Improving the Security of Service Mesh in Kubernetes

Amir Javadpour*[‖], Forough Ja'fari[§]**, Tarik Taleb[¶]††, Chafika Benzaïd[‡]‡‡, Luis Rosa[†], and Luis Cordeiro[†]
*ICTFICIAL Oy, Espoo, Finland [†]OneSource, Consultoria Informática, 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

Abstract—**Bringing flexibility and scalability to 5G networks has expanded networking technology to facilitate the split of service into microservices and how they can communicate. The network layer dedicated to this communication is called service mesh, and it has become a new target for cyber adversaries. The existing service mesh infrastructures, such as Istio and NGINX, apply the mutual TLS (mTLS) protocol to the connections in the service mesh layer to protect the confidentiality of the data transferred in this layer. However, the main challenge of implementing mTLS is its resource restriction, which significantly conflicts with the scalability and flexibility goals. Therefore, this paper proposes an Encryption as a Service (EaaS) framework that can be implemented on Kubernetes, mitigating man-in-the-middle, (distributed) denial of service, and eavesdropping attacks against service mesh. The implementation results show that the proposed framework decreases the adversary's success rate by at least 45% compared to the cases of having microservices apply the cryptographic processes by themselves.**

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

I. Introduction

To provide scalability and flexibility, 5G networks split services into microservices [1]. Using microservices to deploy a service, removes the boundaries of resource capacity. These microservices require a way of communicating with each other. However, changing the microservices themselves to support this communication is not suitable. Because we need a huge change to rewrite the previously written programs, which requires a lot of effort. As a result, an additional layer is considered for this communication, which is called service mesh, [2] and all the required changes are applied using a proxy, which is also known as the sidecar, in this layer for each of the microservices. The program serving a microservice, sends its output to that proxy, it is modified by the proxy, and then sent to other microservices. In a reverse way, when the proxy receives a packet, the required changes are performed and then fed into the program [3]. It is worth noting that other than the sidecar-based architecture for service mesh, library-based and node-based architectures also exist [4]. But their general concept is almost the same.

The basic security mechanism for protecting the communications between microservices is the mTLS (mutual Transport Layer Security) protocol. The proxies establish a connection between themselves, ensuring that both end parties are authenticated through mTLS. Based on this protocol, the two parties share a secret key, and the data transferred between them is encrypted and decrypted using this key. Istio [5] and NGINX [6] are two of the commonly used service mesh infrastructures that can apply this protocol to the communication between the microservices [7, 8]. However, applying this protocol faces challenges, such as the resource restriction it brings. In other words, if a pod in Kubernetes wants to perform the cryptographic processes related to mTLS, it may run out of resources. Hence, the pods have to access rich resources, which conflicts with the main goal of microservices, which is to increase flexibility and scalability.

This paper proposes an Encryption as a Service (EaaS) framework that mitigates three different attack scenarios against service mesh in Kubernetes [9, 10, 11, 12]. A brief architecture of a Kubernetes environment that uses this framework is shown in Figure 1. The key contributions of this paper are as follows:

- Discussing three different attacks against service mesh, containing man in the middle, (distributed) denial of service, and eavesdropping attacks, and explaining how they affect the confidentiality of the transferred data.
- Proposing a security solution for each of the discussed attack scenarios, suggesting a general solution to mitigate all of them, and defining the structure of messages that must be transferred among the main components.
- Proposing an EaaS framework that implements the proposed solution in Kubernetes environments.

The remainder of this paper is organized as follows. In Section II we have discussed three different attack scenarios that may be launched against service mesh in Kubernetes environments and then proposed a solution for each. Section III describes our proposed EaaS framework and presents the algorithms each of its components follows. The security and overhead of our proposed security approach are evaluated and analyzed in Section IV, and finally Section V provides a summary of this work and the plans.

