Improving Microservices Security

Amir Javadpour*‖, Forough Ja'fari§**, Tarik Taleb¶††,Qize Guo*ˣ, Chafika Benzaïd‡‡‡,
Luis Rosa†, and Luis Cordeiro†

*ICTFICIAL Oy, Espoo, Finland †OneSource, Coimbra, Portugal
‡Faculty of Information Technology and Electrical Engineering, University of Oulu, Oulu, Finland
§Department of Computer Engineering, Sharif University of Technology, Iran
¶Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Bochum, Germany
‖a.javadpour87@gmail.com (Corresponding Author) **azadeh.mth@gmail.com
††tarik.taleb@rub.de ‡‡chafika.benzaid@oulu.fi
ˣQize.Guo@rub.de

***Abstract*—To support flexibility and scalability, 5G networks have embraced microservice-based architectures, which require secure and efficient inter-service communication. This is managed by the service mesh layer, which is now a growing target for cyberattacks. While existing platforms like Istio and NGINX use mutual TLS (mTLS) to secure communications, mTLS imposes considerable resource overhead, undermining the goals of scalability and lightweight operation. To overcome this challenge, we propose an Encryption as a Service (EaaS) framework for Kubernetes that mitigates common attacks such as man-in-the-middle, distributed denial-of-service (DDoS), and eavesdropping. Experimental analysis shows that EaaS significantly improves response time and reduces adversary success compared to traditional microservice-side cryptographic handling, with gains varying across different scenarios and cryptographic/deception configurations. While higher EaaS replication slightly increases CPU and memory usage, it leads to better security outcomes and faster service performance. The successful real-world implementation and deployment of the EaaS framework further corroborated these findings.**

***Index Terms*—Kubernetes, Service Mesh, Encryption as a Service (EaaS), Security.**

I. Introduction

To enhance the scalability and modularity of 5G networks, services are decomposed into microservices [1]. This microservice architecture enables independent deployment and flexible resource allocation. However, enabling secure and efficient communication among microservices remains a challenge. Modifying the microservices directly to support communication is inefficient, as it requires extensive changes to existing codebases. To address this, a service mesh layer is introduced [2, 3, 4], where communication tasks are offloaded to proxies (sidecars) attached to each microservice. These sidecars intercept both outgoing and incoming traffic, apply the required transformations, and forward the data accordingly [5, 6].

While the sidecar model is common, alternative service mesh architectures, such as library-based and node-based designs, also exist [7, 8, 9], although their operational concepts are largely similar. A critical component of secure microservice communication is Mutual Transport Layer Security (mTLS), where proxies authenticate endpoints and encrypt data using shared keys. Service mesh platforms like Istio [10] and NGINX [11] widely implement this protocol [12, 13, 14, 15].

Despite its security benefits, mTLS introduces significant resource overhead, especially on resource-constrained pods within Kubernetes. This overhead may contradict the lightweight nature of microservices and hinder scalability.

To overcome these limitations, we propose an **Encryption as a Service (EaaS)** framework that secures service mesh communications in Kubernetes and mitigates three common attack vectors. The high-level architecture of the proposed framework is depicted in Figure 1. **The main contributions of this paper are:**

- We analyze three attack scenarios targeting service mesh environments: Man-in-the-Middle (MitM), Distributed Denial of Service (DDoS), and eavesdropping, and explain their impact on the confidentiality of microservice communication.
- We propose specific countermeasures for each attack and develop a unified mitigation strategy based on secure message structures and encryption control mechanisms.
- We design and implement the EaaS framework that integrates these protections into Kubernetes-based deployments.

In Section II, we examine three potential attack scenarios targeting service mesh. Section III introduces the proposed EaaS framework, and in Section IV, we evaluate its security effectiveness. Section V gives a case study, and Section VI summarizes the key findings.

