Detection of Anomalous e2e Encrypted Function Invocation in FaaS using Zero-Knowledge Proofs

Davide Andreotti and Giacomo Verticale
Dept. of Electronics, Information, and Bioengineering — Politecnico di Milano – Italy
Email: davide.andreotti@polimi.it, giacomo.verticale@polimi.it

***Abstract*—Function-as-a-Service providers manage security devices that are shared among multiple tenants. It is undesirable to give them access to cleartext HTTP requests to perform tasks such as traffic inspection. The recent Zero-Knowledge Middlebox (ZKMB) can be used to enforce network policies on TLS traffic without revealing any information on the content to the policy verifier. In this paper, we describe a ZKMB implementation and a policy designed to check whether the HTTPS function invocations by the clients follow a legitimate pattern. We also present and compare two strategies to distribute allowed patterns, introducing a Moving-Target Defense approach for the function URI randomization, which shows a good tradeoff between detection effectiveness and confidentiality. Performance assessment in our prototype implementation shows that the ZK algorithms are not yet suitable for real-time execution, but current research interest in this technology is expected to narrow this gap.**

Keywords: Zero-Knowledge Proofs, Function-as-a-Service, Middlebox, Moving Target Defense

I. Introduction

Function-as-a-Service (FaaS) is a popular way to execute modular applications that exchange data with clients running in field devices and perform computation in the cloud. In FaaS, clients invoke functions using HTTPS with each function corresponding to a different URI. Multiple clients send their requests to the same ingress controller, which, in turn, launches a container to execute the proper function, collects the answer and routes it to the client.

A threat scenario consists of a compromised device trying to invoke functions for which it has no legitimate reason. In such a scenario, the device might be trying to exploit an unknown vulnerability or to discover network resources in preparation for further attacks. Prevention of these threats relies on the ability of the infrastructure to monitor and block anomalous behavior.

Traditionally, middleboxes play a central role in this purpose, particularly when performing Deep Packet Inspection (DPI) to detect malicious traffic. End-to-end encryption is a major obstacle to the effectiveness of middleboxes operating at the network layer. Consequently, any attempt to detect attacks at the infrastructure layer requires application gateways that break end-to-end encryption, introducing a lack of confidentiality in multi-tenant clouds.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

To overcome this drawback, the recent Zero-Knowledge Middlebox (ZKMB) architecture [1] proposes a network middlebox that employs Zero-Knowledge (ZK) proofs to enforce some policies on the traffic. In addition to its TLS-encrypted packets, the client sends the middlebox a cryptographic proof that the plaintext is policy-compliant. The middlebox can verify compliance without the client disclosing any further information about the plaintext. This work extends the ZKMB architecture in the following directions.

- We implement a working environment to deploy the box at the ingress of the cloud provider verifying that the client is only invoking a function matching an allowlist. In particular we implement session tracking for TLS, a protocol to transfer data between middlebox and clients, and modifications to the existing ZK tools to make some middlebox operations independent of the client.
- We develop a ZK-policy that can be used to prove that the invoked function is allowed and the HTTPS request header carries an authentication token.
- We design and compare two strategies to communicate the list of allowed functions from the server to the client and the middlebox. The URIs in the list can be randomized to prevent resource enumeration through a Moving-Target Defense approach.

In a typical scenario, each client sends a sequence of function invocations to the FaaS ingress controller, according to some application logic. The middlebox detects any request outside of this logic, and raises an alert.

The rest of the paper is structured as follows. Section II discusses the related work. Section III reviews some background concepts. Section IV presents our reference scenario, while Section V presents our proposed detection mechanism. Section VI discusses some performance results. Concluding remarks are left for the final section.

II. Related Work

Existing FaaS security solutions mainly focus on system architecture and application structure. Valve [2] analyzes information flow within internal function calls, without requiring developer intervention. Operations in each container are labeled, to centrally detect known attacks, but the system lacks defense against unknown strategies.

