Work-in-Progress: Optimizing Performance of User Revocation in Cryptographic Access Control with Trusted Execution Environments

Ion Andy Ditu*, Stefano Berlato†, Matteo Busi‡, Roberto Carbone† and Silvio Ranise*†
* *University of Trento, Trento, Italy*
Email: ionandy.ditu@studenti.unitn.it
† *Fondazione Bruno Kessler, Trento, Italy*
Email: {sberlato,carbone,ranise}@fbk.eu
‡ *University of Venice, Venice, Italy*
Email: matteo.busi@unive.it

***Abstract*—The possibility (and convenience) of storing and sharing data through the cloud entails a set of concerns to data security, such as the presence of external attackers, malicious insiders, and *honest-but-curious* cloud providers. Cryptographic Access Control (CAC) addresses these concerns but presents practical limitations, primarily due to the computational overhead of key management. In particular, *user revocation* (that is, revoking a user's access to encrypted data) often requires rotating those Data Encryption Keys (DEKs) to which the revoked user lost access — lest the revoked user might have cached them for future use. Moreover, new DEKs must be distributed to remaining authorized users and data re-encrypted. In this work-in-progress paper, we explore how Trusted Execution Environments (TEEs) may conceal cryptographic keys from users in CAC and improve efficiency in key management during user revocation.**

***Index Terms*—Cryptography, Access Control, Trusted Execution Environment, Key Management**

1. Introduction

Cryptographic Access Control (CAC) is the practice of enforcing Access Control (AC) policies through cryptography: resources[1] are encrypted with Data Encryption Keys (DEKs) which are selectively shared to authorized users via Key Encryption Keys (KEKs). CAC has been extensively explored in the literature, especially for protecting cloud-hosted resources from external attackers, malicious insiders, and honest-but-curious cloud providers.[2]

However, CAC incurs significant overhead, particularly so when AC policies are modified and key management must reflect such modifications. User revocation — that is, revoking permissions from a user — typically involves several cryptographic computations, such as rotating affected DEKs, distributing the new DEKs through KEKs to all other authorized users, and re-encrypting affected resources. These cryptographic computations are necessary to prevent revoked users from colluding with cloud providers and using cached DEK to gain unauthorized access to resources, but are also computationally expensive: a single user revocation may result in hundreds or even thousands of such cryptographic computations [13]. In other words, while allowing for enforcing fine-grained AC policies and providing end-to-end confidentiality and integrity to outsourced resources, the adoption of CAC is often challenging due to its high computational costs.

In this work-in-progress paper, we investigate the use of Trusted Execution Environments (TEEs)[3] to mitigate the computational overhead of user revocation in CAC. More specifically, we propose a methodology leveraging TEEs on users' devices to conceal cryptographic keys while still allowing users to use them. In this way, we avoid the need for rotating and distributing DEKs and KEKs and re-encrypting resources after user revocation. We design our methodology to be adaptable to many CAC schemes[4] and TEE implementations, aiming for broad applicability, compatibility, and interoperability. Moreover, our methodology can be seamlessly applied within a CAC scheme in a blended fashion, whereby only some users have a TEE. Finally, we present a (theoretical) application of our methodology and briefly discuss security concerns.

2. Related Work

Many researchers have proposed interesting solutions for ensuring integrity and confidentiality of cloud-hosted resources. Below, we focus on those solutions which explicitly employ TEEs and aim at enforcing AC policies (e.g., solutions based only on software or designed for other purposes like confidential computing are out of scope). We cluster such solutions based on the chosen deployment strategy — that is, architecture — for TEEs.

Cloud-Side TEE. SeGShare [12], MOSE [16], Pesos [18] and Precursor [20] rely on cloud-side TEEs: users request access to resources via an application running within the TEE which acts as both a policy decision point and a policy enforcement point. Hence, these solutions differ from our methodology as they do not use CAC, i.e., do not enforce AC policies with cryptography.

1. We generically define a resource as a logical vessel for (often sensitive or personal) data, for instance, files and documents.

2. An honest-but-curious cloud provider is assumed to provide genuine services while, at the same time, trying to extract any profitable information from cloud-hosted resources [23].

3. As defined by GlobalPlatform, a TEEs is a hardware-based execution environment that runs alongside but isolated from a device's main operating system, providing security features such as isolated execution, integrity of trusted applications, and integrity and confidentiality of data (https://globalplatform.org/specs-library/tee-system-architecture/).