II. Attacks and Proposed Solutions

In this section, we discuss three main attack scenarios against the confidentiality and availability of service mesh

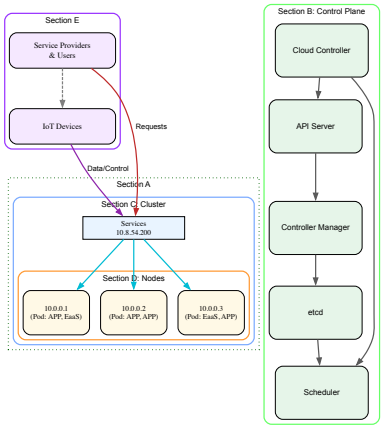


Fig. 1: A brief architecture of Securing Service Mesh in Kubernetes

and the proposed solutions for each of them.

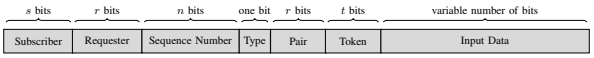
Scenario 1 (Man in the Middle attack)**.** *In this attack scenario, the adversary changes the IP address of their pods, to one of the pods that are involved in the service chain. This may occur because the source microservice sends its data to the malicious pod instead of the intended destination pod. In this condition, the data is revealed to the adversary. It is worth noting that Kubernetes does not handle IP address conflicts, and as a result, this attack poses a valid threat against Kubernetes.*

Solution 1 (Man in the Middle attack)**.** *In the proposed solution, data transmitted within the service mesh is encrypted, allowing only the intended microservice to decrypt and access it. A trusted third-party component manages this process by generating and distributing secret keys exclusively to authorized microservices.*

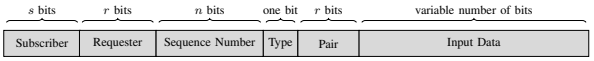
Scenario 2 ((Distributed) Denial of Service attack)**.** *In this attack, the adversary floods a pod/microservice with many requests. When the target pod receives them, it has to decrypt them, which results in a huge resource consumption. When the pod runs out of resources, the microservice cannot handle legitimate requests either, resulting in a denial-of-service attack.*

Solution 2 ((Distributed) Denial of Service attack)**.** *Utilizing the concept of EaaS can help us effectively use the resources. In other words, if we consider third-party components to perform the cryptographic processes, a microservice outsources a significant portion of its processing load. In the case of facing flooded requests, it is only responsible for checking a few bits to determine whether the request is legitimate.*

Scenario 3 (Eavesdropping attack)**.** *When a microservice sends its raw data to the EaaS components to encrypt it, during this transmission, there is also the possibility of being eavesdropped on by the adversary. In other words, similar to the man-in-the-middle attack, the adversary may be located between the microservices and the EaaS components, allowing them to read the confidential data.*



(a) Messages from microservices that are the source of a communication



(b) Messages from microservices that are the destination of a communication

Fig. 2: The structure of the defined messages in the proposed solution

Solving the trade-off between security and its cost is always challenging. However, we can make the attacks more costly to the adversaries. To waste the adversary's resources in Scenario 3, we have proposed the following solution.

Solution 3 (Eavesdropping attack)**.** *To increase the cost of eavesdropping, fake messages are exchanged between microservices and EaaS components. Only one message per batch is genuine, identified by a secret token. This forces the adversary to process all messages without knowing which is real.*

By combining the three proposed solutions, we present a unified security approach that utilizes a third-party EaaS framework structured as microservices. EaaS components handle cryptographic operations for non-EaaS microservices. EaaS first encrypt outgoing traffic before reaching the next service, while incoming traffic is decrypted before delivery. The message structure between EaaS and non-EaaS components is shown in Figure 2. Two message types are defined: one for encryption requests (a) and another for decryption (b). Each begins with s bits for the subscriber IDidentifying the service providerand r bits for the requester ID, representing the non-EaaS microservice. The next n bits form a sequence number, incremented with each request. Together, these fields form the request identifier. A 1-bit field indicates the request type: 0 for encryption, 1 for decryption. The p-bit pair field specifies the other party in the communication (source or destination), and like the requester ID, uses r bits. Encryption requests also include a token for authentication. Hence, in t bits are considered for the token, which is not considered by the other message type. Finally, the last field of both messages is the input data, which is the raw data in encryption requests and the encrypted data in decryption requests. The number of bits in this field depends on the block sizes of the considered cryptographic algorithms.