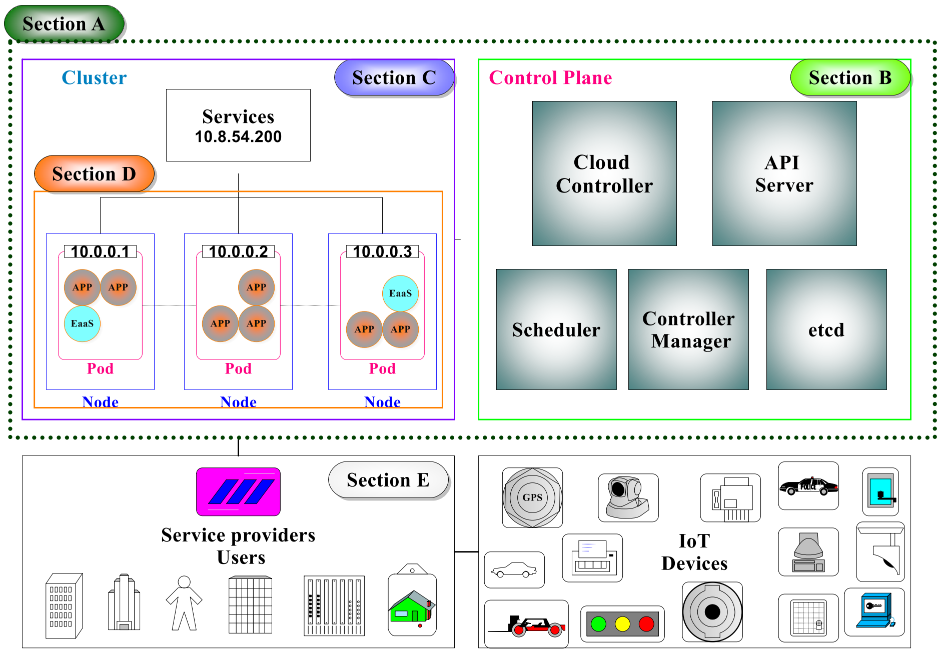


Fig. 1. A brief architecture of the proposed solution in Kubernetes

II. Attacks and Proposed Solutions

In this section, we discuss three main attack scenarios against the confidentiality and availability of service mesh and the proposed solutions for each.

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 involved in the service chain. This may result from the source microservice sending its data to the malicious pod instead of the real destination pod. In this condition, the data is revealed to the adversary. It is worth noting that Kubernetes does not handle the IP address conflict, so this attack is a valid threat against Kubernetes.*

A sample Kubernetes environment and the effect of Scenario 1 on it is shown in Figure 2. A service with three microservices runs on Kubernetes, where each microservice sequentially forwards data: the first to the second, and the second to the third. These are deployed on the second, eighth, and ninth pods with IPs 10.0.1.1, 10.0.1.2, and 10.0.1.3, respectively, each on a separate node. Without attack (Figure 2(a)), the request flow is direct. However, under a man-in-the-middle attack (Figure 2(b)), an adversary assigns a spoofed IP matching the second microservice to the sixth pod. Consequently, the first microservice's output is misrouted to the attacker, who intercepts the data before forwarding it to the legitimate second microservice—remaining undetected while breaching confidentiality.

Now, let us discuss a possible solution for securing service mesh against man-in-the-middle attacks.

Solution 1 (Man in the Middle attack)**.** *In this solution, the data transferred within the service mesh is encrypted so only the desired microservice can decrypt and read it. A third-party component must also be involved in this process to assign secret keys to the authorized microservices and share it with those who are supposed to access it.*

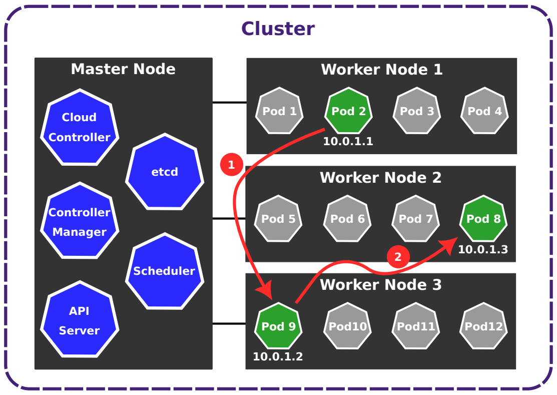
According to Solution 1, the sample Kubernetes described in Figure 2 can be secured against man-in-the-middle attack. When the second pod sends its output by mistake to the sixth pod, as this pod cannot access the related key, the data is not revealed.

The other attack against service mesh, which may occur after applying Solution 1 is (distributed) denial of service.