4. The set of instructions describing how to distribute (or revoke access to) cryptographic keys is usually called CAC scheme.

On-Premise TEE. IBBE-SGX [6], A-SKY [7], and MQT-TZ [28] rely on TEEs within devices hosted on the premises of organizations (e.g., data centers) to mediate accesses to resources. Among these solutions, only IBBE-SGX explicitly considers an integration between CAC and TEEs. However, cryptographic computations are always offloaded to end users — which, consequently, have access and can cache cryptographic keys; in other words, these solutions do not focus on concealing keys from users or mitigating the overhead of user revocation.

Client-Side TEE. NeXUS [9] and LAUXUS [8] rely on TEEs within users' devices. However, these solutions do not use CAC. Instead, the application running within the TEE acts as both a policy decision point and a policy enforcement point. NeXUS and LAUXUS specifically rely on Intel SGX's sealing keys acquired through a key exchange protocol with resources' owners (who must always be available online). Sealing keys are used to decrypt resources — that embed the AC policies used by TEEs for mediating access requests.

Hybrid TEE. MooseGuard [1] and EnShare [15] rely on TEEs on both cloud- and client-side to distribute cryptographic computations: users' TEEs en/decrypt resources, whereas the cloud's TEE handles authentication and enforces AC policies — but not through CAC.

With respect to our methodology, the aforementioned solutions either have a different focus (e.g., data deduplication [12], anonymity [7], auditability [8], pattern obliviousness [16]), or consider a specific TEE implementation (e.g., ARM TrustZone [28], Intel SGX [6], [9]), or rely on TEEs as policy decision and enforcement point. Except for IBBE-SGX, no solution considers using TEEs in CAC and, to the best of our knowledge, none focus on relieving the computational overhead of user revocation in CAC by exploring the use of TEEs to conceal cryptographic keys.

3. Background

AC consists in mediating every request to resources maintained by a system and determining whether the request should be granted or denied [25]. AC expects an AC policy declaring what agents (e.g., users) can perform what actions (e.g., read) on what resources. The AC policy is typically defined by an administrator, which usually corresponds to the owner of the resources or the system. The AC policy is formally represented by an AC model giving the semantics for granting or denying requests — e.g., Role-based Access Control (RBAC) [26], Attribute-based Access Control (ABAC) [17]. The software enforcing an AC policy based on the chosen AC model is called AC enforcement mechanism (or simply "mechanism"). RBAC [26] is an AC model widely adopted in both academia and industry due to its simple implementation and intuitive role assignment process [5]. In RBAC, users are assigned to roles, and roles are assigned to permissions (a permission represents the privilege of performing an action over a resource — for instance, read a document). Consequently, users activate roles to access permissions. In the core RBAC model considered by the National Institute of Standards and Technology (NIST) [11], the state of a RBAC policy can be represented as a tuple $\langle \mathbf{U}, \mathbf{R}, \mathbf{F}, \mathbf{UR}, \mathbf{PA} \rangle$, where $\mathbf{U}$ is the set of users, $\mathbf{R}$ is the set of roles, $\mathbf{F}$ is the set of resources, $\mathbf{UR} \subseteq \mathbf{U} \times \mathbf{R}$ is the set of user-role assignments and $\mathbf{PA} \subseteq \mathbf{R} \times \mathbf{PR}$ is the set of role-permission assignments, being $\mathbf{PR} \subseteq \mathbf{F} \times \mathbf{OP}$ a derivative set of $\mathbf{F}$ combined with a fixed set of actions $\mathbf{OP}$ (e.g., Read for read, Write for write). Both $\mathbf{OP}$ and $\mathbf{PR}$ are not part of the state, as $\mathbf{OP}$ remains constant over time and $\mathbf{PR}$ is derivative of $\mathbf{F}$ and $\mathbf{OP}$. We note that role hierarchies can always be compiled away by adding suitable pairs to $\mathbf{UR}$. A user u can use a permission $\langle f, op \rangle$ (i.e., perform the action op over the resource f) via a role r if $\exists r \in \mathbf{R} \mid (u, r) \in \mathbf{UR} \wedge (r, \langle f, op \rangle) \in \mathbf{PA}$.