Encrypted traffic inspection is normally achieved by obtaining access to the plaintext, for example through key export from one end [3]. An issue may arise if the middlebox is

not trusted to handle sensitive data. Users' privacy can be guaranteed through many different techniques, among which the use of a trusted execution environment, such as mbTLS [4] or modifications to the TLS protocol, as in the case of multi-context TLS [5], which allows partial access to requests content by the middlebox. Both are limited by the compatibility of all parties to the employed technologies and protocol modifications.

The third option is to use a Zero-Knowledge Middlebox (ZKMB) [1], in a system where clients can prove to the middlebox that their communication is policy-compliant, without revealing secret information. The authors present a general architecture, which they further improve in [6]. However, their DNS blocking application pays for the lack of performance of the Zero-Knowledge algorithms, which we think can find a better implementation in detection tools. We followed the same approach to preserve privacy without any specific protocol or hardware compatibility, introducing a specific policy and doing some implementation adjustments that make the ZKMB architecture applicable to the FaaS scenario.

III. Background on Zero-Knowledge Proof

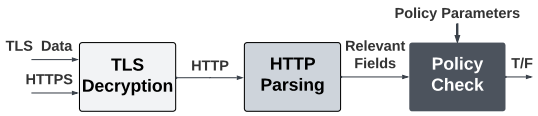

Fig. 1. Scheme of the ZKMB circuit pipeline, with inputs and outputs.

A zk-SNARK (Zero-Knowledge Succinct Non-Interactive ARguments of Knowledge) is a protocol that allows a **prover** to convince a **verifier** of the validity of a given computation without the prover disclosing some secret input to the verifier *(for example, TLS keys)*. [7] The *Zero-Knowledge Proof* (ZKP) is a short piece of data, that reveals nothing about the underlying secret inputs. The protocol consists of three phases: the proving and verification keys are generated in the **setup** phase by the verifier, in the **proving** phase the prover computes the ZKP, and in the **verification** phase the verifier establishes the validity of the ZKP against the verification key and other public inputs. Grubbs *et al.* [1] propose using ZKPs for firewalling DNS queries over TLS with a network middlebox acting as the verifier. The client, acting as the prover, shows that the domain in the request is allowed, by proving a modular pipeline in Fig. 1. The policy is written using a high-level language and compiled into an arithmetic circuit using xJsnark [8], while the secret inputs include TLS keys and public inputs include ciphertexts and other policy metadata. The backend operations are implemented in Libsnark [9].

IV. System Model

A. Parties and Security Assumptions

As depicted in Figure 2, the interaction involves three parties: one or more clients, the FaaS server and the ZKMB.

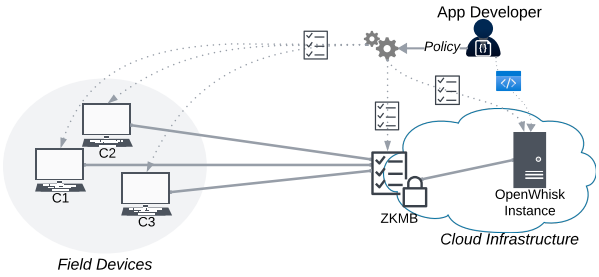

Fig. 2. Network topology and interactions between clients, middlebox, FaaS. The application developer distributes new policies and function code.

Clients: Field devices invoke functions on the server via HTTPS requests, sending collected data or performing critical operations. In addition, the clients execute the Zero-Knowledge toolchain for proof generation and implement a protocol to send the proof to the middlebox. We assume that each function call starts a new TLS session and the client applications do not use hardcoded URIs when invoking server functions but obtain the available URIs at runtime from the server. A compromised client can run a modified software stack to invoke malicious sequences on the server, for example by attempting to use server exploits or exfiltrate data. Still, the attacker cannot break Zero-Knowledge algorithms. While the architecture presented here is general, we will consider TLS 1.3 with the `TLS_AES_128_GCM_SHA256` cipher suite.