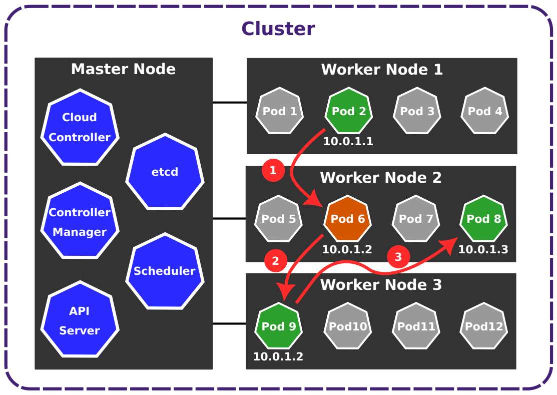
Scenario 2 ((Distributed) Denial of Service attack)**.** *The adversary floods a pod/microservice with many requests in this attack. When the target pod receives them, it has to decrypt them, resulting in a huge resource consumption. When the pod runs out of resources, the microservice cannot handle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

Figure 3 illustrate a sample attack based on Scenario 2. We can see that many of the existing pods are sending requests to the ninth pod. As this pod gets overwhelmed, it cannot distinguish the flooded requests and the legitimate ones, which are sent from the second pod. Hence, its output is not generated and hence, nothing is received by the last microservice located on the eighth pod.

We suggest a solution to solve the (distributed) denial of service attack.



(a) No attack



(b) Under attack

Fig. 2. A sample Kubernetes environment that may be targeted by a man in the middle attack

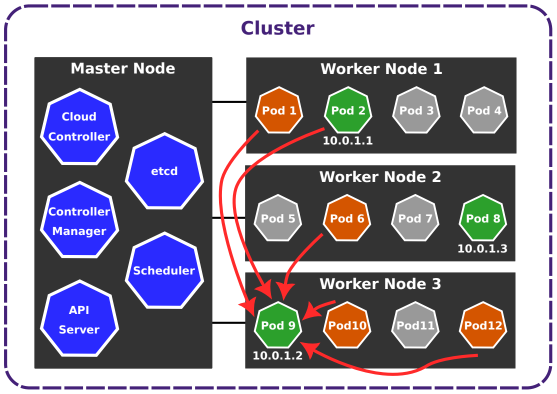


Fig. 3. A sample Kubernetes environment targeted by a distributed denial of service attack

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 much of its processing load. In the case of facing flooded requests, it is just responsible for checking a few numbers of bits to determine whether or not the request is legitimate.*

Implementing an EaaS framework for service mesh is a main challenge.

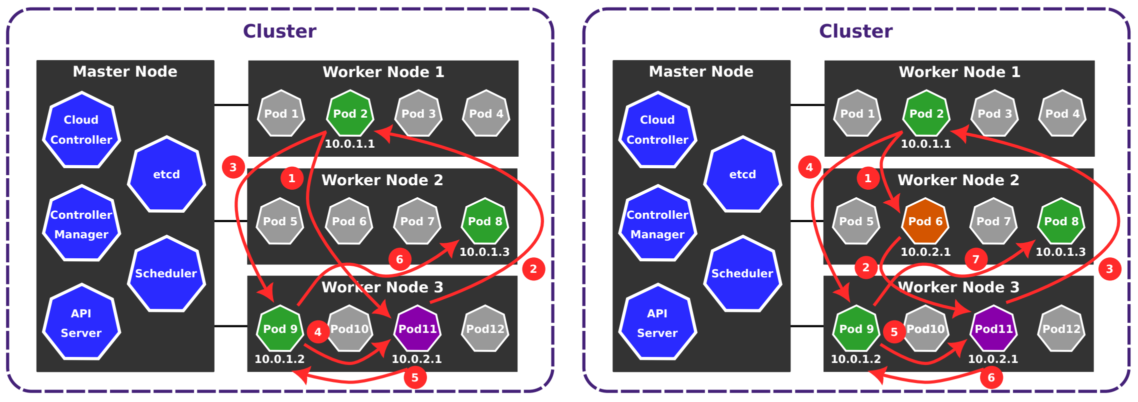
Scenario 3 (Eavesdropping attack)**.** *When a microservice sends its raw data to the EaaS components to encrypt it, during this transmission, there is also the possibility of the adversary eavesdropping it. In other words, like the man-in-the-middle attack, the adversary may be located between the microservices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

Another sample Kubernetes environment is shown in Figure 4, where one of the pods hosts the EaaS components. In the environment without a malicious pod (Figure 4(a)), the second pod sends its data to the EaaS component, with the IP address of 10.0.2.1 (i.e., the eleventh pod), and receives the encrypted form. Then, the encrypted data is sent to the second microservice located at the ninth pod. Then, the ninth pod sends the encrypted data to the EaaS pod and receives the decrypted data. The procedure is completed similarly to the example mentioned in Figure 2. It is worth noting that the communication between the second and the third microservices is assumed to be secure in this example. On the other hand, when an eavesdropping attack is performed, the sixth pod is located between the second pod and the eleventh pod to read their data, and the data is revealed.