Cryptographic Access Control. CAC generally expects each resource to be associated with a symmetric key (i.e., a DEK) and each user to be associated with a pair of public-private keys (i.e., a KEKs). Different CAC schemes may encode AC policies through key management differently; common approaches involve Public Key Infrastructure (PKI), Proxy Re-Encryption (PRE), and Attribute-Based Encryption (ABE). Despite their differences, CAC schemes proposed in the literature typically expect at least 3 primary operations — onboarding of new users, access to resources (e.g., read, write), and user revocation — and are built upon a set of 4 common entities [3]: the *proxy* interfaces users with data by performing cryptographic computations, the *Reference Monitor (RM)* mediate users' requests to add or write over resources, the *Metadata Manager (MM)* manages AC policy-related metadata such as encrypted DEKs, and the *Data Manager (DM)* manages resources containing encrypted data. To provide a concrete example, we now describe the CAC scheme for RBAC proposed in [2]. In that CAC scheme, each resource f is associated with a symmetric key $\mathbf{k}_f^{\mathbf{sym}}$ (i.e., a DEK), each role r is associated with a pair of public-private keys for encryption and decryption $(\mathbf{k}_r^{\mathbf{enc}}, \mathbf{k}_r^{\mathbf{dec}})$ (i.e., a KEKs), and each user u is associated with $(\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}})$. The DEK of a resource is encrypted with the KEKs of all roles authorized to access that resource, and the KEKs of a role are encrypted with the KEKs of all users assigned to that role. In other words, a user u assigned to a role r authorized to access a resource f is encoded as $\{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}}$ and $\{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}}$. Consequently, u can decrypt $\{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}}$ with $\mathbf{k}_u^{\mathbf{dec}}$ — obtaining $\mathbf{k}_r^{\mathbf{dec}}$ — and then decrypt $\{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}}$ with $\mathbf{k}_r^{\mathbf{dec}}$ — obtaining $\mathbf{k}_f^{\mathbf{sym}}$. The encrypted DEKs and KEKs are attached to decorated sets representing the state of the RBAC CAC policy: $\langle \mathbf{U_c}, \mathbf{R_c}, \mathbf{F_c}, \mathbf{UR_c}, \mathbf{PA_c} \rangle$ — where the subscript $_\mathbf{c}$ stands for "cryptographic". Below, we report a simplified version of these decorated sets:[5]

$$\begin{gathered} \langle u, \mathbf{k}_u^{\mathbf{enc}} \rangle \in \mathbf{U_c} \\ \langle r, \mathbf{k}_r^{\mathbf{enc}} \rangle \in \mathbf{R_c} \\ \langle f \rangle \in \mathbf{F_c} \\ \langle u, r, \{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}} \rangle \in \mathbf{UR_c} \\ \langle r, f, \{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}}, op \rangle \in \mathbf{PA_c} \end{gathered}$$

5. "Simplified" because we omit 3 components secondary in our discussion: *pseudonyms* (strings used to hide the actual identifiers of users, roles, and resources), *version numbers* (integers used for differentiating between old and new DEKs and KEKs), and *digital signatures* (applied on the decorated sets against tampering or accidental modifications).

We report in Algorithms 1, 2, and 3 the pseudocode of the CAC scheme in [2] for adding users, revoking users from roles, and revoking permissions from roles, respectively. We note that revoking a permission $\langle f, op \rangle$ from a role r (Algorithm 3) entails rotating the DEK $\mathbf{k}_f^{\mathbf{sym}}$ and distributing the new DEK $\mathbf{k}_f^{\mathbf{sym}\prime}$ to all other authorized roles. Similarly, revoking a user u from a role r (Algorithm 2) entails rotating the KEKs $(\mathbf{k}_r^{\mathbf{enc}}, \mathbf{k}_r^{\mathbf{dec}})$, distributing the new KEKs $(\mathbf{k}_r^{\mathbf{enc}\prime}, \mathbf{k}_r^{\mathbf{dec}\prime})$ to all other authorized users, rotate the DEKs of all resources to which u had access through r, and distributing the new DEKs to all authorized roles. In other words, user revocation is complex and computationally demanding to the point of incurring likely prohibitive overheads [13].