ZKMB: It is deployed at the ingress of the cloud infrastructure and managed by the cloud provider. All TLS handshakes and requests directed to the customer FaaS instance are intercepted and logged, but not decrypted. This device can run the Zero-Knowledge toolchain for proof verification and must execute a protocol to receive proofs and distribute policy metadata and code. It also has a connection to the FaaS server to obtain updated policies based on the client interaction. The ZKMB is honest-but-curious, it correctly executes the protocols with the clients and the server, but can try to guess the content of clients' invocations, in particular, functions and other HTTP fields.

FaaS Server: The FaaS server is responsible for receiving function invocations from the client and answering them. It also randomizes the URIs and sends them to the client and the ZKMB. It is honest. The FaaS controller implements logging of the invocation history and session blocking before scheduling a container.

B. FaaS Invocation Patterns

In the cloud infrastructure, a client interacts with the FaaS server multiple times, invoking multiple functions over time in accordance with some pattern that depends on the application logic. For example, a client may need to retrieve an authorization token before being allowed to send collected data. We assume that is it possible to define a set of invocation sequences U that is consistent with the application logic, e.g $U = \{(u_1, u_2, u_3), (u_4, u_2), (u_1, u_5, u_6)..., (u_i, u_j, u_k, u_l)\}$ where u_n is a certain function available in the FaaS system.

Hence, based on the history of the interaction, at any given point there will be a limited list of functions that can be called next. In the above example, if u_1 is called, then only u_2 or u_5 are allowed. Any other function invocation is anomalous and potentially malicious. We assume that the FaaS server knows, at any time, which function invocations are legitimate and can send this list to the client.

C. Security Definition

With respect to a compromised client, the attacker's goal is to successfully invoke a sequence of URIs having a Hamming distance of at least K from the nearest sequence in U without being detected. The system is secure if the client-side attacker has a probability below a given threshold ε to succeed.

From a curious middlebox point of view, we can define two security models. We say that the system is **weakly secure** if the middlebox has non-negligible probability of guessing the invocation sequence, but has negligible probability of learning any other information, such as the invocation parameters. We define the system as **strongly secure** if the middlebox has negligible probability of learning any information about the exchange, including the invoked functions.

V. Detection Mechanism

Our contribution spreads in two directions: a new policy, and two policy distribution strategies.

A. Zero-Knowledge Policy

Similarly to previous work, we employ the xJsnark framework to create high-level parsing instructions and translate them into an arithmetic circuit. Our backend runs the *Groth16* [10] proof system, which uses the circuit and inputs to run the three protocol phases in Sec. III. The arithmetic circuit formally represents the computation in the ZKP system as a composition of interconnected gates performing arithmetic operations. Policy metadata (or inputs) are translated in a format compatible with the circuit structure. Each policy consists of code and metadata. The *policy code* instructs on how or what to check in a packet and can be used for multiple clients. An example of code are the rules to locate the URI in the encrypted packet, which can be used for multiple users. *Policy metadata* differ from one user to the other and specify some information necessary to verify the validity of the policy, e.g. a string containing the URI of the allowed function, which depends on the user rights in the system.

The policy code initially derives the traffic key from the TLS handshake transcript, which is public, and the handshake secret computed at the client side. In turn, the key is used to decrypt the request in the following non-handshake packet. This request contains the URI corresponding to a function invocation, and other HTTPS headers. It is worth noting that the traffic decryption circuit is not actually executed by the middlebox, which does not know the handshake secret; instead, the middlebox verifies that the client correctly executed the circuit on the encrypted packets.

A list of allowed function URIs (*allowlist*), is distributed by the server to each client and to the middlebox. The list is organized into a Merkle Tree structure.

When a function is invoked, the client must first generate the Merkle tree *membership proof* for the given function URI. This preprocessing step, which outputs the URI membership proof, is not included in the policy code. A leaf of the tree containing the specific URI and the membership proof are client secrets, while the tree root is public.