$(54)_{10}$	$(1)_{10}$	$(9)_{10}$	enc	$(2)_{10}$	fake	random data
00110110	00001	000000001001	0	00010	00	11000000111010
$(54)_{10}$	$(1)_{10}$	$(9)_{10}$	enc	$(2)_{10}$	real	real data
00110110	00001	000000001001	0	00010	01	01101000100010
$(54)_{10}$	$(1)_{10}$	$(9)_{10}$	enc	$(2)_{10}$	fake	random data
00110110	00001	000000001001	0	00010	10	00010000000011
$(54)_{10}$	$(1)_{10}$	$(9)_{10}$	enc	$(2)_{10}$	fake	random data
00110110	00001	000000001001	0	00010	11	10010111001111

Fig. 3: The sample messages transferred between a non-EaaS microservice and an EaaS microservice

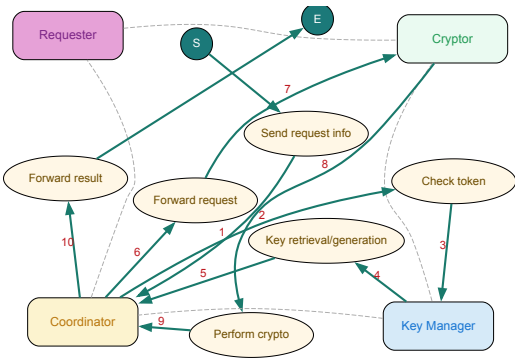


Fig. 4: The registration workflow in the proposed EaaS framework.

Now, we give an example of how the messages are constructed. Assume that the service provider owning the sample service in Figure 3 has an identifier of 54, with $s = 8$ and $r = 5$, and a token equal to 01, which means $t = 2$. When its first microservice wants to send data, say 01101000100010, to the second microservice, and it is its ninth request with $n = 12$, four messages as shown in Figure 3 are generated.

III. Proposed Framework

The proposed EaaS framework is composed of four core components: Coordinator, Registrar, Key Manager, and Cryptor. The Coordinator oversees request handling and orchestrates interactions among the other components. The Registrar assigns cryptographic services to providers based on their specified requirements and desired level of deception. The registration workflow is illustrated in Figure 4.

A service provider registers by sending its identifier (S_ID) and the desired deception factor (D_Factor) to the registrar. The registrar generates a token accordingly and forwards it, along with the subscriber's ID, to a coordinator, which then assigns resources and returns its own ID (C_ID). The registrar relays the coordinator's ID and token back to the subscriber, who can then share this information with its microservices.

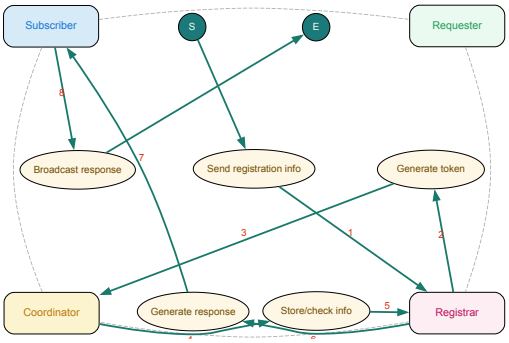


Fig. 5: The service request workflow in the proposed EaaS framework.

The key manager generates cryptographic keys and assigns them to specific pairs of microservices. The cryptor simply performs the encryption and decryption processes.

The workflow for requesting a cryptographic service is shown in Figure 5.

