Work-in-Progress: A Sidecar Proxy for Usable and Performance-Adaptable End-to-End Protection of Communications in Cloud Native Applications

1st Stefano Berlato
Fondazione Bruno Kessler
Trento, Italy
sberlato@fbk.eu

2nd Matteo Rizzi
Fondazione Bruno Kessler
Trento, Italy
mrizzi@fbk.eu

3rd Matteo Franzil
Fondazione Bruno Kessler
Trento, Italy
mfranzil@fbk.eu

4rd Silvio Cretti
Fondazione Bruno Kessler
Trento, Italy
scretti@fbk.eu

5th Pietro De Matteis
Dedagroup SpA
Trento, Italy
pietro.dematteis@dedagroup.it

6th Roberto Carbone
Fondazione Bruno Kessler
Trento, Italy
carbone@fbk.eu

***Abstract*—The characteristics of cloud native applications — like the inherent decentralization, the intricate threat model, and the presence of highly dynamic and interconnected microservices — bring forth a number of challenges to the security of the (often sensitive) data exchanged in cloud native applications. Besides, data security is not absolute, and its achievement must be mindful of relevant performance and usability aspects (e.g., minimal overhead, transparency, automation, interoperability with external services). In this work-in-progress paper, we discuss the use of Cryptographic Access Control (CAC) in sidecar proxies as a means to guarantee End-to-End (E2E) protection — in terms of confidentiality and integrity — for communications in cloud native applications, as well as usability and adaptable performance.**

1. Introduction

Cloud native applications (or "applications") are software applications developed and operated following the principles of cloud computing — in particular, scalability, automation, and resiliency. Cloud native applications are usually composed of small, independent, and loosely coupled microservices that communicate with each other and with third parties (e.g., end users, external services) to provide business functions [14]. Communications may be either internal — that is, occurring within (usually private) networks among the microservices — or external — that is, occurring within (usually public) networks with the third parties. Furthermore, communications may be either direct (i.e., point-to-point) or indirect (i.e., mediated by a message broker for, e.g., enhancing modularity) [11].

Intuitively, communications in cloud native applications should be protected, as per the well-known security-by-design and zero trust principles [16]. In other words, cloud native applications often confront a multifaceted threat landscape in which the level of trust assigned to participating parties is inherently limited. Consequently, the lack of assurances on the confidentiality and integrity of communications might expose the data contained therein to a heterogeneous set of threats including external attackers, eavesdroppers (e.g., for communications happening between different zones or computing regions), malicious insiders and tenants [10], honest-but-curious public cloud providers [15], and compromised microservices (which may share common default security configurations, such as digital certificates, and thus be capable of listening to all communications). Besides, when adopted, security mechanisms typically have a negative impact on performance [6]. This impact is exacerbated if applications are deployed in resource-constrained environments — e.g., the Internet of Things (IoT) — or operate in delicate fields (e.g., eHealth, automotive) offering critical business functions (e.g., remote monitoring, cooperative vehicle maneuvering) where quality of service is a key requirement. Finally, the decentralized essence of cloud native applications — whereby often no fully trusted central party can be relied upon — makes it difficult to provide privileged access management to data effectively with traditional centralized security mechanisms [13].

Therefore, in this work-in-progress paper, we discuss the use of Cryptographic Access Control (CAC) to provide End-to-End (E2E) protection of communications in cloud native applications. In brief, CAC consists in employing cryptography to specify and enforce fine-grained Access Control (AC) policies over data. In other words, in CAC, a piece of data (e.g., a network message) is encrypted, and the permission to access the encrypted data is embodied by the corresponding cryptographic decrypting key — which is distributed to authorized agents (e.g., microservices) only according to an AC policy. Concretely, we consider a specific implementation of CAC [4] provided by a security mechanism called CryptoAC and developed as a microservice available as open-source software.[1] In the context of a cloud native application orchestrated with Kubernetes (K8s), CryptoAC can be used as a sidecar proxy[2] automatically injected into each pod.[3] In this way, CryptoAC can guarantee the confidentiality and integrity of data in transit

1. CryptoAC Documentation (https://cryptoac.readthedocs.io/)

2. A sidecar proxy is an additional microservice deployed alongside business microservices in a pod to provide proxying capabilities

3. A pod is a set of one or more logically-related microservices deployed together, i.e., on the same node. Notably, all microservices within a given pod in K8s share the same network stack

in both direct and indirect communications. Moreover, besides automation and ease of deployment, CryptoAC is completely transparent with respect to the application — which does not need to be modified. Notably, CryptoAC can also be distributed externally to cloud native applications as software for desktops and mobile devices (as well as servers). Most importantly, different installations of CryptoAC can be synchronized on the same AC policy, thus providing interoperability and effectively protecting both internal and external communications. Finally, as not all communications may be worth the computational overhead of cryptography (e.g., according to the sensitivity of the data), CryptoAC allows for deciding whether to apply cryptography to communications on a case-by-case basis — adapting its performance accordingly.

