of the AC policy. For instance, whenever the administrator revokes a permission from a role r over a file fn, the related key $\mathbf{k}_{fn}^{\mathbf{sym}}$ is discarded and a new key $\mathbf{k}_{fn}^{\mathbf{sym}'}$ is generated (see Section 2.2). However, the new key $\mathbf{k}_{fn}^{\mathbf{sym}'}$ has to be distributed to (i.e., encrypted with the public key of) all other roles which still have access to fn. Depending on the number of these roles, this operation may take more or less time. Therefore, we also explore more in detail the performance of those operations whose execution time depends on the state of the AC policy (Section 5.2).

5.1 Evaluation Per Operation

We report in Table 4 the performance evaluation of CryptoAC for each operation described in Table 2. We omit the operations for deleting users and roles as their timings can be derived in a straightforward way from *revokeU(u,r)* and *revokeP$_{RW}$(r,fn)*, respectively. We use the symbol $|\cdot|$ for representing the cardinality of a set (i.e., the number of its elements). We run all tests 1,000 times to reduce measurements errors on a Ubuntu 18.04 Virtual Machine with 4 threads on an Intel(R) Core(TM) i7-7700HQ and 8GB of RAM.

In the "Cryptographic Computations" column, we list the number and type of cryptographic computations involved in each operation, as described in (Garrison et al., 2016), namely generation, encryption, and decryption for symmetric-key ($\mathbf{Gen^S}, \mathbf{Enc^S}, \mathbf{Dec^S}$) and public-key ($\mathbf{Gen^P}, \mathbf{Enc^P}, \mathbf{Dec^P}$) cryptography together with generation and verification of digital signatures ($\mathbf{Gen^{Sig}}, \mathbf{Ver^{Sig}}$). Asymmetric keys are RSA 2048, while symmetric keys are AES 256. $\mathbf{Enc^P}$ and $\mathbf{Dec^P}$ encrypt and decrypt a pair of asymmetric keys (4096 bits) while the execution times of $\mathbf{Enc^S}$ and $\mathbf{Dec^S}$ are calculated per MB of data. Signatures are 256 bytes long and are generated from 16,743 bytes of data (i.e., the biggest block of metadata that is produced (Garrison et al., 2016)). We measure the execution time of each computation individually using the default `SUN` Java cryptographic provider[21] and report the results in Table 5. In the "Cryptographic Time" column, we calculate the execution time of each operation by summing the execution time of all cryptographic computations involved (taken from Table 5). For instance, the *addU(u)* operation involves the generation of two pairs of asymmetric keys (for encryption and digital signatures). As the execution time for $\mathbf{Gen^P}$ is 112.673ms, the execution time of *addU(u)* is $2 \cdot 112.673$ ms = 225.346 ms. The performance

[17] https://cloudify.co/ [18] https://kubernetes.io/
[19] https://docs.docker.com/ [20] https://github.com/stfbk/CryptoAC

[21] https://docs.oracle.com/en/java/javase/11/security/oracle-providers.html

Table 4: Cryptographic computations per operation and execution time (in milliseconds).