In the first step, a microservice (the requester in our framework) sends an identifier (ID) comprising its subscriber ID, microservice ID, and request sequence number along with the request type (Type), the target communication pair (Pair), and input data (Inp) to the coordinator assigned to its subscriber (Step 1). For encryption requests, a token is also included. The coordinator validates the token; if it does not match the one assigned to the subscriber, the request is discarded. Otherwise, the coordinator forwards the ID and Pair to the Key Manager (Step 2). The Key Manager either generates or retrieves the corresponding key (Key) and returns it to the coordinator (Step 3). Upon receiving the key, the coordinator packages it with the ID, request type, and input data, then sends it to a Cryptor (Step 4). The Cryptor performs encryption or decryption and returns the output (Out) to the coordinator (Step 5), which then forwards it to the original requester (Step 6). For decryption requests, the token is also returned to the requester.

The coordinator maintains two databases: one for subscriber information and another for requests. Upon receiving a packet, it extracts the source and payload, then performs actions depending on the source component. If the source is a registrar, the coordinator extracts the subscriber identifier and token from the payload, stores them in the subscribers' database, finds an appropriate coordinator for the subscriber, and sends the coordinator information back to the registrar. If the source is a key manager, the coordinator extracts the request identifier and key from the payload, retrieves request details from the requests database, selects a cryptor, and forwards the request information and key to it. If the source is a cryptor, the coordinator extracts the request identifier and output, then retrieves associated subscriber

and request information. For encryption requests, the output is sent directly to the requester. Otherwise, to protect privacy, the coordinator sends multiple messages, including fake data based on the subscriber's token length. If the source is a subscriber, indicating a request from one of its microservices, the coordinator extracts request details and verifies the token if the request is of encryption type. If the token is invalid, the packet is ignored. Otherwise, the request is stored, and the relevant key manager is notified with the request information. If the packet originates from an unknown source, an error message is sent back.

IV. Security and Overhead Analysis

To deploy our proposed solution in a Kubernetes environment, we developed a Python-based proxy using the `scapy` module. This proxy captures network packets and forwards them to the appropriate EaaS component. When microservices run on Kubernetes pods, the proxy is deployed alongside them on the corresponding pods.

To evaluate the security effectiveness of our framework, we implemented a distributed database service in which microservices exchange passwords to access the database. We simulated an attack by compromising one of the pods and allowing the adversary a limited time to retrieve the password and access the database. An attack is considered successful if the adversary manages to read at least one database.

To evaluate the proposed framework, we implemented and compared various symmetric cryptographic algorithms, including DES, AES, and Blowfish. Three deployment scenarios were considered based on the number of EaaS component replicas. In scenario α, each component has a single replica. Scenario β includes one coordinator and two replicas each for the Key Manager and Cryptor. Scenario λ scales the system with five replicas of each component. In all scenarios, if no available component is found to process a request, the request is dropped and the raw data is forwarded to the next microservice.

For baseline comparison, we also implemented two additional setups: one with no encryption and another with local encryption, where microservices perform cryptographic operations themselves. In the latter case, if a microservice lacks sufficient resources, it transmits the data in plain form.

The impact of different deception factor values on the adversary's success rate is illustrated in Figure 6a.

The first point about this chart is its descending nature. As the value of the deception factor increases, the number of successful attacks decreases. This is because the adversary must process and verify the links for a longer period, as the number of fake requests increases when the deception factor is high. The average values of the reported results show that our framework reduces the adversary's success rate by about 61% and 45% compared to the case that no encryption method and only local encryption are utilized, respectively. The reason that the baselines related to No Encryption and Local Encryption are horizontal is that the changes in deception factors do not affect them. Moreover, the adversary's success rate in Local Encryption scenarios is not 100% because some of the data can find available resources and are encrypted. Finally, we can see that the security approach in scenario λ outperforms scenario α and β by about 24%. This is because the EaaS load is distributed among more replicas in this scenario and almost all the transferred data can be encrypted in this case.