The rest of the paper is structured as follows. In Section 2 we provide the background, while in Section 3 we give an overview of how CAC (and CryptoAC) would apply to a real-world cloud native application. We compare CAC (and CryptoAC) with related work in Section 4 and conclude the paper with final remarks and future work in Section 5.

2. Background on Access Control

AC is a fundamental component of any application [1] and is defined as the process of mediating every request to resources managed by an application and determining whether the request should be granted or denied [17]. AC expects the presence of an AC policy (or simply "policy") declaring what agents (e.g., microservices) can perform what actions on what resources — where resources are logical vessels for data (e.g., communication channels). The AC policy is typically defined by an administrator, which usually corresponds to the owner of the resources or the application (e.g., the developers). Finally, the AC policy is formally represented by an AC model giving the semantics for granting or denying users' requests. The software enforcing a policy based on the chosen AC model is called AC enforcement mechanism. Role-Based Access Control (RBAC) [18] is an AC model widely adopted in both academia and industry [5] — and also in K8s.[4] In RBAC, agents are assigned to roles which are in turn assigned to permissions (for executing actions on resources), and agents activate roles to access permissions. The state of a RBAC policy can be described as a tuple $\langle \mathbf{U}, \mathbf{R}, \mathbf{F}, \mathbf{UR}, \mathbf{PA} \rangle$, where $\mathbf{U}$ is the set of agents, $\mathbf{R}$ 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, 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 $\mathbf{OP}$. An agent u can use a permission $\langle f, op \rangle$ if $\exists r \in \mathbf{R} \mid (u, r) \in \mathbf{UR} \wedge (r, \langle f, op \rangle) \in \mathbf{PA}$.

Cryptographic Access Control. AC enforcement mechanisms may be either traditional (i.e., relying on a trusted central party) — which, however, may not always be suitable for cloud native applications, as said in Section 1 — or based on CAC, which is ideal for decentralized applications [3]. Typically, CAC employs symmetric and asymmetric cryptography to encrypt (data contained in) resources and distribute the corresponding cryptographic decrypting keys to authorized agents, respectively. The use of both symmetric and asymmetric cryptography is called *hybrid cryptography* [8]. Authenticated Encryption with Associated Data (AEAD) [2] (e.g., AES-GCM, xsalsa20-poly1305) is usually employed for symmetric cryptography, while different asymmetric cryptosystems and techniques — like Public Key Infrastructure (PKI) and Attribute-Based Encryption (ABE) — may be employed.

RBAC policies can be enforced with CAC, as discussed in, e.g., [8], [4]. As an example, consider the presence of a resource f (e.g., a communication channel), a role r, and an agent u. In CAC, r and u are both equipped with a pair of (asymmetric) public-private keys $(\mathbf{k}_r^{\mathbf{enc}}, \mathbf{k}_r^{\mathbf{dec}})$ and $(\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}})$ for encryption and decryption, respectively. Now, assume that the administrator wants to grant u read access over the content fc of f through r. With this setup, the administrator first generates a symmetric key $\mathbf{k}_f^{\mathbf{sym}}$ to encrypt the data fc contained in f, resulting in $\{fc\}_{\mathbf{k}_f^{\mathbf{sym}}}$. Then, the administrator encrypts $\mathbf{k}_f^{\mathbf{sym}}$ with $\mathbf{k}_r^{\mathbf{enc}}$, resulting in $\{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}}$. Afterward, the administrator encrypts $\mathbf{k}_r^{\mathbf{dec}}$ with $\mathbf{k}_u^{\mathbf{enc}}$, resulting in $\{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}}$. As shown in Table 1, the state of the resulting CAC RBAC policy is encoded with (possibly decorated versions of) the same sets of a traditional RBAC policy. With such a policy, to access fc, u would:

- decrypt $\{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}}$ with $\mathbf{k}_u^{\mathbf{dec}}$, obtaining $\mathbf{k}_r^{\mathbf{dec}}$;
- decrypt $\{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}}$ with $\mathbf{k}_r^{\mathbf{dec}}$, obtaining $\mathbf{k}_f^{\mathbf{sym}}$;
- decrypt $\{fc\}_{\mathbf{k}_f^{\mathbf{sym}}}$ with $\mathbf{k}_f^{\mathbf{sym}}$, obtaining fc.