Trusted Execution Environments. TEEs are often used for digital wallets, secure booting, and confidential computing as well as key management. Although TEEs may be implemented with different technologies — and their level of security may vary accordingly — basic features inherent to any (or at least the vast majority of) TEEs are remote attestation, isolated execution, and secure channels [24]. *Remote attestation* allows for verifying the integrity (hence, the trustworthiness) of the TEE and its contents; one possible implementation involves generating a report containing measurements of the TEE's status (e.g., deployed software, firmware) using cryptographic hash functions: the report is signed with a dedicated private key and sent to the *attestation authority* — a trusted third party that usually coincides with the manufacturer of the TEE — which verifies the signature of the report and compares the measurements contained therein against expected values. Then, *isolated execution* typically relies on secure enclaves to protect software running within the TEE either at the Virtual Machine (VM) level or at the application level. Protection at the VM level (e.g., AMD SEV[6]) often packages applications as VMs and protects them from the underlying hypervisor while still granting users a certain degree of transparency and visibility on the (data and code running within) the VM, while protection at the application level (e.g., Intel SGX[7]) prevents even users from accessing data and code running within the TEE — that is, users interact with the TEE as a black box. In both cases, the (portion of the) application running inside the TEE is usually referred to as "trusted application" Finally, TEEs can establish local (e.g., I/O) and remote *secure channels* (e.g., with TLS) for data transmission protected against external attacks (e.g., key-logging, replay, eavesdropping, tampering). Intuitively, some TEEs may offer further features (e.g., sealing, resistance to side-channel attacks, obliviousness) which, albeit useful, are not required in our methodology.

Algorithm 1: add u in [2]

```
1 $u$ generates $(\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}})$
2 $u$ sends $\mathbf{k}_u^{\mathbf{enc}}$ to the administrator
3 if (verify $\mathbf{k}_u^{\mathbf{enc}}$'s authenticity) then
4     add $\langle u, \mathbf{k}_u^{\mathbf{enc}} \rangle$ to $\mathbf{U_c}$
5 else
6     return $\perp$
```

Algorithm 2: revoke u from r in [2]

```
1  delete $\langle u, r, * \rangle$ from $\mathbf{UR_c}$
2  generate new $(\mathbf{k}_r^{\mathbf{enc}\prime}, \mathbf{k}_r^{\mathbf{dec}\prime})$
3  replace $\langle r, \mathbf{k}_r^{\mathbf{enc}} \rangle$ with $\langle r, \mathbf{k}_r^{\mathbf{enc}\prime} \rangle$ in $\mathbf{R_c}$
4  forall $\langle u', r, * \rangle \in \mathbf{UR_c} : u \neq u'$ do
5      replace $\langle u', r, * \rangle$ with $\langle u', r, \{\mathbf{k}_r^{\mathbf{dec}\prime}\}_{\mathbf{k}_{u'}^{\mathbf{enc}}} \rangle$ in $\mathbf{UR_c}$
6  forall $\langle r, f, *, * \rangle \in \mathbf{PA_c}$ do
7      generate new $\mathbf{k}_f^{\mathbf{sym}\prime}$
8      re-encrypt data in $f$ with $\mathbf{k}_f^{\mathbf{sym}\prime}$
9      replace $\langle r, f, *, * \rangle$ with $\langle r, f, \{\mathbf{k}_f^{\mathbf{sym}\prime}\}_{\mathbf{k}_r^{\mathbf{enc}\prime}}, * \rangle$ in $\mathbf{PA_c}$
10     forall $\langle r', f, *, * \rangle \in \mathbf{PA_c} : r \neq r'$ do
11         replace $\langle r', f, *, * \rangle$ with $\langle r', f, \{\mathbf{k}_f^{\mathbf{sym}\prime}\}_{\mathbf{k}_{r'}^{\mathbf{enc}}}, * \rangle$ in $\mathbf{PA_c}$
```

Algorithm 3: revoke $\langle f, op \rangle$ from r in [2]

```
1 delete $\langle r, f, *, * \rangle$ from $\mathbf{PA_c}$
2 generate new $\mathbf{k}_f^{\mathbf{sym}\prime}$
3 re-encrypt data in $f$ with $\mathbf{k}_f^{\mathbf{sym}\prime}$
4 forall $\langle r', f, *, * \rangle \in \mathbf{PA_c} : r \neq r'$ do
5     replace $\langle r', f, *, * \rangle$ with $\langle r', f, \{\mathbf{k}_f^{\mathbf{sym}\prime}\}_{\mathbf{k}_{r'}^{\mathbf{enc}}}, * \rangle$ in $\mathbf{PA_c}$
```

4. Methodology

Below, we first consider alternative deployment strategies for TEE (Section 4.1), then explain the design of our methodology (Section 4.2), and finally present a (theoretical) application to a specific CAC scheme (Section 4.3).