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 make the eavesdropping attack on the channel between the microservices and EaaS components more costly to the adversary, we have considered transmitting fake data from the microservices to the EaaS components and vice versa. In this solution, when a microservice wants to encrypt its data, multiple requests are sent to the EaaS components, among which only a single request is real and the others are fake. The fake messages are sent from the EaaS components to the microservices for the decryption process. Therefore, if the adversary tries to eavesdrop on the related link, all the requests must be investigated and processed, and the adversary is not sure which one is real. A secret token in the requests specifies which of them are real and which are fake.*

Now, putting all three proposed solutions together, we can suggest a general security solution for protecting service mesh. Based on these solutions, we need a third-party EaaS framework that performs the cryptography processes but is split into microservices. In other words, EaaS microservices help other microservices to be secure. The egress traffic of a non-EaaS microservice is first sent to the EaaS components. Then, the encrypted data is forwarded to the next non-EaaS microservice in the service chain. The ingress traffic to a non-EaaS microservice is also sent to the EaaS components, and the decrypted data is received. We define the structure of the messages transferred between the non-EaaS and EaaS microservices as Figure 5. Two different messages are defined. The first type is for requesting an encryption process (Fig-



(a) No attack (b) Under attack

Fig. 4. A sample Kubernetes environment that an eavesdropping attack may target

s bits	r bits	n bits	one bit	r bits	t bits	variable number of bits
Subscriber	Requester	Sequence Number	Type	Pair	Token	Input Data

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

s bits	r bits	n bits	one bit	r bits	variable number of bits
Subscriber	Requester	Sequence Number	Type	Pair	Input Data

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

Fig. 5. The structure of the defined messages in the proposed solution

ure 5(a)), and the second one is for decryption (Figure 5(b)). The first s bits of both messages indicate the subscriber's identifier. Subscriber is the service provider that allocates its microservices on Kubernetes and wants to use cryptographic services for them. The value of s is fixed for a Kubernetes environment. However, this value depends on the maximum number of subscribers a Kubernetes environment wants to support. The next r bits of both messages specify the requester identifier. The non-EaaS microservices are called requesters as they request cryptographic services. Similar to the value of s, the value of r is fixed for a Kubernetes environment, but depends on the maximum number of microservices a subscriber can have. The next n bits of both messages is the sequence number. When a microservice sends a request, this number is incremented. We can say that the combination of subscriber's identifier, requester's identifier, and sequence number can be the request identifier. The sequence number field consists of n bits, and its length is fixed. A single bit is also dedicated to the request type. If this bit is zero, the requester wants to encrypt its data. Otherwise, it specifies a decryption request. The next p bits of both messages are called the pair field and indicate the other end of the communication. In other words, the pair field indicates one of the non-EaaS microservices, which is the source or the destination of the connection this message has passed through. As the pair and the requester fields point to the same component types, they are both r bits. In the encryption requests, the requester has to specify the token. Hence, in Figure 5(a) 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.

Now, we give an example of how the messages are constructed based on the defined structure. Assume that the service provider owning the sample service in Figure 4(a) 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 II are generated.

III. Proposed Framework

Our EaaS framework comprises four main components: Coordinator, Registrar, Key Manager, and Cryptor. The coordinator is responsible for handling the requests and coordinating the other three components. The registrar is responsible for assigning cryptography services to service providers. The service providers register by specifying their requirements and the necessary deception level. The registration workflow is shown in Figure 7. A service provider, who is a subscriber in our framework, sends its identifier (S_ID) and required deception factor (D_Factor) to the registrar component to register for receiving cryptography services (Step 1). The deception factor is the number of bits required to present the token. The registrar generates a token (T) based on the provided deception factor and sends it together with the subscriber's identifier to

$(54)_{10}$	$(1)_{10}$	$(9)_{10}$	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. 6. The sample messages transferred between a non-EaaS microservice and an EaaS microservice

Algorithm 1 The registrar's procedure

```
Require: $coordinators$ (the list of authorized coordinators)
1:  for each arrived packet as $p$ do
2:      $src$, $payload$ ← extract $p$ fields
3:      if $src \in coordinators$ then
4:          $subs$, $coord$, $token$ ← extract $payload$ fields
5:          send $(coord, token)$ to $subs$
6:      else
7:          $subs$, $d$ ← extract $payload$ fields
8:          $t$ ← generate random binary number with $d$ bits
9:          $dst$ ← find a coordinator for subscription
10:         send $(subs, t)$ to $dst$
```