To write on f (e.g., to send a new message fc' in the communication channel f), u uses $\mathbf{k}_f^{\mathbf{sym}}$ to encrypt fc' before transmitting the new message. The entity storing the CAC RBAC policy state — usually a database — is called Metadata Manager (MM) [3]. Note that the CAC encoding for RBAC described above is a naive encoding provided as an example, and we refer the interested reader to [8], [4] for the complete encoding.

CryptoAC. CryptoAC[5] implements CAC for RBAC as described in [4], being therefore capable of interacting and synchronizing policies with traditional RBAC enforcement mechanisms (e.g., DynSec[6] of the Mosquitto message broker). In fact, CryptoAC allows for specifying an enforcement type $\mathbf{Enf}$ for each resource; the enforcement type can be either cryptographic ($\mathbf{Enf_c}$) or traditional ($\mathbf{Enf_t}$) — where the subscripts "c" and "t" stand for "cryptographic" and "traditional", respectively — hence $(f, \mathbf{Enf}) \in \mathbf{F}$. Intuitively, CryptoAC applies

4. Using RBAC Authorization — Kubernetes (https://kubernetes.io/docs/reference/access-authn-authz/rbac/)

5. CryptoAC Documentation (https://cryptoac.readthedocs.io/)

6. DynSec (https://mosquitto.org/documentation/dynamic-security/)

TABLE 1. EXAMPLE CAC RBAC POLICY STATE

$\mathbf{U}_c = \{(u, \mathbf{k}_u^{\mathbf{enc}})\}$	$\mathbf{R}_c = \{(r, \mathbf{k}_r^{\mathbf{enc}})\}$	$\mathbf{F}_c = \{(f, \mathbf{Enf_c})\}$
$\mathbf{UR}_c = \{(u, r, \{\mathbf{k}_r^{\mathbf{dec}}\}_{\mathbf{k}_u^{\mathbf{enc}}})\}$	$\mathbf{PA}_c = \{(r, f, \mathsf{Read}, \{\mathbf{k}_f^{\mathbf{sym}}\}_{\mathbf{k}_r^{\mathbf{enc}}})\}$	

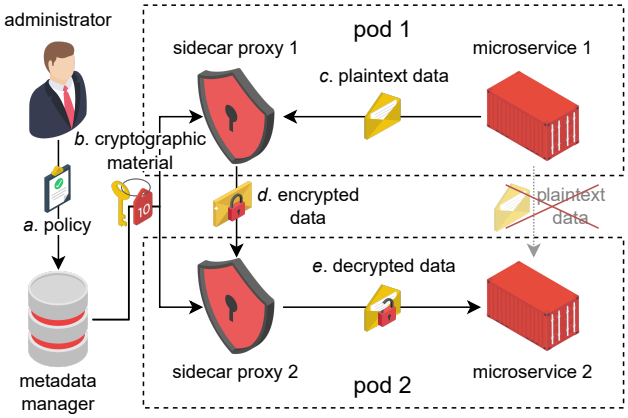

Figure 1. Overview of CAC for cloud native applications

cryptography (as previously described) only in the former case. CryptoAC is open-source, modular, written in the (multiplatform) Kotlin language,[7] and designed as a microservice distributed in a Docker[8] image to guarantee seamless integration with cloud native applications. Moreover, the multiplatform capability of Kotlin allows for native and mobile (e.g., Android and IoS) deployments of CryptoAC. As cryptographic provider, CryptoAC uses Sodium,[9] a modern and portable cryptographic library whose implementation was thoroughly audited.[10]

3. Cryptographic Access Control for Cloud Native Applications

We report in Figure 1 a representation of how CAC can be used in sidecar proxies for protecting communications in cloud native applications. In particular, we consider an application (orchestrated with K8s) composed of two pods — pod 1 and 2 — containing two microservices — microservice 1 and 2, respectively — that communicate with each other. In this context, a sidecar proxy is automatically injected into each pod as a privileged microservice — that is, capable of mediating communications incoming to and outgoing from the pod; the MM (e.g., a Redis data store[11]) can be either orchestrated with K8s or offered as an external service.