Operation	Cryptographic Computations	Cryptographic Time	Implementation Time
addU(u):	$2 \cdot \mathbf{Gen^P}$	225.346	276.612
addR(r):	$2 \cdot (\mathbf{Gen^P} + \mathbf{Enc^P}) + \mathbf{Gen^{Sig}}$	227.836	255.192
addF(fn, f):	$2 \cdot (\mathbf{Gen^{Sig}} + \mathbf{Ver^{Sig}}) + \mathbf{Gen^S} + \mathbf{Enc^S} + \mathbf{Enc^P}$	4.300 + $\mathbf{Enc^S}$	59.804 + $\mathbf{Enc^S}$
deleteF(fn):	No cryptographic computation involved	0	45.817
assignU(u, r):	$2 \cdot (\mathbf{Enc^P} + \mathbf{Dec^P}) + \mathbf{Gen^{Sig}} + \mathbf{Ver^{Sig}}$	26.731	68.770
revokeU(u, r):	$\mathbf{Dec^P} + (2 \cdot \mathbf{Enc^P} + \mathbf{Gen^{Sig}} + \mathbf{Ver^{Sig}}) \cdot \lvert\{\langle *, r\rangle \in UR\}\rvert + (\mathbf{Gen^S}) \cdot \lvert\{\langle r, *\rangle \in PA\}\rvert + (\mathbf{Enc^P} + \mathbf{Gen^{Sig}} + \mathbf{Ver^{Sig}}) \cdot \lvert\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}\rvert + 2 \cdot \mathbf{Gen^P}$	$(2.759) \cdot \lvert\{\langle *, r\rangle \in UR\}\rvert + (0.033) \cdot \lvert\{\langle r, *\rangle \in PA\}\rvert + (2.342) \cdot \lvert\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}\rvert + 237.33$	$(19.423) \cdot \lvert\{\langle *, r\rangle \in UR\}\rvert + (9.872) \cdot \lvert\{\langle r, *\rangle \in PA\}\rvert + (6.791) \cdot \lvert\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}\rvert + 247.735$
assignP(r, fn, ∗):	if $\exists \langle r, \langle fn, *\rangle\rangle \in PA$, $\mathbf{Gen^{Sig}} + 2 \cdot \mathbf{Ver^{Sig}}$; else, $\mathbf{Gen^{Sig}} + 2 \cdot \mathbf{Ver^{Sig}} + \mathbf{Enc^P} + \mathbf{Dec^P}$	if $\exists \langle r, \langle fn, *\rangle\rangle \in PA$, 2.194; else, 14.597	if $\exists \langle r, \langle fn, *\rangle\rangle \in PA$, 7.385; else, 49.167
revokeP$_W$(r, fn):	$2 \cdot (\mathbf{Ver^{Sig}} + \mathbf{Gen^{Sig}})$	3.850	45.674
revokeP$_{RW}$(r, fn):	$\mathbf{Gen^S} + (\mathbf{Ver^{Sig}} + \mathbf{Enc^P} + \mathbf{Gen^{Sig}}) \cdot \lvert\{\langle r', \langle fn, *\rangle\rangle \in PA : r \neq r'\}\rvert$	$(2.342) \cdot \lvert\{\langle r', \langle fn, *\rangle\rangle \in PA : r \neq r'\}\rvert + 0.033$	$(37.360) \cdot \lvert\{\langle r', \langle fn, *\rangle\rangle \in PA : r \neq r'\}\rvert + 0.034$
readF(fn):	$3 \cdot \mathbf{Ver^{Sig}} + 2 \cdot \mathbf{Dec^P} + \mathbf{Dec^S}$	24.779 + $\mathbf{Dec^S}$	45.559 + $\mathbf{Dec^S}$
writeF(fn, f′):	$5 \cdot \mathbf{Ver^{Sig}} + 2 \cdot (\mathbf{Dec^P} + \mathbf{Gen^{Sig}}) + \mathbf{Enc^S}$	28.629 + $\mathbf{Enc^S}$	62.145 + $\mathbf{Enc^S}$

Table 5: Cryptographic computations execution time.

$\mathbf{Gen^S}$	0.033ms	$\mathbf{Gen^P}$	112.673ms
$\mathbf{Enc^S}$	11.860ms/MB	$\mathbf{Dec^S}$	16.820ms/MB
$\mathbf{Enc^P}$	0.417ms	$\mathbf{Dec^P}$	11.986ms
$\mathbf{Gen^{Sig}}$	1.656ms	$\mathbf{Ver^{Sig}}$	0.269ms

of *revokeU(u, r)* and *revokeP$_{RW}$(r, fn)* depends on the status of the RBAC policy; they are discussed in Section 5.2. In the "Implementation Time" column, we measure the execution time of each operation when running CryptoAC. This includes the time spent for the cryptographic computations and the overhead due to retrieving metadata from the MS database, connections among the modules, parameters validation, and so on. To exclude network latency, we deployed all docker containers on the same device used for testing.

From Table 4, we can see that the maximum overhead brought by CryptoAC implementation is 58.504 ms *addF(fn, f)*) while the minimum is 20.78 ms (*readF(fn)*). While the overhead in percentage points can be high (as the cryptographic time of some operations is really low), the absolute values are reasonably bounded to few dozens of milliseconds.