Algorithm 2 The key manager's procedure

```
Require: $coordinators$ (the list of authorized coordinators)
1:  $keys$ ← initiate a database
2:  for each arrived packet as $p$ do
3:      $src$, $payload$ ← extract $p$ fields
4:      if $src \in coordinators$ then
5:          $id$, $pair$ ← extract $payload$ fields
6:          $subs$, $req$, $i$ ← extract $id$ fields
7:          if $(subs, req, pair) \in keys$ then
8:              $k$ ← retrieve $keys[subs, req, pair]$
9:          else if $(subs, pair, req) \in keys$ then
10:             $k$ ← retrieve $keys[subs, pair, req]$
11:         else
12:             $k$ ← generate a new key
13:             store $(subs, req, pair, k)$ in $keys$
14:         $out$ ← $(id, k)$
15:     else
16:         $out$ ← $(error)$
17:     send $out$ to $src$
```

a coordinator (Step 2). The coordinator stores the subscriber's identifier and the token assigned to it and then checks the available resources of other coordinators to find one to handle the subscriber's requests. The identifier of the selected coordinator (C_ID) is then sent back to the registrar, the subscriber's identifier, and the assigned token (Step 3). Once the registrar receives this message, the coordinator's identifier and the token are forwarded to the related subscriber (Step 4). The subscriber can now broadcast this information to its microservices, or in other words, requesters (Step 5). The detailed algorithm of the registrar procedure is shown in Algorithm 1. When a registrar receives a packet, first, it is split into two parts. This packet's first s bits indicate the source component identifier (Line 2). Suppose the source is among the valid coordinators' identifiers. In that case, it means that a coordinator has sent it to approve a registration process, equivalent to the third step of the registration workflow. Hence, the related subscriber, the coordinator, and the token assigned to it are extracted from the payload (Line 4), and the coordinator identifier and the token are sent back to the subscriber (Line 5). Otherwise, the packet is considered to be sent from one of the subscribers, equivalent to the first step of the registration workflow. The subscriber identifier and the deception factor are extracted from the packet payload (Line 7), the token is generated (Line 8), a free coordinator is found (Line 9), and the request is forwarded to it (Line 10).

The key manager generates cryptographic keys and manages which keys are assigned to which pair of microservices. The details of this component are presented in Algorithm 2. A key manager stores a database containing the keys generated for each pair of microservices to communicate with each other (Line 1). When a key manager receives a packet, it is split into two parts. Its first s bits indicate the source component (Line 3). If the source component is one of the valid coordinators (Line 4), a coordinator has requested the key used for two microservices. Hence, the packet's payload is extracted, and the related identifiers are obtained (Line 5 and 6). If the pair of microservices is found in the key manager's database, they are retrieved (Line 7 to Line 10). Otherwise, a new key is generated (Line 12) and stored in the database (Line 13). Finally, the request identifier and the key are considered to be sent back (Line 14). If the packet is received from an invalid coordinator (Line 15), an error message is considered the response (Line 16). Now, the key manager sends the response message to the source component (Line 17).

The cryptor simply performs the encryption and decryption processes. We can see its procedure in Algorithm 3. When