First, the AC policy is added to the MM (step *a* in Figure 1). The policy may be either defined by an administrator or automatically inferred from environmental variables or similar deployment information — we leave the analysis of these options as future work. Afterward, the two sidecar proxies fetch the necessary cryptographic material from the MM during the initialization of their pod (step *b*). Finally, microservice 1 securely communicates with microservice 2: sidecar proxy 1 intercepts and encrypts the outgoing plaintext data (step *c*) which are sent toward pod 2 (step *d*), while sidecar proxy 2 intercepts and decrypts the encrypted data — as shown in Section 2 — and deliver the decrypted data to microservice 2 (step *e*).

7. Kotlin (https://kotlinlang.org/)
8. Docker (https://www.docker.com/)
9. Sodium (https://libsodium.gitbook.io/doc)
10. Libsodium Audit Results (https://www.privateinternetaccess.com/blog/libsodium-audit-results/)
11. Redis (https://redis.io/)

3.1. CryptoAC in a Real-World Application

We now discuss the use of CryptoAC in a real-world cloud native application for Enterprise Resource Planning (ERP) developed by Dedagroup and linked to the Co-Innovation Lab CLEANSE[12] More in detail, Dedagroup's application is aimed at organizations in the clothing industry and comprises a set of microservices implementing related business functions (e.g., concerning supply chain and production, sales, and distribution). Notably, Dedagroup's application can integrate with external services (e.g., warehouse management, billing, accounting) and interact with end-users directly (e.g., through desktops and mobile apps). In other words, Dedagroup's application provides a platform built to connect multiple parties — essentially, clothing brands and suppliers — while allowing the integration of external services. To provide modular and flexible communications, Dedagroup's application implements an event-based architecture where events (e.g., concerning clothing products) are made available to participants through message queues handled by a common message broker, Apache Kafka.[13] Access to the message queues is mediated by a RBAC policy — that is, resources correspond to message queues — which, by default, identifies three kinds of roles for each brand: employees, suppliers, and external services.

CryptoAC as Sidecar Proxy in Dedagroup's application. To showcase our approach, we provide a simplified representation of Dedagroup's application in Figure 2: we consider two pods with one microservice each for employees (pod 1) and a supplier (pod 2), a pod for a different unrelated brand (pod n), a pod for the message broker, and an external service for billing. Employees communicate sales data to the billing service (channel 3), while the supplier communicates tracking data (e.g., for shipments) to employees (channel 1) and invoice data to both employees and the billing service (channel 2). While sales and invoice data are deemed sensitive (i.e., $\mathbf{Enf_c}$), tracking data are not (i.e., $\mathbf{Enf_t}$).

12. Co-Innovation Lab — Deda Group (https://www.deda.group/deda/innovazione/co-innovation-lab)
13. Apache Kafka (https://kafka.apache.org/)

TABLE 2. CAC RBAC POLICY STATE IN DEDAGROUP'S APPLICATION*

$\mathbf{U}_c$	=	$\{(m_1, \mathbf{k}^{\mathbf{enc}}_{m_1}), (m_2, \mathbf{k}^{\mathbf{enc}}_{m_2}), (m_n, \mathbf{k}^{\mathbf{enc}}_{m_n}), (es, \mathbf{k}^{\mathbf{enc}}_{es})\}$
$\mathbf{R}_c$	=	$\{(e, \mathbf{k}^{\mathbf{enc}}_{e}), (s, \mathbf{k}^{\mathbf{enc}}_{s}), (ob, \mathbf{k}^{\mathbf{enc}}_{ob}), (b, \mathbf{k}^{\mathbf{enc}}_{b})\}$
$\mathbf{F}_c$	=	$\{(c_1, \mathbf{Enf_t}), (c_2, \mathbf{Enf_c}), (c_3, \mathbf{Enf_c})\}$
$\mathbf{UR}_c$	=	$\{(m_1, e, \{\mathbf{k}^{\mathbf{dec}}_{e}\}_{\mathbf{k}^{\mathbf{enc}}_{m_1}}), (m_2, s, \{\mathbf{k}^{\mathbf{dec}}_{s}\}_{\mathbf{k}^{\mathbf{enc}}_{m_2}}), (m_n, ob, \{\mathbf{k}^{\mathbf{dec}}_{ob}\}_{\mathbf{k}^{\mathbf{enc}}_{m_n}}), (es, b, \{\mathbf{k}^{\mathbf{dec}}_{b}\}_{\mathbf{k}^{\mathbf{enc}}_{es}})\}$
$\mathbf{PA}_c$	=	$\{(s, c_1, \mathsf{Write}, -), (e, c_1, \mathsf{Read}, -), (s, c_2, \mathsf{Write}, \{\mathbf{k}^{\mathbf{sym}}_{c_2}\}_{\mathbf{k}^{\mathbf{enc}}_{s}}), (e, c_2, \mathsf{Read}, \{\mathbf{k}^{\mathbf{sym}}_{c_2}\}_{\mathbf{k}^{\mathbf{enc}}_{e}}), (b, c_2, \mathsf{Read}, \{\mathbf{k}^{\mathbf{sym}}_{c_2}\}_{\mathbf{k}^{\mathbf{enc}}_{b}}), (e, c_3, \mathsf{Write}, \{\mathbf{k}^{\mathbf{sym}}_{c_3}\}_{\mathbf{k}^{\mathbf{enc}}_{e}}), (b, c_3, \mathsf{Read}, \{\mathbf{k}^{\mathbf{sym}}_{c_3}\}_{\mathbf{k}^{\mathbf{enc}}_{b}})\}$