4.1. Architecture

The integration of TEEs with CAC requires identifying the optimal deployment strategy — that is, the architecture — for TEEs among those identified in Section 2: cloud-side, on-premise, client-side, and hybrid. Regardless of the chosen architecture, KEKs and DEKs used within TEEs are always concealed from both users and the cloud provider. Hence, we consider further criteria for choosing among the available TEE architectures: ease of deployment, End-to-End Encryption (E2EE), and scalability. Concerning *ease of deployment*, client-side architectures are more challenging than others since not all users may have a TEE-equipped device. Moreover, TEEs may come with heterogeneous implementations. Conversely, on-premise and cloud-side architectures are likely to provide the same TEE implementation. Then, concerning *E2EE*, client-side and on-premise architectures do provide true E2EE, meaning that resources remain encrypted with the same DEK from the sender to the recipient (or, at least, to the organization). On the other hand, in cloud-side architectures, encrypted data in resources would be decrypted within the cloud-side TEE and users would then need to establish a secure channel toward the TEE to access plaintext data. Finally, concerning *Scalability*, TEEs in cloud-side and on-premise architectures may quickly become a single point of failure and a bottleneck for performance when handling a large number of users'

6. AMD Secure Encrypted Virtualization — SEV (https://www.amd.com/en/developer/sev.html)

7. Intel® Software Guard Extensions (https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html)

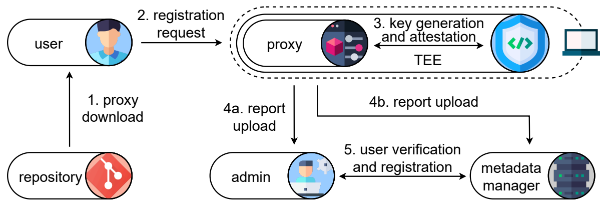

Figure 1: Onboarding

requests. Client-side architectures, instead, distribute cryptographic computations and inherently yield scalability — as argued in [20] and experimentally shown in [15]. Intuitively, a hybrid architecture may combine the aforementioned advantages and disadvantages depending on where TEEs are employed. As a final remark, we note that CAC is not strictly needed in on-premise, hybrid, and cloud-side architectures. In fact, having a TEE mediating users' requests in a centralized manner amounts at having a centralized trusted entity which can enforce AC policies in a traditional manner. Consequently, as argued in [3], in these 3 architectures simple symmetric encryption would suffice: the TEE would protect each resource with a dedicated DEK, and this DEK would not be distributed to users as the AC policy would be enforced by the TEE itself. For all of these considerations, we choose to investigate how TEEs may improve key management efficiency during user revocation in CAC within a client-side architecture.

4.2. Design

Aiming for broad applicability, compatibility, and interoperability with different CAC schemes and TEE implementations, we build our methodology upon primary operations of CAC (i.e., onboarding, access to resources, and user revocation), common entities of CAC (i.e., proxy, RM, MM, DM) and basic features of TEEs (i.e., remote attestation, application-level isolated execution, secure channels). In this way, we do not tie our methodology to a specific CAC scheme or TEE implementation (which may be deprecated, as it happened with client-side Intel SGX[8]). Moreover, we can ideally employ different TEE implementations within the same deployment of our methodology, covering a wide spectrum of scenarios (e.g., involving desktops, mobile, and IoT devices). Our methodology comprises 3 steps corresponding to the primary operations of CAC: onboarding, access to resources, and user revocation. Below, we describe these 3 steps in a generic scenario where an organization employs CAC for controlled sharing of cloud-hosted resources among its employees. For the sake of simplicity, here we assume all users to possess TEE-equipped devices — we discuss deployments in a blended fashion in Section 4.3.