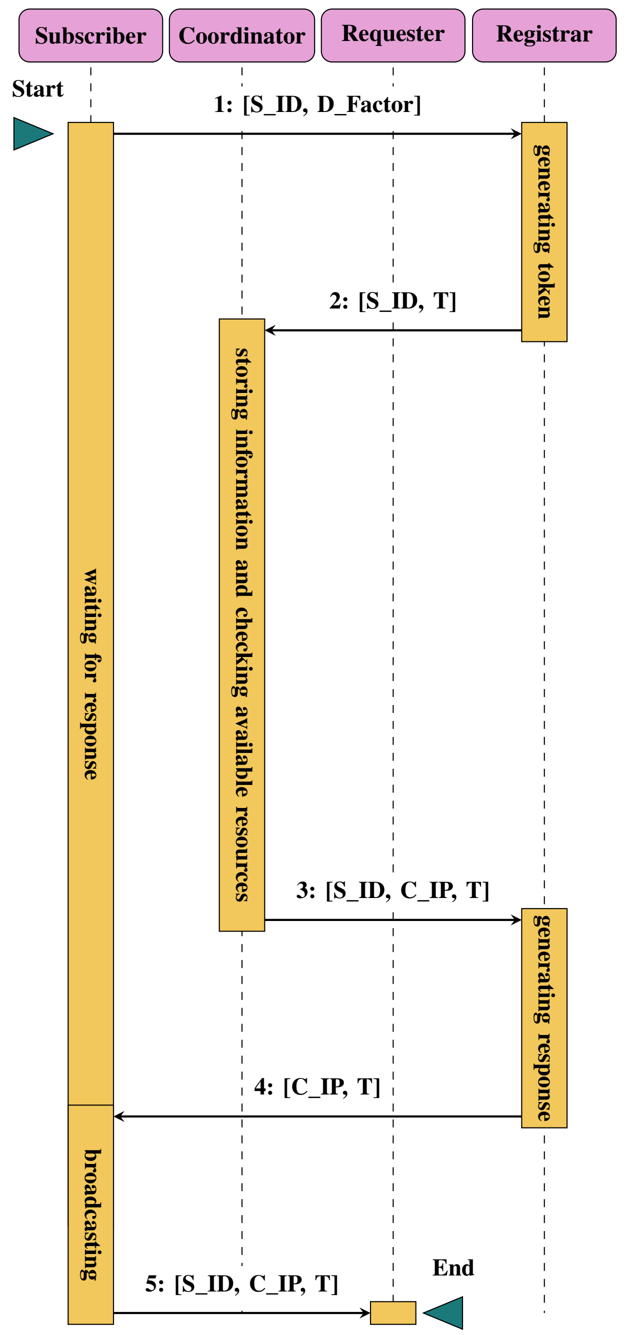


Fig. 7. The workflow of registration in the proposed EaaS framework

Algorithm 3 The cryptor's procedure

```
Require: $coordinators$ (the list of authorized coordinators)
1:  for each arrived packet as $p$ do
2:      $src$, $payload$ ← extract $p$ fields
3:      if $src \in coordinators$ then
4:          $id$, $type$, $key$, $inp$ ← extract $payload$ fields
5:          if $type = 0$ then
6:              $out$ ← encrypt $inp$ with $key$
7:          else
8:              $out$ ← decrypt $inp$ with $key$
9:          $out \leftarrow (id, out)$
10:     else
11:         $out \leftarrow (error)$
12:     send $out$ to $src$
```

a cryptor receives a packet, its first s bits are considered the source component (Line 2). If the source component is one of the valid coordinators (Line 3), the request identifier, its type, the related key, and the input data are extracted from the packet payload (Line 4). If the request type is zero (Line 5), it means that an encryption service is requested. Therefore, the input data is encrypted using the provided key (Line 6). Otherwise, it is decrypted (Line 7 and Line 8). Finally, the generated data and the request identifier are considered to be sent back (Line 9). An error message must be sent if the source component is not a valid coordinator (Line 10 and 11). In the last step, the generated response is sent back to the source component (Line 12).

The workflow of the requesting process for a cryptographic service is shown in Figure 8. In the first step, a microservice, which is equivalent to the requester in our framework, sends an identifier (ID), containing its subscriber's identifier, its identifier, and the sequence number of its requests, together with its request type (Type), the pair requester to communicate with (Pair), and its input data (Inp) to the coordinator that was assigned to its subscriber (Step 1). If the request is of encryption type, the token is also sent. The coordinator checks the provided token. The request is ignored if it does not equal the token assigned to that subscriber. Otherwise, the coordinator sends the identifier and the pair requester to a key manager (Step 2). The key manager generates a key (Key) for that request or, if it is previously generated, retrieves it and sends it to the coordinator (Step 3). Once the coordinator receives the key, the related request's identifier, its type, its data, and that key is sent to a cryptor (Step 4). The cryptor encrypts/decrypts the input and sends the output (Out) to the coordinator (Step 5). This message is sent to the coordinator and forwarded to the related requester (Step 6). It is worth noting that if the request is of decryption type, the token is also sent to the requester. Based on this workflow, the coordinator works as shown in Algorithm 4. The coordinator maintains two databases: one for subscriber information (Line 1) and another for tracking requests (Line 2). Upon receiving a packet, it extracts the source and payload (Line 4). Based on the source, different actions are taken:

If the source is a *registrar* (Line 5), the subscriber ID and token are extracted (Line 6) and stored (Line 7). A coordinator is assigned and sent back (Lines 8–9).