*Users: *m* stands for microservice, *es* for external service. Roles: *e* stands for employees, *s* for supplier, *ob* for other brand, *b* for billing. Resources: *c* stands for channel; "-" represents an empty value.

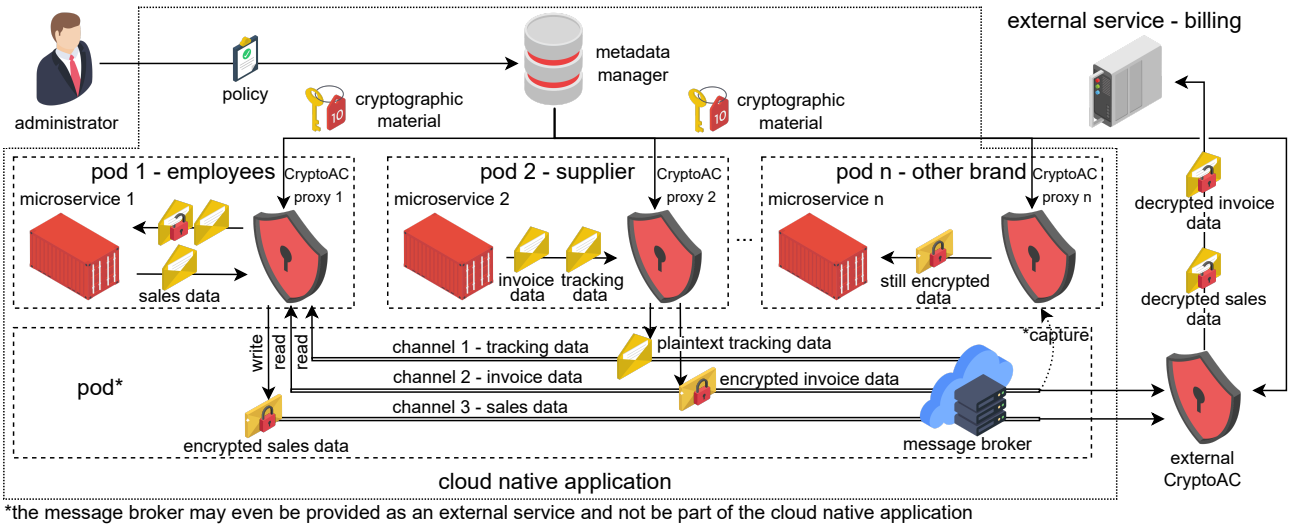

Figure 2. Our approach applied to Dedagroup's application

With these considerations, the state of the resulting CAC RBAC policy would be as shown in Table 2.