After the membership proof, the ZKP can be produced. The circuit logic implements the verification of the validity of the membership proof: the hashes generated climbing from the bottom to the top of the tree must generate a root that is equal to the publicly available one. The leaf string must also match the URI in the HTTPS request.

Additionally, a specific header must be included in the HTTP request, containing a pre-distributed FaaS authorization token. This becomes useful, to verify that client-server authentication is successful from the middlebox, to prevent identity spoofing by an attacker.

Header check is implemented by searching for a `CRLF` sequence until the desired line is reached. The value in the request must match the one known to the middlebox, passed as policy metadata.

B. Allowlist Distribution Strategies

Allowlists are crucial for both the middlebox and the clients, to let them know which policy is enforced. We present two methods to distribute the allowlist.

***Cleartext Real-time List** (CRL):* One approach is to generate a new allowlist after each function invocation based on the current client-server interaction. A sequence diagram of this method is in Figure 3: the server distributes the first allowlist to both the client and the middlebox. If the client calls an allowed function and generates a valid proof, the middlebox correctly verifies that, and the server replies distributing a new allowlist for the next interaction to both client and middlebox. If the client invokes a function that is not part of the list, the proof generated is invalid.

The new list is included in the FaaS HTTPS response to the client and it is also distributed to the middlebox, resulting in the client not being able to generate the proof for any malicious function invocation sequence. This scheme achieves security

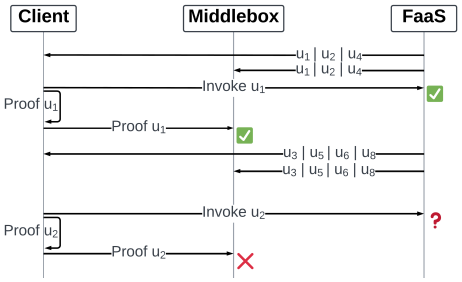

Fig. 3. Policy distribution and interactions in the *CRL* method

against malicious clients with very high probability, at the cost of higher communication overhead between the server and the other parties. Additionally, an honest-but-curious middlebox can infer the invoked functions based on the sequence of allowlist updates.

Moving Randomized Full List *(*RFL*):* Alternatively, the allowlist can contain the entire list of URIs but randomized. This allows hiding functions that are not yet allowed to be executed, adding client-side security. Having the full list, hence not sending real-time updates to the middlebox, also means solving the privacy issue in *CRL*.

A sequence diagram of the standard interaction with this method is in Figure 4: the server distributes the full randomized allowlist to both parties, but the client is also told the relation between each allowed function and random strings. If the client invokes an allowed function, it generates a valid proof, and receives new function-to-random-string definition. Instead, if u_2 is called, its correct location is not known by the attacker, so a random string is picked from the list. A correct proof is produced, but the probability of it being the desired function is $\frac{1}{N}$.

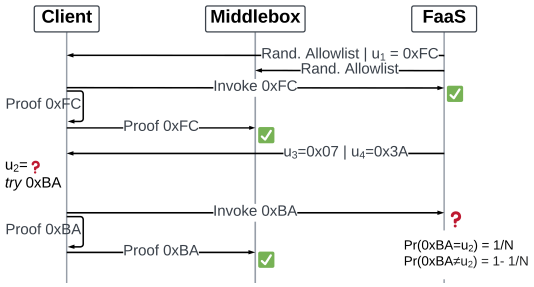

Fig. 4. Policy distribution and interactions in the *RFL* method

The allowlist is refreshed every T_m mutation interval and includes all functions in the application logic in multiple randomizations, each valid only once, to prevent replay attacks. A set of padding strings, acting as decoys, is included to make bruteforce attacks harder. The FaaS controller implements logging of the invocation history and detection of replay attacks or forbidden calls, before starting up a new container.