Onboarding. A new user downloads the proxy, packaged as a trusted application, in a TEE-equipped device at registration time — that is, when the user is added to

8. Intel SGX deprecated in 11th Gen processors. (https://community.intel.com/t5/Intel-Software-Guard-Extensions/Intel-SGX-deprecated-in-11th-Gen-processors/m-p/1351848)

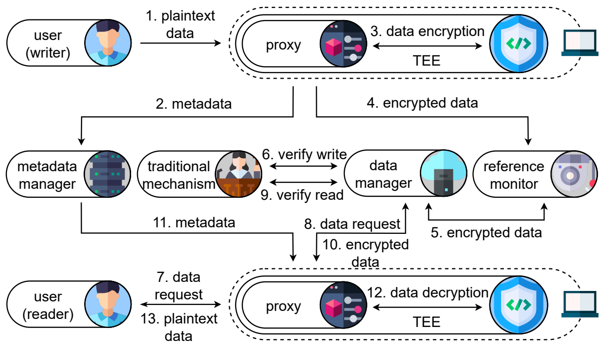

Figure 2: Access to Resources

the AC policy (see step 1 in Figure 1). The user requests to register (step 2) and, after a successful authentication (user authentication is out of scope), the proxy generates the users' KEKs within the TEE through application-level isolated execution. Note that any communication involving the proxy occurs over secure channels. Through the remote attestation feature, the TEE creates a report for verifying the integrity of both the TEE and the proxy. Furthermore, the report binds the newly generated KEKs — more precisely, the public encryption key — to the user, preventing deceptive practices where a user generates a valid report without associated KEKs (step 3). Moreover, being generated within the TEE, users do not have access to their KEKs. The report is either sent to the administrator (step 4a) or uploaded in the MM (step 4b) for user verification and registration (step 5); the interaction with the attestation authority is omitted for simplicity. Summarizing, the onboarding step in our methodology allows for establishing a robust user-KEKs association (see Section 5 for a discussion on security).

Access to Resources. A user who wants to add a new resource or modify an existing resource sends plaintext data to the proxy (see step 1 in Figure 2). The proxy — using the corresponding DEK fetched from the MM[9] and decrypted using the user's KEKs stored within the TEE (step 2) as described in Section 3 — encrypts the plaintext data (step 3) and sends the encrypted data to the RM (step 4). The RM validates the user's request, in the sense that the request is well-formatted according to the underlying CAC scheme (e.g., digital signatures are valid, version numbers are correct), and sends the encrypted data to the DM (step 5). Before adding the encrypted data to the resource and possibly overwriting old encrypted data, the DM may verify the compliance of the user's request against the AC policy via a traditional mechanism (step 6). In fact, cryptography alone cannot prevent malicious users from tampering with resources, e.g., by overwriting their content with spurious data. Hence, CAC schemes may require a traditional mechanism to verify the privileges of users who request to add or modify existing resources — more often, this verification is

9. This assumes that the administrator previously generated and distributed the DEK according to the AC policy. For new resources, some CAC schemes expect users themselves to generate a new DEK and distributed it to the administrator instead.

carried out by the RM [2].[10] A user who wants to read a resource sends a request to the proxy (step 7) which asks the DM to download the encrypted data contained in the resource (step 8). The DM — optionally after a successful verification with a traditional mechanism (step 9) — allows the proxy to download the encrypted data (step 10). Finally, the proxy — using the corresponding DEK fetched from the MM and decrypted using the user's KEK stored within the TEE (step 11) — decrypts the encrypted data (step 12) and sends the plaintext data to the user (step 13). The proxies delete DEKs after use.

User Revocation. Whenever the administrator needs to perform user revocation — and the revocation entails that the user will lose access to one or more DEKs or KEKs — the administrator simply updates the AC policy in the MM. In fact, DEKs and KEKs were never accessed by the user nor cached by the proxy, thus it is not necessary to rotate/distribute DEKs and KEKs or re-encrypt resources.

4.3. Application

To offer a first insight on how our methodology may relieve computational overhead of user revocation in CAC, below we discuss a (theoretical) application to the CAC scheme for RBAC proposed in [2]. Concretely, we modify the pseudocode of the operations described in Algorithms 1, 2, and 3 for adding users, revoking users from roles, and revoking permissions from roles, respectively; we report the correspondingly modified pseudocode in Algorithms 4, 5, and 6. In particular, adding a user with our methodology now requires the user's KEKs ($\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}}$) to be generated within the TEE. Moreover, a report has to be generated and subsequently verified (see Algorithm 4). Hence, the addition of new users — which is performed only once per user during onboarding — arguably presents higher computational costs. However, user revocation now requires a simple modification to the state of the RBAC CAC policy (see Algorithms 5 and 6). In fact, there is no need to rotate DEKs and DEKs, distribute them, or re-encrypt resources. As a consequence, our methodology

10. Some CAC schemes may omit this verification, either because resources cannot be overwritten but only appended (e.g., as in [4]) or because resources are versioned and new versions of resources are validated by the next user accessing the resources (e.g., as in [10]).