Discussion. As shown in Figure 2 CryptoAC automatically encrypts and decrypts data in transit in Dedagroup's application according to the CAC RBAC policy, providing true E2E protection. In particular, only microservices 1 and 2 and the external service for billing can read the content of the messages exchanged over channel 2, while all other parties (e.g., the message broker and microservice n) cannot. We highlight that, even if the CryptoAC proxy n is able somehow to capture messages exchanged over channel 2 (e.g., by sniffing network traffic or colluding with the message broker), it cannot possibly decrypt them since it lacks the necessary cryptographic material, as shown in Table 2 and described in Section 2. Similarly, only microservice 1 and the external service for billing can read the content of the messages exchanged over channel 3, while messages exchanged over channel 1 — which do not contain sensitive data — are not encrypted, avoiding unnecessary cryptographic computations, relieving overhead, and improving performance. We remark that the injection of CryptoAC is automatic and transparent to the microservices of Dedagroup's application. Moreover, if the application scales, new instances of CryptoAC are created with each new pod and configured during the initialization of the pod according to the AC policy. Finally, CryptoAC is not limited to K8s and can easily interoperate with external services.

4. Related Work

The problem of securing data in transit in cloud native applications has been tackled by several researchers and practitioners, resulting in a rich ecosystem of security mechanisms providing excellent solutions to this problem — e.g., see the Cloud Native Computing Foundation (CNCF).[14] One of the most commonly adopted technological patterns among these security mechanisms is the service mesh [12]. In essence, a service mesh allows for controlling how different services — where a service usually corresponds to a microservice or a set of logically related microservices (e.g., a pod) — in cloud native applications exchange data by abstracting the logic governing service-to-service communication. Typically, a service mesh expects a (network) proxy running as a sidecar alongside each service. Besides guaranteeing security, a service mesh allows for gathering performance metrics (to, e.g., optimize communication) and collecting logs and analytics (to, e.g., provide observability). The proxies operate in the data plane — that is, at the same level where data of cloud native applications transit — and are managed by a control plane, which does not handle data but instead acts as a (usually single and centralized) source of truth for the configuration of the proxies.

Below, we compare CryptoAC with 4 popular security mechanisms for service mesh, i.e., Kuma,[15] Istio,[16] Consul,[17] and Linkerd,[18] reporting the results of the comparison in Table 3 — in the table, we consider those aspects already elicited in Section 3. Kuma, Istio, and Consul consist of control planes for service mesh using Envoy[19] as sidecar proxy, while Linkerd consists of a control plane for service mesh using an ad-hoc sidecar proxy named Linkerd2-proxy.[20] We choose Istio, Linkerd, and Consul as they are the most widely adopted security mechanisms for service mesh according to the CNCF[21] — we exclude Traefik[22] (which does not use sidecars proxies), AWS App Mesh[23] (which is not open source),

14. Cloud Native Computing Foundation (https://www.cncf.io/)

15. Kuma (https://kuma.io/)
16. Istio (https://istio.io/)
17. HashiCorp Consul (https://www.consul.io/)
18. Linkerd (https://linkerd.io/)
19. Envoy proxy (https://www.envoyproxy.io/)
20. Why Linkerd doesn't use Envoy (https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/index.html)
21. Service meshes are on the rise – but greater understanding and experience are required — CNCF (https://www.cncf.io/blog/2022/05/17/service-meshes-are-on-the-rise-but-greater-understanding-and-experience-are-required/)
22. Traefik (https://traefik.io/traefik/)
23. AWS App Mesh (https://aws.amazon.com/app-mesh/)

and Open Service Mesh[24] (which has been archived by the CNCF). Moreover, we consider Kuma since it is an emerging and interesting CNCF sandbox project developed for multi-cloud and universal environments.

Point-to-point protection. That is, the confidentiality and integrity of network messages are guaranteed point-to-point (i.e., direct communication, as described in Section 1). Kuma, Istio, Consul, and Linkerd provide such protection with mutual TLS, whereas CryptoAC provides such protection with CAC (see Section 3).

E2E protection. That is, the confidentiality and integrity of network messages are guaranteed end-to-end (i.e., indirect communication, as described in Section 1). Unlike CAC, TLS cannot protect network messages against honest-but-curious agents mediating communication at the application level (e.g., like the message broker in Section 3.1), as also argued in [19], [4]. Therefore, security mechanisms relying on TLS for protecting communications (like Kuma, Istio, Consul, and Linkerd) are intrinsically incapable of providing true E2E protection.