Server responses contain a structure that associates the newly allowed functions with the respective random strings in the complete allowlist. To complete an attack, a malicious user must find which functions to call in the randomized pool. Decoy strings or inactive randomizations (past or future) trigger a session reset from the FaaS, making the attacker lose track of their discovery process. Relying on the randomization means that the attack success probability depends on the allowlist size and the length of the attack in terms of number of functions. A deeper analysis is discussed in Section VI.

VI. Performance Evaluation

A. Methodology

Tests are performed on Ubuntu 20.04 on a 6-core/12-thread Ryzen 5600X platform with 16GB of RAM. I/O operations use a PCIe Gen4 interface. Libsnark is set to use multicore and no point compression, leading to measuring the fastest possible execution times. For results in Sec.VI-B the request size is set to 400 Bytes and the allowlist has 32 elements.[1]

B. Zero-Knowledge Algorithm Performance

Resource Consumption*:* It is possible to gather some preliminary performance indicators for *Setup* and *Prove* for the presented policy. The Prover Key is 1.12 GB, which is significant. This is due to the implementation complexity of AES and SHA. However, this cost is amortized over the policy lifetime, still, memory reading costs must be paid for each proof. The maximum memory usage is 6.28 GB and proof generation takes 41.2 s, and the same reasons apply, limiting the applicability of a ZKP system in constrained devices.

Operations Timeline*:* The timeline on Fig. 5 highlights which operations characterize each protocol phase, and their duration. Preprocessing operations, including Circuit Generation (java-to-arithmetic-circuit translation) and Libsnark warmup (circuit parsing into internal representation), must be repeated for every request on both the client and the middlebox, occupying a considerable slice of time. An improvement is obtained by running the preprocessing operations in parallel on both prover and verifier, as soon as a TLS exchange is detected on the channel, before the proof is received. This way, the client only waits about two seconds for a response after sending the proof.

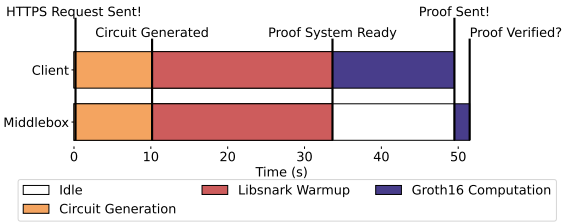

Fig. 5. Operations in a standard ZKP iteration in client and middlebox.

C. Security of Policy Distribution Strategies

We compare the security of the distribution strategies described in Sec. V. Table I resumes the security guarantees that each distribution strategy can provide, in relation to the security definitions described in Sec. IV-C.

The *CRL* approach is by far the most secure from the server point of view: no proof can be generated for a function not part of the latest allowlist. However, this approach brings a higher communication overhead between the middlebox and the server, as well as some privacy issues: a curious middlebox aware of the application logic, can rebuild the U allowlist and deduce the invocation sequence based on the server updates. This is a loss of confidentiality on the invocation sequence. Still, the remaining packet content (e.g. header, body) is protected by encryption, resulting in the respect of only a *weak* middlebox security definition.

[1]Source code is available at https://github.com/bonsai-lab-polimi/zkIDS

TABLE I
SECURITY GUARANTEES OF THE THREE DISTRIBUTION STRATEGIES

Method	Client	Middlebox
Clear Real-time List (CRL)	✓	*Weak*
Randomized Full List (RFL)	✓ (*with prob.* π_K)	*Strong*

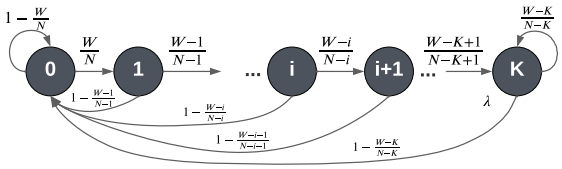

Fig. 6. MTD game for the *RFL* method.