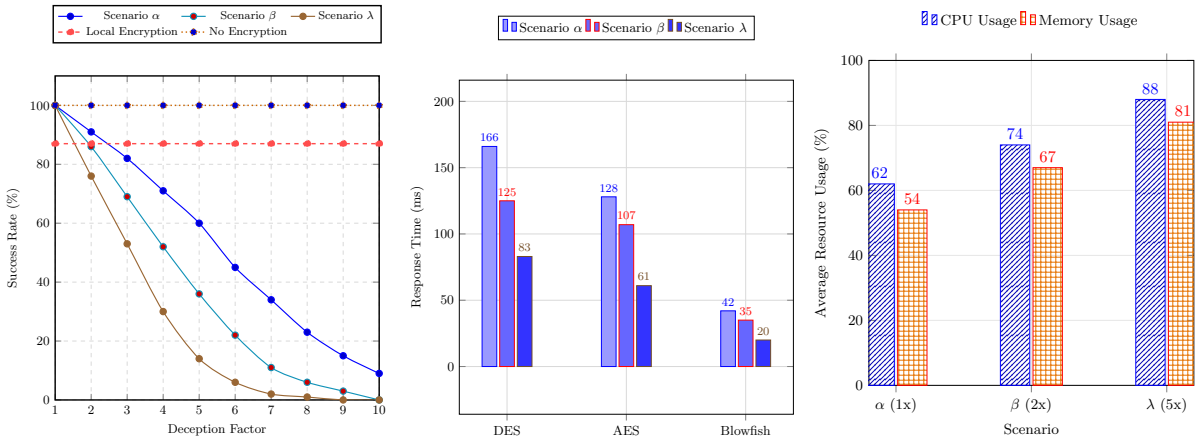
To evaluate the overhead of our proposed EaaS framework, we have compared the response times of non-EaaS services in Figure 6b. We can see that the service response time when our framework implements DES is higher than when it implements AES. The response time for both of these cases is also higher than the case of Blowfish as the cryptography algorithm served by our framework. The other point about this graph is that the response time when scenario λ is applied is lower than that of scenarios α and β. This is because the load is distributed among a higher number of replicas in scenario λ.

To complement the security analysis, we examined how increasing the number of EaaS component replicas (α, β, λ) affects system resource usage. As shown in Figure 6c, higher replication improves security and performance but significantly raises both CPU and memory consumption. For example, CPU usage increases from 62% in scenario α to 88% in scenario λ. This highlights a clear trade-off, and the number of replicas should be chosen based on both security needs and available resources.

To clearly visualize the trade-offs between security, performance, and resource usage in our proposed EaaS framework, Figure 7 illustrates a stacked area chart based on realistic values extracted from our experiments (see Figures 10–12). The three plotted metrics include:

- **Security Improvement**: Derived by inverting the adversary's success rate under scenario λ. As the deception factor increases, the number of fake messages confuses the attacker, reducing successful interception and resulting in higher normalized security levels.
- **Response Time Reduction**: Based on the normalized and inverted response time values when using Blowfish in scenario λ (lowest among tested algorithms). Offloading cryptographic operations to multiple EaaS replicas significantly reduces latency as the load is distributed.
- **Resource Usage**: Calculated using the normalized CPU consumption of non-EaaS microservices in scenario λ. Increasing the number of fake or legitimate encryption/decryption operations naturally increases CPU load.

As shown in the figure, increasing the deception factor re-



(a) Success rate of attacks regarding different deception factors.

(b) Response time for different cryptographic algorithms.

(c) Average CPU and memory usage for non-EaaS microservices.