5.2 Policy-Dependent Operations

We now discuss the performance of the *revokeP$_{RW}$(r, fn)* and *revokeU(u, r)* operations. The former is influenced by a single parameter, i.e., the number of roles having permission over *fn* (except *r*), while the latter is influenced by three parameters, i.e., the number of users assigned to *r*, the number of files *r* has permission over and the number of roles sharing a permission with *r*. We stress that these parameters are related to the users, roles and files concerned by the *revokeP$_{RW}$(r, fn)* and *revokeU(u, r)* operations, and not to the total number of users, roles and files in the AC policy (which may be more).

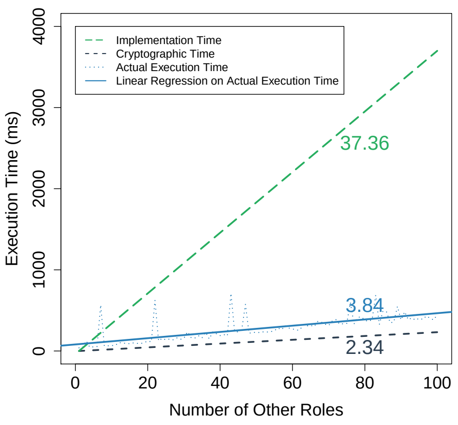


Figure 4: *revokeP* varying $\lvert\{\langle r', \langle fn, *\rangle\rangle \in PA : r \neq r'\}\rvert$.

RevokeP. To determine whether the cryptographic and implementation execution times presented in Table 4 are accurate, we measure the actual execution time of the *revokeP$_{RW}$(r, fn)* operation when varying its influencing parameter from 1 to 100 (i.e., we run the operation 100 times, each time increasing the parameter by 1). We report the 100 measurements in Figure 4 (dotted line) and model the actual execution time through linear regression (solid line). We derive the cryptographic (dashed line) and implementation (long-dashed line) execution times for each point on the X axis by simply multiplying the base values in the "Cryptographic Time" and "Implementation Time" columns of Table 4, respectively. Near each line, we report the relative angular coefficient.

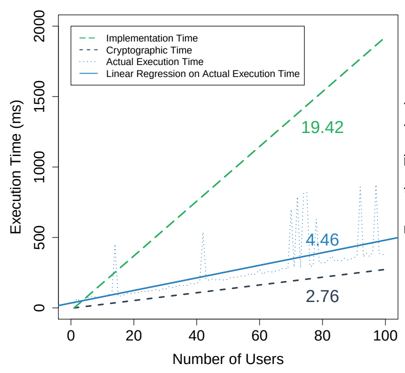


Figure 5: *revokeU* when varying $|\{\langle *, r\rangle \in UR\}|$.

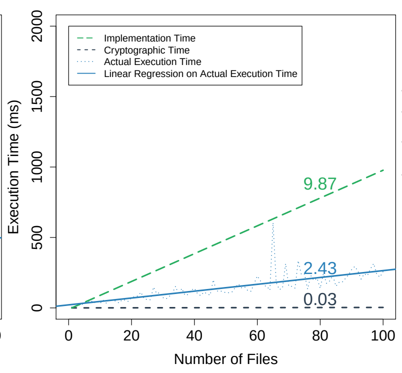


Figure 6: *revokeU* when varying $|\{\langle r, *\rangle \in PA\}|$.

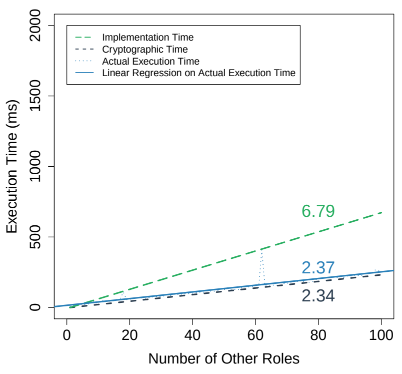


Figure 7: *revokeU* when varying $|\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}|$.