Dynamic fine-grained AC. That is, it is possible to specify, enforce, and dynamically modify fine-grained AC policies over resources. Kuma allows for defining (dynamic but coarse-grained) AC policies over network communication based on source and destination services only.[25] Linkerd follows an analogous approach,[26] and Consul uses AC lists associating one or more policies with a token which is then distributed among authorized services.[27] Istio, instead, considers further aspects such as namespaces, ports, (HTTP) methods, and conditions specified over additional attributes, hence allowing for specifying and enforcing fine-grained AC policies.[28]

Not limited to K8s. That is, whether the use of the security mechanism is limited to K8s. Similarly to CryptoAC (see Section 3), Kuma can run in cloud native applications orchestrated with K8s but also in generic applications via the universal Kuma Application Programming Interface (API) server and Kuma resources — hosted in a dedicated PostgreSQL database acting as MM (similarly to the MM described in Section 3).[29] Instead, Istio is design primarily for K8s, even though some of its security features can be used in generic applications.[30] Finally, Consul[31] and Linkerd[32] allows for connecting external services to their service mesh within K8s, although following a not always straightforward process and often requiring the installation of additional software (e.g., SPIRE for Linkerd) and Docker support.

Interoperability. That is, whether different deployments of the security mechanism (e.g., in different K8s clusters or environments) can interoperate with each other. Kuma supports different deployment modes (e.g., multi-zone, multi-tenancy, multi-cloud),[33] similarly to the cross-cluster interoperability feature of Istio,[34] Linkerd,[35] and Consul.[36] As shown in Section 3, CryptoAC is natively interoperable, potentially even with end-users' devices.

Adaptable performance. That is, it is possible to decide what resources to encrypt and what resources not to encrypt to relieve the cryptographic computational overhead. Istio and Linkerd partially support this possibility by allowing for, respectively, deciding whether to use TLS between pairs of services[37] and specifying for what ports bypass the proxy,[38] while Kuma and Consul seemingly encrypt all communication.

To summarize, as shown in Table 3, the use of CAC in cloud native applications allows for higher security — in particular, true E2E protection and enforcement of fine-grained AC policies in a distributed fashion — adaptable performance and easy interoperability with external services. Nevertheless, we acknowledge and remark that Kuma, Istio, Consul, and Linkerd are widely popular and mature security mechanisms for service mesh still capable of enhancing the security of data in transit in cloud native applications while providing a large number of advanced features (especially for what concerns network observability, optimization, and load balancing) which we do not discuss here. In this sense, CAC could even be integrated with security mechanisms for service mesh and K8s, e.g., as a standalone sidecar proxy or plugin for Envoy.

24. Open Service Mesh (https://openservicemesh.io/)

25. General notes about Kuma policies (https://kuma.io/docs/2.6.x/policies/general-notes-about-kuma-policies/)

26. Authorization Policy — Linkerd (https://linkerd.io/2.15/features/server-policy/)

27. Access Control List (ACL) - Overview — Consul — HashiCorp Developer (https://developer.hashicorp.com/consul/docs/security/acl)

28. Istio / Authorization Policy (https://istio.io/latest/docs/reference/config/security/authorization-policy/)

29. Architecture — Kuma (https://kuma.io/docs/2.6.x/introduction/architecture/)

30. Istio / Deployment Models (https://istio.io/latest/docs/ops/deployment/deployment-models/)

31. External Services to Consul (https://developer.hashicorp.com/consul/docs/k8s/deployment-configurations/clients-outside-kubernetes)

32. Adding non-Kubernetes workloads to your mesh — Linkerd (https://linkerd.io/2.15/tasks/adding-non-kubernetes-workloads/)

33. Multi-zone deployment — Kuma (https://kuma.io/docs/2.6.x/production/deployment/multi-zone/)

34. Istio / Multi-cluster Traffic Management (https://istio.io/latest/docs/ops/configuration/traffic-management/multicluster/)

35. Multi-cluster communication — Linkerd (https://linkerd.io/2.15/features/multicluster/)

36. Consul Across Multiple Clusters (https://developer.hashicorp.com/consul/docs/k8s/deployment-configurations/single-dc-multi-k8s)

37. Istio / Understanding TLS Configuration (https://istio.io/latest/docs/ops/configuration/traffic-management/tls-configuration/)

38. Proxy Configuration — Linkerd (https://linkerd.io/2.15/reference/proxy-configuration/)

TABLE 3. COMPARISON OF CRYPTOAC WITH SECURITY MECHANISMS FOR SERVICE MESH

Criterion	Security Mechanism				
	Kuma	Istio	Consul	Linkerd	**CryptoAC**
point-to-point protection	●	●	●	●	●
end-to-end protection	○	○	○	○	●
dynamic fine-grained AC	◐	●	◐	◐	●
not limited to K8s	●	◐	●	●	●
interoperability	●	●	●	●	●
adaptable performance	○	●	○	●	●

5. Conclusion

In this work-in-progress paper, we proposed and discussed the use of CAC for the E2E protection of communications in cloud native applications. As presented in Section 3, the use of CryptoAC — a security mechanism implementing CAC — as a sidecar proxy would provide E2E protection, transparency, adaptable performance, and interoperability with external services in cloud native applications.