If the source is a *key manager* (Line 10), the request ID and key are extracted (Line 11), and the full request info (type, pair, input) is retrieved (Line 12). A cryptor is selected, and the data is forwarded (Lines 13–14).

If the source is a *cryptor* (Line 15), it sends the request ID and output (Line 16). The request ID is unpacked into subscriber, requester, and sequence (Line 17), and the full request is retrieved (Line 18). If it's encryption (Line 19), the output is sent to the requester (Line 20). Otherwise, the coordinator retrieves the real token (Line 22), computes its length d (Line 23), and sends 2^d messages (Line 24): the real output for the true token (Line 26), and random sequences

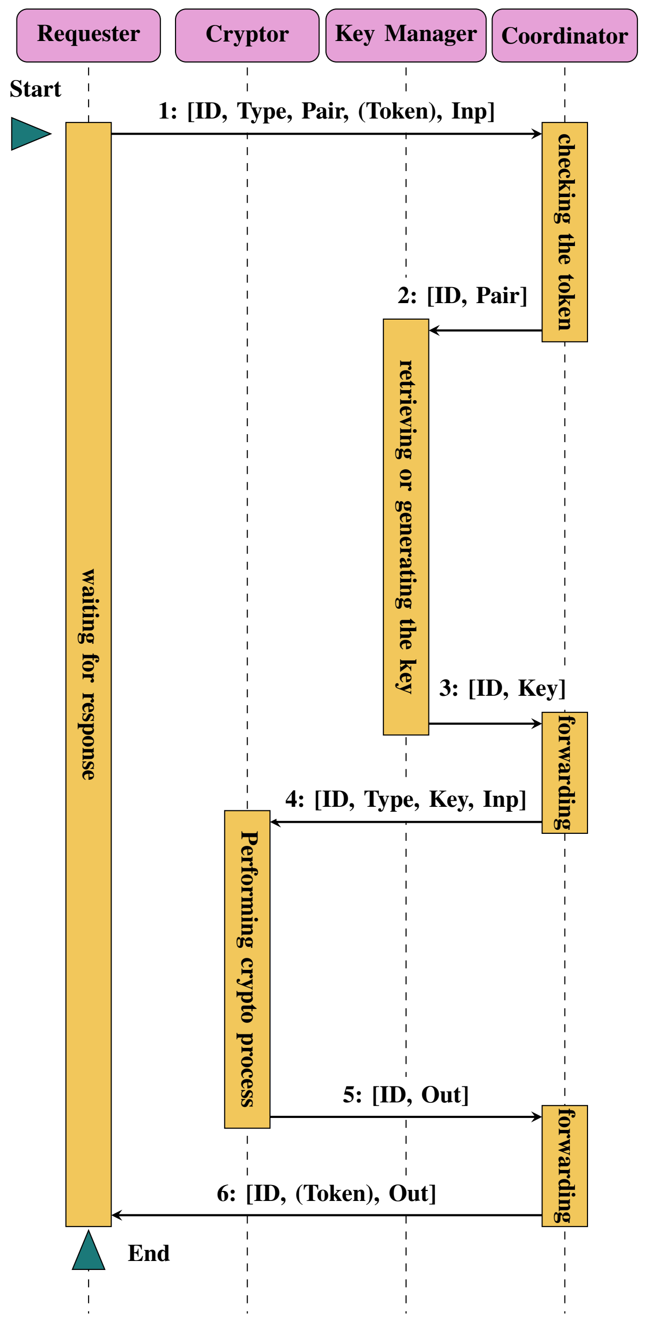


Fig. 8. The workflow of requesting in the proposed EaaS framework

otherwise (Lines 27–29).

If the source is a *subscriber* (Line 30), the request type is determined (Line 31). For encryption (Line 32), the token is validated: the real token and its length d are retrieved (Lines 33–34), and the first d bits of the payload are compared (Line 35). If invalid, the request is ignored (Lines 36–38). Valid requests or decryption requests proceed: the request ID is created

IV. Security and Overhead Analysis

To implement our proposed solution in Kubernetes, we have written a Python proxy that uses the *scapy* module to sniff the packets and then forward them to the appropriate component.