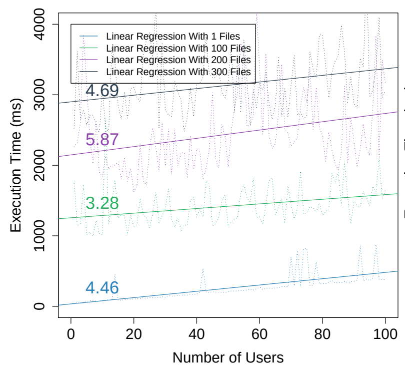


Figure 8: *revokeU* when varying $|\{\langle *, r\rangle \in UR\}|$ and $|\{\langle r, *\rangle \in PA\}|$.

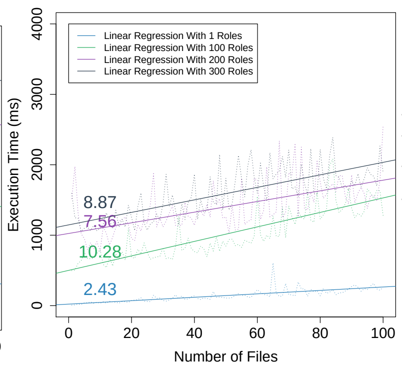


Figure 9: *revokeU* when varying $|\{\langle r, *\rangle \in PA\}|$ and $|\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}|$.

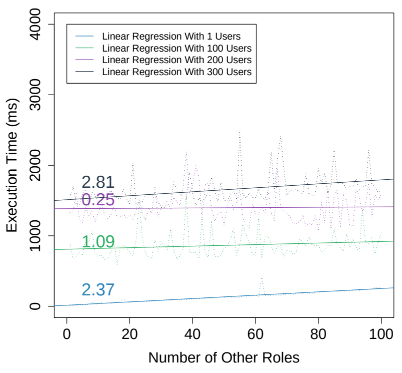


Figure 10: *revokeU* when varying $|\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}|$ and $|\{\langle *, r\rangle \in UR\}|$.

RevokeU. We repeat the same process for the *revokeU(u, r)* operation, with the only difference that the execution time depends on three parameters. As such, we measure the actual execution time when varying these three parameters from 1 to 100 one by one. In other words, we fix two parameters to the value of 1 while ranging on the remaining from 1 to 100. We report the results in Figure 5, 6 and 7.

To check the interdependency of these three parameters, we execute a further experiment. We repeat the analysis while ranging two parameters at the same time and report the results in Figure 8, 9 and 10.

Discussion. Excluding outliers in measurements (due to, e.g., cache misses, operative system tasks), in Figure 4, 5, 6 and 7 we observe that the cryptographic execution time is a lower bound of the actual time, while the implementation execution time is an upper bound. We can justify this behaviour by considering that some costs (e.g., opening a MySQL connection toward the MS database) are incurred only once, independently from the number of users, files or roles involved. Consequently, these costs do not increase proportionally to the number of elements involved, while cryptographic costs instead do. In general, the execution time of *revokeP$_{RW}$(r,fn)* and *revokeU(u, r)* when varying a single parameter remains within reasonable boundaries (i.e., less than half a second) for any practical RBAC policy. This is true also when varying the pairs of parameters ($|\{\langle *, r\rangle \in UR\}|$, $|\{\langle r, *\rangle \in PA\}|$) and ($|\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}|$, $|\{\langle *, r\rangle \in UR\}|$), the overhead being around 3 seconds (Figure 8) and at most 2 seconds (Figure 10). In all cases, the slope of the lines obtained by linear regression are gentle and suggest good scalability.

The behavior of *revokeU(u, r)* when varying the pair ($|\{\langle r, *\rangle \in PA\}|$, $|\{\langle *, \langle fn, *\rangle\rangle \in PA : \exists \langle r, \langle fn, *\rangle\rangle \in PA\}|$) of parameters is less scalable because of the interplay between them. Indeed, the third parameter (i.e., the number of roles sharing a permission with r) includes also the permissions of r itself (i.e., the second parameter). Therefore, increases in the second parameter imply increasing the third one as well. The slope of the lines obtained by linear regression is steeper than before; however, the overhead is at most 2 seconds for the maximum values of the parameters.