Future Work. Besides an engineering effort toward a concrete deployment of CryptoAC in a real-world cloud native application (to, e.g., experimentally evaluate and collect performance metrics like latency, packet size overhead, scalability, and throughput), we can identify a number of compelling research directions stemming from our proposal of using CAC in cloud native applications. First, CAC is often used to protect data when at rest, e.g., stored in a database hosted in the cloud [8], [4]. In this context, it would be interesting to design a security mechanism employing CAC to guarantee the confidentiality and integrity of data in cloud native applications when both in transit and at rest, thus achieving a higher level of protection. Intuitively, data may also need to be protected when in use (i.e., when being processed). To provide truly comprehensive protection, we note that CAC can arguably be integrated with already existing cryptographic-based solutions to protect the confidentiality of data in use and the processing itself, such as homomorphic encryption and functional encryption [9]. For instance, data may be encrypted with functional encryption, and the secret keys allowing to compute functions over the encrypted data (i.e., the secret functional encryption keys) may be distributed through CAC. Then, as mentioned in Section 3, AC policies could be automatically inferred from, e.g., deployment information or behavioral models — detailing the interactions among a set of microservices — derived from logs and event traces through process mining [7]. Finally, to further adapt performance, it may be interesting to investigate a context-aware methodology capable of evaluating the importance and sensibility of a piece of data (e.g., a network message) on the fly and accordingly decide the most appropriate security mechanism (e.g., whether to protect that data with CAC).

Acknowledgements

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Yosef Ashibani and Qusay H. Mahmoud. Cyber physical systems security: Analysis, challenges and solutions. *Computers & Security*, 68:81–97, 2017.

[2] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.

[3] Stefano Berlato, Roberto Carbone, Adam J. Lee, and Silvio Ranise. Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. *ACM Trans. Priv. Secur.*, 25(1), nov 2021.

[4] Stefano Berlato, Umberto Morelli, Roberto Carbone, and Silvio 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] Fangbo Cai, Nafei Zhu, Jingsha He, Pengyu Mu, Wenxin Li, and Yi Yu. Survey of access control models and technologies for cloud computing. *Cluster Computing*, 22:6111–6122, 2019.

[6] Marco Centenaro, Stefano Berlato, Roberto Carbone, Gianfranco Burzio, Giuseppe Faranda Cordella, Roberto Riggio, and Silvio Ranise. Safety-related cooperative, connected, and automated mobility services: Interplay between functional and security requirements. *IEEE Vehicular Technology Magazine*, 16(4):78–88, 2021.

[7] Luca Compagna, Daniel Ricardo Dos Santos, Serena Elisa Ponta, and Silvio Ranise. Aegis: Automatic enforcement of security policies in workflow-driven web applications. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 321–328. ACM, 2017.

[8] William C. Garrison, Adam Shull, Steven Myers, and Adam 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. event-place: San Jose, CA.

[9] Christian Gottel, Rafael Pires, Isabelly Rocha, Sebastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 133–142. IEEE, 2018.

[10] Abdelhakim Hannousse and Salima Yahiouche. Securing microservices and microservice architectures: A systematic mapping study. *Computer Science Review*, 41:100415, 2021.

[11] Isil Karabey Aksakalli, Turgay Celik, Ahmet Burak Can, and Bedir Tekinerdogan. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180:111014, 2021.

[12] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225. IEEE, 2019.

[13] Theo Lynn, John G. Mooney, Brian Lee, and Patricia Takako Endo. The cloud-to-thing continuum: Opportunities and challenges in cloud, fog and edge computing. In *The Cloud-to-Thing Continuum*, Palgrave Studies in Digital Business & Enabling Technologies. Springer International Publishing, 2020.

[14] Anelis Pereira-Vale, Eduardo B. Fernandez, Raúl Monge, Hernán Astudillo, and Gastón Márquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200, 2021.

[15] 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.

[16] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture, 2020.

[17] Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171, pages 137–196. Springer Berlin Heidelberg, 2000.

[18] Ravi Sandhu. Access control: principle and practice. *Advances in Computers*, 46:237 – 286, 10 1998.

[19] Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio 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.