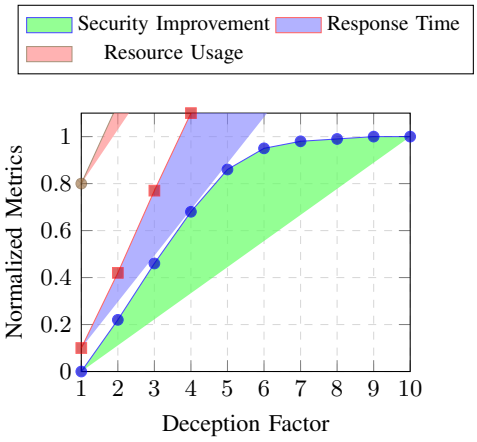
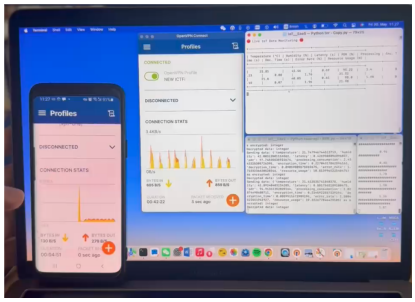
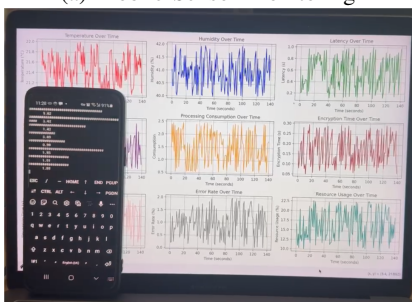


Fig. 7: Trade-off between security improvement, response time, and resource usage.



(a) Mobile Sensor Monitoring



(b) Encrypted Client-Server Communication

Fig. 8: Live deployment and monitoring of the EaaS framework. (a) Real-time monitoring of environmental and network metrics from a rooted Android phone as an IoT sensor. (b) Secure, end-to-end encrypted data exchange between the IoT device and the cloud.

sults in improved security and reduced response time, but at the cost of higher resource consumption. This visual trade-off helps determine an optimal operating point depending on the system's security requirements and infrastructure constraints.

V. Case Study: EaaS Integration with IoTSCP

In addition to the mobile IoT sensor use case, we further evaluated the practicality and scalability of the EaaS framework within an IoT-based Smart City Platform (IoTSCP). To demonstrate the practical effectiveness and flexibility of the proposed Encryption as a Service (EaaS) framework, we implemented a real-world deployment centered on a rooted Android phone functioning as a programmable IoT sensor node within a cloud-native microservice environment. The mobile device was prepared by installing Ubuntu 22, which enabled low-level system access and the development of a custom API interface for sensor data collection and remote control. Through this API, the phone exposes various real-time system and network metrics, including temperature, humidity, latency, and bandwidth, to the monitoring platform. A Python-based passive Quality of Service (QoS) monitoring agent was developed to capture latency, jitter, and bandwidth metrics in real time. Latency is measured

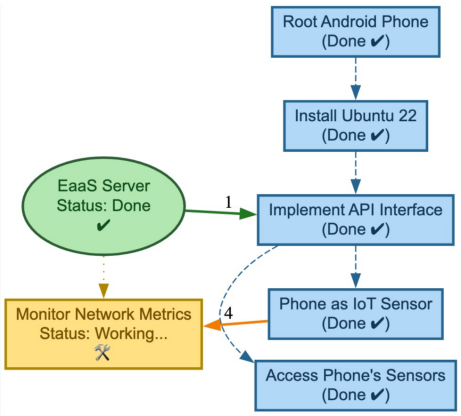


Fig. 9: EaaS-enabled in microservices.

using TCP handshake timings, jitter is calculated as the variation in latency, and bandwidth (uplink/downlink) is passively recorded via operating system counters. This passive approach avoids generating any synthetic network load, ensuring that monitoring does not interfere with normal device or service operation. All metrics collected from the phone are formatted into timestamped payloads and transmitted securely to the cloud using the EaaS API. The EaaS server handles all cryptographic operations, performing encryption and decryption transparently for each submitted sample. Encryption algorithms and parameters can be reconfigured on the fly by simply updating the API request, enabling rapid adaptation to new security requirements or compliance needs. Figure 8 shows the deployment: panel (a) features the mobile sensor's live monitoring dashboard, and panel (b) presents real-time encrypted client-server communication and performance metrics. The EaaS framework improved response times, reduced adversary success rates, and lowered resource overhead compared to traditional methods.