Algorithm 4 Coordinator Procedure (Simplified)

```
Require: Lists: $registrars$, $managers$, $cryptors$
1: Initialize $subscribers$, $requests$
2: for each packet $p$ do
3:     Extract $src$, $payload$ from $p$
4:     if $src \in registrars$ then
5:         Store subscriber info in $subscribers$
6:         Assign coordinator and notify $src$
7:     else if $src \in managers$ then
8:         Retrieve request from $requests$
9:         Forward to appropriate cryptor
10:    else if $src \in cryptors$ then
11:        Retrieve request and subscriber info
12:        if type $= 0$ then
13:            Send output to requester
14:        else
15:            Generate $2^d$ outputs, only one valid
16:            Send all to requester
17:    else if $src \in subscribers$ then
18:        if type $= 0$ then
19:            Verify token; drop if invalid
20:        Store request and notify key manager
21:    else
22:        Send error to $src$
```

When the microservices are located on the Kubernetes pods, this proxy is also executed on the related pod.

To compare the security level of Kubernetes environments that utilize our proposed framework and those without any security approach, we have implemented a distributed database service, where the microservices transfer the password for using the database between each other. Then, we compromised one pods and tried to attack them. The adversary has a limited time to find the password and read the databases. If the adversary reads at least one of the databases, this attempt is considered as a successful attack.

We have considered different cryptography algorithms for the proposed framework and compared the results. The implemented algorithms are symmetric: DES, AES, and Blowfish. Moreover, three scenarios for the number of replicas of the EaaS components are considered. In scenario α, only a single replica of each component exists. In scenario β, there is a single coordinator but two key managers and two cryptors. In scenario λ there are five replicas of each component. In each scenario, if available components are not found for a request, the request is rejected and the raw data is transmitted to the next microservice. To have baselines for our evaluation, we have also implemented the No Encryption and Local Encryption scenarios. In the latter scenario, if a microservice wants to send data to the next microservice in a chain and runs out of resources, the data is sent in its raw form.

The adversary's success rate when different values of deception factors are considered is shown in Figure 9. The chart shows a clear downward trend: as the deception factor increases, the success rate of attacks decreases due to the adversary needing to process more fake requests. On average, our framework reduces the adversary's success rate by approx-

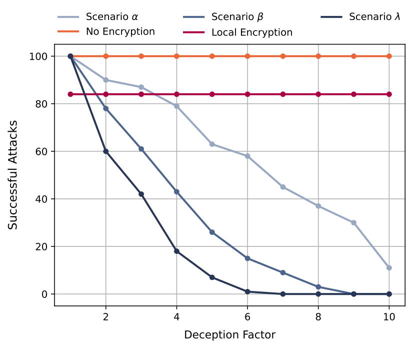


Fig. 9. Comparing the adversary's success in different scenarios with different deception factors

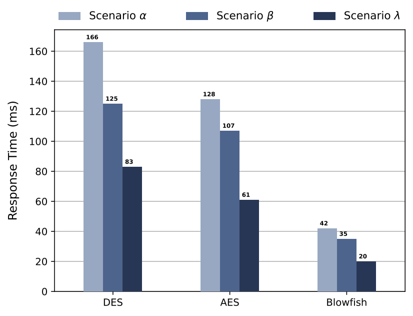


Fig. 10. Comparing the non-EaaS services' response time in different scenarios with different cryptographic algorithms

imately 61% and 45% compared to *No Encryption* and *Local Encryption*, respectively. These baselines remain flat, as they are unaffected by deception. *Local Encryption* is below 100% since some data is encrypted. Notably, scenario λ outperforms α and β by about 24% due to better load distribution and broader encryption coverage.

To evaluate the overhead of our proposed EaaS framework, we have compared the non-EaaS services' response time in Figure 10. 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 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 scenario α and β. This is because the load is distributed among more replicas in scenario λ.

V. Conclusion

Service mesh is a network layer that enables communication between microservices, but adversaries can target it to disrupt functionality. This paper discusses three types of attacks on this layer: man-in-the-middle, (distributed) denial-of-service, and eavesdropping attacks. We propose a solution involving a third-party component, the EaaS component, which provides cryptographic services for non-EaaS microservices. To complicate eavesdropping, we suggest sending fake requests between the EaaS and non-EaaS microservices. A framework based on this solution can be implemented in Kubernetes, along with detailed algorithms. Our results indicate that this approach can reduce adversary success rates by at least 45% compared to self-managed cryptographic processes. We aim to enhance these results by integrating multiple cryptography algorithms and employing a machine learning model to select the optimal algorithm for 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] 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.

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

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

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

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

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

[9] A. Javadpour, F. Ja'Fari, T. Taleb, C. Benzaïd, 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.

[10] 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.

[11] 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*, p. 103792, 2024.

[12] 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.