Algorithm 4: add u with TEE

```
1 $u$ generates ($\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}}$) and the attestation $\mathbf{Att}(\mathbf{k}_u^{\mathbf{enc}})$
    within the TEE
2 $u$ sends $\mathbf{k}_u^{\mathbf{enc}}$ and $\mathbf{Att}(\mathbf{k}_u^{\mathbf{enc}})$ to the administrator
3 if (verify $\mathbf{k}_u^{\mathbf{enc}}$'s and $\mathbf{Att}(\mathbf{k}_u^{\mathbf{enc}})$'s authenticity) then
4   add $\langle u, \mathbf{k}_u^{\mathbf{enc}} \rangle$ to $\mathbf{U_c}$
5 else
6   return $\perp$
```

Algorithm 5: revoke u from r with TEE

```
1 delete $\langle u, r, * \rangle$ from $\mathbf{UR_c}$
```

Algorithm 6: revoke $\langle f, op \rangle$ from r with TEE

```
1 delete $\langle r, f, *, * \rangle$ from $\mathbf{PA_c}$
```

avoids all cryptographic computations in user revocation, significantly reducing the overall overhead of CAC while preserving the same level of security (see Section 5 for a discussion on security). The pseudocode of all other operations of the CAC scheme in [2] (e.g., access to resources) is not modified — the only difference is that the proxy runs within a TEE and not on generic hardware. Finally, we highlight that our methodology can be deployed in a blended fashion, whereby only some users employ a TEEs: simply, the operations in Algorithms 1, 2, and 3 apply to any user not employing a TEE, while the operations in Algorithms 4, 5, and 6 apply to any user employing a TEE. The administrator may keep track of users employing a TEE, e.g., by adding a flag to $\mathbf{U_c}$.

5. Security

Our methodology aims at integrating TEEs in CAC to improve efficiency in key management during user revocation by allowing administrators to "skip" some cryptographic computations — namely, resources re-encryption, DEKs and DEKs rotation and distribution. The underlying intuition is that revoked users cannot access nor cache cryptographic keys if these are concealed by TEEs. However, our methodology makes an important assumption: the basic features of remote attestation, application-level isolated execution, and secure channels provided by TEEs are indeed secure and implemented correctly. In fact, TEE implementations have been successfully attacked in the past [21], [22], [27], [29]. In our context, such attacks may ultimately expose cryptographic keys, allowing revoked users to gain unauthorized access to resources by colluding with cloud providers. The proxy itself can be seen as a single point of failure, and its implementation may present vulnerabilities. Another point of concern is the need to trust the TEE manufacturer and the attestation authority. Malicious manufacturers may introduce vulnerabilities in the design of TEEs, while malicious or compromised attestation authorities may validate forged reports. In the latter case, the binding between the identity of a user and its KEK would be violated. Regardless, we note that an administrator can fix an eventual exposure of cryptographic keys by simply performing the "skipped" cryptographic computations — which, as a precaution, can be performed anyway when the system is idling (e.g., during nights). Also, in Section 4.2, we say that a proxy running within a TEE deletes DEKs after having used them; as a corollary, the proxy fetches cryptographic material from the MM at every access to a resource. To further improve performance, one may allow the proxy to cache DEKs for a certain time interval — at the cost of allowing revoked users to still access resources during that time interval. As an intermediate solution, the proxy may cache DEKs but still query the traditional mechanism to verify the user's privileges before granting access to a resource. In general, there is a trade-off between security and performance, which is typical of distributed systems and is related to the CAP theorem [14]. Instead, replay attacks (e.g., a revoked user providing an old encrypted DEK to the proxy) are not a concern given the use of secure channels for remote communication.

6. Conclusion

This work-in-progress paper presented a methodology leveraging TEEs to mitigate the computational overhead of user revocation in CAC. By relying on primary operations and common entities of CAC and basic features of TEEs, our methodology aims to be suitable for different CAC schemes and TEE implementations. Moreover, we presented a (theoretical) application of our methodology to the CAC scheme for RBAC proposed in [2] to give a first insight of the potential performance benefits and briefly discussed security concerns.

Future Work. We plan to provide a proof-of-concept implementation of our methodology when applied to the CAC scheme in [2] and experimentally measure performance improvements. Then, we plan to provide a formal security analysis of our methodology (e.g., using the approach in [19]) and further discuss its robustness to known TEE vulnerabilities and attacks.