Finally, we observe that *revokeP$_{RW}$(r,fn)* and *revokeU(u,r)* are administrative operations. As such, they can be scheduled during low-load states (e.g., overnight) without affecting users' operations. All experimental results are available online.[22]

6 Related Work

To the best of our knowledge, there is no toolchain combining automated trade-off analysis for selecting the most suitable architecture with the deployment of CAC schemes as in COERCIVE. While referring to (Berlato et al., 2020) for the works related to TradeOffBoard, we observe that several researchers proposed high-level designs of CAC schemes, e.g., (Rezaeibagha and Mu, 2016) and (Bacis et al., 2016). Given our focus on the implementation of cryptographic enforcement of AC policies, below we focus on works that are closely related to CryptoAC by considering tools from both the academia (Section 6.1) and commercial contexts (Section 6.2).

6.1 Cryptographic Access Control Schemes Implementations

In (Berlato et al., 2020), we proposed an architectural model for CAC schemes to assist administrators to evaluate different architectures. Our contributions stem from (Berlato et al., 2020), on top of which we propose the toolchain COERCIVE and CryptoAC. We avoid the use of Cloudify,[23] favor a (more agile) microservice approach based on containers (i.e., Docker) and provide the implementation of all entities (i.e., proxy, RM, MS and DS).

In (Sato and Fugkeaw, 2015), the authors implemented a scheme with multiple data owners in `CLOUD-CAT` to investigate scalability and performance in `read` and `write` operations. However, they assumed a fixed architecture and did not discuss the interaction with the CSP. In (Zhou et al., 2013), the authors proposed a scheme with a fixed architecture in which a public cloud stores encrypted data (i.e., DS) while a private cloud stores metadata (i.e., MS). However, we note that it may be unappealing for small organisations to maintain both a public and a private cloud. The authors implemented their scheme just for analysing the performance of `read` and `write` operations. In (Zarandioon et al., 2012), the authors proposed a CAC scheme emphasizing users' privacy by enabling anonymous access to data. They implemented a prototype (with a fixed architecture) interacting with AWS. Noticeably, they provided an interface so that further CSPs can be supported. In (Tang et al., 2012), Tang et al. proposed a scheme with a quorum of key managers deployed as a centralized entity. Users interact with a proxy running either in the client$_u$ domain or on-premise; this is the first scheme allowing to slightly modify the architecture. Moreover, multiple CSPs can theoretically be supported. In (Ghita et al., 2017), the authors implemented a role-based CAC scheme. However, the architecture is fixed and many pragmatic aspects, like run-time modifications to the AC policy, were overlooked. As such, the scheme cannot be used in dynamic scenarios where permissions are assigned and revoked. In (Qi and Zheng, 2019), the authors proposed a CAC scheme similar to (Garrison et al., 2016) while using onion encryption for revocation but obtaining worse performance with respect to (Garrison et al., 2016) (i.e., 7.2%). Each time a permission is revoked, the CSP adds an encryption layer on each affected file. For reading a file, an authorized user has to decrypt all layers. The authors implemented their scheme in a proof-of-concept prototype, `Crypto-DAC`. We note that this approach implicitly assumes that no collusion can happen between revoked users and the CSP. In (Jang-Jaccard, 2018), the authors implemented a CAC scheme based on ABE with AWS. Unfortunately, the authors themselves admitted using a not portable programming library. More importantly, the scheme does not support the dynamicity of the policy. In (Djoko et al., 2019), the authors proposed a CAC scheme leveraging Trusted Execution Environments (TEEs). The authors developed NEXUS to enforce AC policies over shared volumes (i.e., at directory level) by encrypting each file in a volume with a symmetric key. Each key is encrypted with the volume rootkey. Volume sharing happens by sending the rootkey to other users through SGX Remote Attestation. Unfortunately, this scheme requires that each user is equipped with a TEE.

Summarising, the focus of these works is mainly proposing new CAC schemes with novel features. As such, little space is left for additional analysis on alternative architectures responsive to different scenarios. As far as we know, CryptoAC is the only CAC scheme implementation designed for real deployment and not for performance evaluation only.

6.2 Commercial Tools

There are many free and commercial tools to protect sensitive data stored in the cloud. However, the ma-

[22] https://st.fbk.eu/complementary/SECRYPT2021_2 [23] https://cloudify.co/

jority of these tools (e.g., Mega,[24] AxCrypt,[25] Kruptos 2,[26] Cubbit[27]) target individuals or have 1-to-1 sharing of data. Below, we focus on those tools which target organisations and support protection against the honest but curious CSP while enabling controlled information sharing among employees.