VI. Conclusion

Service mesh enables communication among microservices but is vulnerable to attacks such as man-in-the-middle, (distributed) denial-of-service, and eavesdropping. This paper proposes an EaaS component providing cryptographic services to non-EaaS microservices. To thwart eavesdroppers, fake requests are also sent. We define the message structure between EaaS and non-EaaS microservices and present a Kubernetes-based framework with detailed algorithms. The reported results show that our platform can reduce the adversary's success rate by at least 45% compared to the scenario of implementing a cryptographic process by the non-EaaS microservices themselves. We plan to improve the results by providing multiple cryptography algorithms simultaneously and utilizing a machine learning model that can determine the optimal algorithm in each situation.

Acknowledgment

This research is partially supported by the European Union's Horizon Europe research and innovation program under the RIGOUROUS project (Grant No. 101095933).

References

[1] M. Baboi, A. Iftene, and D. Gîfu, "Dynamic microservices to create scalable and fault tolerance architecture," *Procedia Computer Science*, vol. 159, pp. 1035–1044, 2019.

[2] A. El Malki and U. Zdun, "Guiding architectural decision making on service mesh based microservice architectures," in *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*. Springer, 2019, pp. 3–19.

[3] A. Javadpour, F. Ja'Fari, T. Taleb, C. Benzaid, L. Rosa, P. Tomás, and L. Cordeiro, "Deploying testbed docker-based application for encryption as a service in kubernetes," in *2024 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2024, pp. 1–7.

[4] M. Matias, E. Ferreira, N. Mateus-Coelho, and L. Ferreira, "Enhancing effectiveness and security in microservices architecture," *Procedia Computer Science*, vol. 239, pp. 2260–2269, 2024.

[5] M. R. S. Sedghpour and P. Townend, "Service mesh and ebpf-powered microservices: A survey and future directions," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 176–184.

[6] A. Javadpour, F. Ja'fari, T. Taleb, Y. Zhao, B. Yang, and C. Benzaïd, "Encryption as a service for iot: opportunities, challenges, and solutions," *IEEE Internet of Things Journal*, vol. 11, no. 5, pp. 7525–7558, 2023.

[7] A. Joshi *et al.*, "Selecting a service mesh implementation for managing microservices," 2023.

[8] A. Javadpour, F. Ja'fari, and T. Taleb, "Encryption as a service: A review of architectures and taxonomies," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2024, pp. 36–44.

[9] J. Robberechts, A. Sinaeepourfard, T. Goethals, and B. Volckaert, "A novel edge-to-cloud-as-a-service (e2caas) model for building software services in smart cities," in *2020 21st IEEE international conference on mobile data management (MDM)*. IEEE, 2020, pp. 365–370.

[10] I. Authors, "Simplify observability, traffic management, security, and policy with the leading service mesh," https://istio.io/, 2024, [Accessed: February 2024].

[11] N. team, "Nginx service mesh documentation," https://docs.nginx.com/nginx-service-mesh/, 2024, [Accessed: February 2024].

[12] L. Calcote and Z. Butcher, *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media, 2019.

[13] R. Amir, "Managing kubernetes traffic with f5 nginx," 2022.

[14] A. Javadpour, F. Ja'fari, T. Taleb, C. Benzaïd, Y. Bin, and Y. Zhao, "Encryption as a service (eaas): Introducing the full-cloud-fog architecture for enhanced performance and security," *IEEE Internet of Things Journal*, 2024.

[15] A. Javadpour, F. Ja'fari, T. Taleb, M. Shojafar, and C. Benzaïd, "A comprehensive survey on cyber deception techniques to improve honeypot performance," *Computers & Security*, vol. 140, p. 103792, 2024.