Acknowledgements

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU and by Ministero delle Imprese e del Made in Italy (IPCEI Cloud DM 27 giugno 2022 – IPCEI-CL-0000007) and European Union (Next Generation EU).

References

[1] J. Baker. MooseGuard: secure file sharing at scale in untrusted environments. Master's thesis, University of Pittsburgh, 2020.

[2] S. Berlato. *A Security Service for Performance-Aware End-to-End Protection of Sensitive Data in Cloud Native Applications*. Phd thesis, University of Genoa, May 2024. Available at https://hdl.handle.net/11567/1174596.

[3] S. Berlato, R. Carbone, A. J. Lee, and S. Ranise. Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. *ACM Transactions on Privacy and Security*, 25(1):1–37, 2022.

[4] S. Berlato, U. Morelli, R. Carbone, and S. Ranise. End-to-end protection of IoT communications through cryptographic enforcement of access control policies. In Shamik Sural and Haibing Lu, editors, *Data and Applications Security and Privacy XXXVI*, volume 13383, pages 236–255. Springer International Publishing, 2022. Series Title: Lecture Notes in Computer Science.

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

[6] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Reveillere. IBBE-SGX: Cryptographic group access control using trusted execution environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 207–218. IEEE, 2018.

[7] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Reveillere. Anonymous and confidential file sharing over untrusted clouds. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–2110. IEEE, 2019.

[8] L. Desausoi. Building a secure and auditable Personal Cloud. Master's thesis, Louvain School of Engineering, Louvain, Belgium, 2020.

[9] J. B. Djoko, J. Lange, and A. J. Lee. 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, 2019.

[10] A. L. Ferrara, G. Fachsbauer, B. Liu, and B. Warinschi. Policy privacy in cryptographic access control. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 46–60. IEEE, 2015. event-place: Verona.

[11] American National Standard for Information Technology. Role based access control. Standard, American National Standards Institute, Inc., February 2004.

[12] B. Fuhry, L. Hirschoff, S. Koesnadi, and F. Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 476–488. IEEE, 2020.

[13] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. 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, 2016.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[15] Y. He, X. Jia, S. Zhang, and L. Chitkushev. EnShare: Sharing files securely and efficiently in the cloud using enclave. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 225–232. IEEE, 2022.

[16] T. Hoang, R. Behnia, Y. Jang, and A. A. Yavuz. MOSE: Practical multi-user oblivious storage via secure enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 17–28. ACM, 2020.

[17] V. C. Hu, D. F. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2014.

[18] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–17. ACM, 2018.

[19] E. Lanckriet, M. Busi, and D. Devriese. $\pi_{\mathbf{RA}}$: A $\pi\text{-calculus}$ for verifying protocols that use remote attestation. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 537–551. IEEE, 2023.

[20] I. Messadi, S. Neumann, N. Weichbrodt, L. Almstedt, M. Mahhouk, and R. Kapitza. Precursor: a fast, client-centric and trusted key-value store using RDMA and intel SGX. In *Proceedings of the 22nd International Middleware Conference*, pages 1–13. ACM, 2021.

[21] A. Muñoz, R. Ríos, R. Román, and J. López. A survey on the (in)security of trusted execution environments. *Computers & Security*, 129:103180, 2023.

[22] A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on intel SGX, 2020. Version Number: 1.

[23] E. Ramirez, J. Brill, M.K. Ohlhausen, J.D. Wright, and T. McSweeny. Data brokers: A call for transparency and accountability. In *Data brokers: A call for transparency and accountability*, pages 1–101. CreateSpace Independent Publishing Platform, January 2014.

[24] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, pages 57–64. IEEE, 2015.

[25] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, volume 2171, pages 137–196. Springer Berlin Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[26] R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[27] M. Schwarz and D. Gruss. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Security & Privacy*, 18(5):18–27, 2020.

[28] C. Segarra, R. Delgado-Gonzalo, and V. Schiavoni. MQT-TZ: Hardening IoT brokers using ARM TrustZone : (practical experience report). In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 256–265. IEEE, 2020.

[29] R. Stajnrod, R. Ben Yehuda, and N. J. Zaidenberg. Attacking TrustZone on devices lacking memory protection. *Journal of Computer Virology and Hacking Techniques*, 18(3):259–269, 2022.