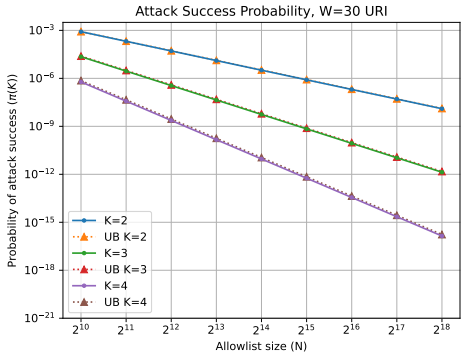

Fig. 7. Numerical and approximation (UB) of the attack success probability for the *RFL* method. The number of functions is $W = 30$.

The *RFL* method complies with the *strong* security definition since the middlebox cannot access any part of the request.

Client-side security is evaluated in terms of attack success probability using a Markov Moving-Target Defense (MTD) model inspired by [11].

We consider the following model parameters: the complete list size, N, the successful attack sequence length, K, and the total number of functions in the application, W.

On the discrete-time Markov Chain in Fig.6, each state defines the number of malicious invocations that have succeeded in the current session. When the attacker wants to make an invocation move, they choose a random URI from the allowlist and invoke the corresponding function. With a probability that depends on N and W, the attacker's guess corresponds to an existing function whose usage is potentially malicious. Otherwise, the attacker's guess corresponds to a decoy URI, which the server will detect and cut the session with the client, resetting all of the progress on the chain.

For a closed-form upper-bound to the stationary distribution, we consider each forward action having probability $\frac{W}{N}$ and each reset action having probability $1 - \frac{W}{N}$. In this case:

$$\pi_i = \left(1 - \frac{W}{N}\right)\left(\frac{W}{N - W}\right)^i \quad 0 \leq i \leq K \tag{1}$$

Fig.7 shows the probability of attack success for different values of K. Both the numerical solution and the upper-bound are depicted. The defender wants to consider as harmful any request that includes one of the W URI that randomly picks in the current interaction, whether or not they are part of an attack. It is worth noting that the approximation is very close to the numerical solution of the chain.

The application developer can choose the allowlist size (both decoy and per-function randomizations) according to the maximum probability threshold ε tolerated by their security definition. It must be noted that the mutation interval, T_m, and the client request rate, T_r, influence the number of per-function randomizations that must be included in the allowlist. Considering the worst case, there must be $\frac{T_m}{T_r}$ replicas of each function. Finally, the parameter K depends on the application robustness.

VII. CONCLUSIONS

We propose an application of the ZKMB to the detection of anomalous function invocation sequences in FaaS scenarios, where end-to-end encryption is used and the middlebox is honest-but-curious. Two strategies for distributing updated allowlists are described, where the tradeoff between performance, attack probability, and communication privacy is studied. Performance is assessed through a proof-of-concept system and a Markov chain model, showing promising balance. While feasible, high computational costs hinder mainstream deployment readiness. Further work will be necessary for optimization and testing on a real-world deployment, while ongoing research suggests near-future improvements on the ZK algorithms side.

REFERENCES

[1] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, "Zero-Knowledge middleboxes," in *31st USENIX Security*, 2022.

[2] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proceedings of The Web Conference 2020*. ACM, 2020.

[3] T. Rutkowski and R. Eriksson, "ETSI White Paper No. 43. Redefining Network Security: The Standardized Middlebox Security Protocol (MSP)," 2021.

[4] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste, "And then there were more: Secure communication for more than two parties," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*. ACM, 2017.

[5] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-context tls (mctls): Enabling secure in-network functionality in tls," *SIGCOMM Comput. Commun. Rev.*, 2015.

[6] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish, "Zombie: Middleboxes that Don't snoop," in *21st USENIX NSDI*, Apr. 2024.

[7] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Commun. ACM*, 2015.

[8] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[9] SCIPR Lab, "libsnark: a c++ library for zksnark proofs," 2017. [Online]. Available: https://github.com/scipr-lab/libsnark

[10] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*. Springer Berlin Heidelberg, 2016.

[11] H. Maleki, S. Valizadeh, W. Koch, A. Bestavros, and M. van Dijk, "Markov modeling of moving target defense games," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*. ACM, 2016.