Qumulo[28] offers a software-defined file system running both on-premise and on CSPs like AWS and GCP. From version 3.1.5, Qumulo features encryption at rest to preserve the confidentiality of sensitive data in on-premise clusters. Unfortunately, Qumulo still relies on cloud-side encryption for cloud clusters, not preventing CSPs from accessing sensitive data. Zadara[29] is a storage solution compatible with AWS, GCP and Azure. Although Zadara offers encryption at rest, this feature is on a coarse-grained volume-by-volume basis and is disabled by default. A volume is encrypted with a symmetric key which is in turn encrypted with a master key derived from a user's password. Boxcryptor[30] provides end-to-end encryption and data sharing at both user and group level. Each user has a pair of public-private keys, encrypted with a symmetric key derived from the user's password and stored client-side. The modification of the AC policy following the revocation of permissions is not specified in the technical report. LucidLink[31] simulates a Network Attached Storage on users' devices, while data are stored in one of the supported CSPs. Each file is encrypted using AES-256, and the encryption key of the parent folder wraps each file key. The administrator gives a user access to a folder (or file) by encrypting the key of the folder (or file) with the user's public key. Metadata are synchronized through a central service provided by LucidLink. Again, the modification of the AC policy following the revocation of permissions is not specified in the technical reports.

Summarising, there are several tools for preserving the confidentiality of data stored in partially trusted CSP while also enforcing AC policies. However, these tools suffer several drawbacks with respect to COERCIVE, including the lack of support for organisations (i.e., Mega, AxCrypt, Kruptos 2, Cubbit), incomplete protection of data (i.e., Qumulo), coarse-grained AC (i.e., Zadara), or constraints on the architecture to adopt (i.e., Boxcryptor and LucidLink require a central service to function). While not having the same level of maturity of these tools—for instance, some tools support two-factor authentication and chats and video-calls—CryptoAC instead addresses exactly these issues, as shown in Table 6.

[24] https://mega.nz/
[25] https://www.axcrypt.net/
[26] https://www.kruptos2.co.uk/
[27] https://www.cubbit.io/technology
[28] https://qumulo.com/
[29] https://www.zadara.com/
[30] https://www.boxcryptor.com
[31] https://www.lucidlink.com/

Table 6: Comparison with commercial tools.

	Mega	AxCrypt	Kruptos2	Cubbit	Qumulo	Zadara	Boxcryptor	LucidLink	**CryptoAC**
Open Source	●	●							●
Multi-Platform	●	●	●	●	●	●	●	●	●
Mobile Support	●	●	●				●	●	●[29]
Fine-grained AC							●		●
All Features Free							●[28]		●
Policy Dynamicity	●	●		●	●		?	?	●
Multi-Cloud Support		●	●		●		●	●	●
Flexible Architecture					●	●			●
Optimised Architecture									●
Confidentiality w.r.t. CSP	●	●	●	●		●	●	●	●
Two-Factor Authentication	●					●	●		
Chat and Video Calls Support	●					●			

[28]Free for non-commercial use only; [29]Through browser only;

7 Conclusion

In this paper, we have proposed COERCIVE, a toolchain for optimal enforcement of CAC policies in the cloud. COERCIVE is composed of two tools: TradeOffBoard and CryptoAC. The former is already described in (Berlato et al., 2020) and it allows for identifying the most suitable architecture among dozens of possible alternatives for a given use case scenario by automatically solving an optimization problem. The latter is instead an original contribution and allows for the portable and efficient deployment of a cryptographic enforcement of role-based AC policies in several CSPs by using (Docker) containers technology. Furthermore, we conducted a thorough performance evaluation of CryptoAC to demonstrate its scalability and efficiency. COERCIVE is available online as open-source software.[32]

Future Directions. We are implementing further functionalities of CryptoAC, like multi-administrator support and tree-based file sharing. Moreover, we are working on the secure management of cryptographic material (e.g., users' private keys) to seamlessly support solutions like Trusted Execution Environments. To extend the applicability of COERCIVE, we are planning to integrate CryptoAC with Hyperledger Fabric to replace the cloud with a decentralized solution. Finally, in this paper we analyzed the performance of CryptoAC with respect to scalability and asymptotic behaviours, which may be unlikely in real-world scenarios. To further study the performance of CryptoAC, we are developing a dedicated simulator to evaluate CryptoAC on arbitrary AC policies through real-world workflows (i.e., sequences of operations).

[32] https://github.com/stfbk/CryptoAC

Acknowledgments

The authors were supported in part by the Integrated Framework for Predictive and Collaborative Security of Financial Infrastructures (FINSEC) project and the 5G-CARMEN project that received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreements no. 786727 and no. 825012, respectively.

REFERENCES

Bacis, E., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Rosa, M., and Samarati, P. (2016). Mix&slice: Efficient access revocation in the cloud. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 217–228. ACM.

Berlato, S., Carbone, R., Lee, A. J., and Ranise, S. (2020). Exploring architectures for cryptographic access control enforcement in the cloud for fun and optimization. In *15th ACM ASIA Conference on Computer and Communications Security (ASIACCS 2020)*. ACM.

Cai, F., Zhu, N., He, J., Mu, P., Li, W., and Yu, Y. (2019). Survey of access control models and technologies for cloud computing. *Cluster Computing*, 22:6111–6122.

Djoko, J. B., Lange, J., and Lee, A. J. (2019). NeXUS: Practical and secure access control on untrusted storage platforms using client-side SGX. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413. IEEE.

Domingo-Ferrer, J., Farràs, O., Ribes-González, J., and Sánchez, D. (2019). Privacy-preserving cloud computing on sensitive data: A survey of methods, products and challenges. *Computer Communications*, 140-141:38–60.

Garrison, W. C., Shull, A., Myers, S., and Lee, A. J. (2016). On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 819–838. IEEE.

Ghita, V., Costea, S., and Tapus, N. (2017). Implementation of cryptographically enforced rbac. *The Scientific Bulletin - University Politehnica of Bucharest*, 79(2):9–3–102.

Horandner, F., Krenn, S., Migliavacca, A., Thiemer, F., and Zwattendorfer, B. (2016). CREDENTIAL: A Framework for Privacy-Preserving Cloud-Based Data Sharing. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 742–749, Salzburg, Austria. IEEE.

Jang-Jaccard, J. (2018). A Practical Client Application Based on Attribute Based Access Control for Untrusted Cloud Storage. In *Computer Science & Information Technology*, pages 01–15. Academy & Industry Research Collaboration Center (AIRCC).

Premarathne, U., Abuadbba, A., Alabdulatif, A., Khalil, I., Tari, Z., Zomaya, A., and Buyya, R. (2016). Hybrid Cryptographic Access Control for Cloud-Based EHR Systems. *IEEE Cloud Computing*, 3(4):58–64.

Qi, S. and Zheng, Y. (2019). Crypt-DAC: Cryptographically Enforced Dynamic Access Control in the Cloud. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1.

Rezaeibagha, F. and Mu, Y. (2016). Distributed clinical data sharing via dynamic access-control policy transformation. *International Journal of Medical Informatics*, 89:25–31.

Samarati, P. and de Capitani di Vimercati, S. (2000). Access control: Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171, pages 137–196. Springer Berlin Heidelberg.

Sato, H. and Fugkeaw, S. (2015). Design and Implementation of Collaborative Ciphertext-Policy Attribute-Role based Encryption for Data Access Control in Cloud. *Journal of Information Security Research*, 6(3):71–84.

Tang, Y., Lee, P. P., Lui, J. C., and Perlman, R. (2012). Fade: Secure overlay cloud storage with file assured deletion. *IEEE Transactions on Dependable and Secure Computing*, 9(6):903–916.

Zarandioon, S., Yao, D., and Ganapathy, V. (2012). K2c: Cryptographic Cloud Storage with Lazy Revocation and Anonymous Access. In Rajarajan, M., Piper, F., Wang, H., and Kesidis, G., editors, *Security and Privacy in Communication Networks*, volume 96, pages 59–76. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zhou, L., Varadharajan, V., and Hitchens, M. (2013). Achieving Secure Role-Based Access Control on Encrypted Data in Cloud Storage. *IEEE Transactions on Information Forensics and Security*, 8(12):1947–1960.