Secure and Scalable Service-to-Service Interaction in Serverless Microservices Environments

Roshan Mahant[1], Sumit Bhatnagar[2], Vikas Mendhe[3]

[1]LaunchIT Corp, Urbandale, IA USA
Corresponding Author Email: roshanmahant[at]gmail.com

[2]JPMorgan Chase & Co., New Jersey, USA
Email: sumit.bhatnagar[at]outlook.com

[3]LaunchIT Corp, Urbandale, IA USA
Email: Vikas.mendhe[at]gmail.com

Abstract: ***Serverless computing environments consist of standalone microservices that operate independently and scale up autonomously. To enable decentralized communication among these services, peer-to-peer protocols can be employed. This paper introduces Themis, a framework designed for secure service-to-service interaction within serverless environments and underlying service mesh architectures. Themis leverages decentralized identity management to facilitate confidential and authenticated communication between services without relying on a centralized certificate authority. The framework adopts a layered architecture, with a lower layer comprising a core communication protocol pair that offers strong security guarantees without relying on a centralized authority. Building upon this foundation, an upper layer provides actions related to communication and identifier management, such as store, find, and join operations. The paper examines the security properties of Themis's protocol suite and demonstrates its decentralized and flexible communication platform. Evaluation of the Themis prototype, implemented in JavaScript for serverless applications, reveals that these security benefits are accompanied by minimal runtime latency, throughput overheads, and modest startup delays. Themis offers a promising solution for secure and scalable service-to-service interaction in serverless computing environments.***

Keywords: Lambda, microservices architectures, Themis, secure communication, service mesh, decentralized identity management, serverless applications

1. Introduction

The serverless architecture, much like the microservices architecture, is broken down into a number of fundamental components on its own. In serverless computing, functionalities are broken down into more granular components, but in microservices, related capabilities are grouped together into a single service. Developers are responsible for writing their own unique code, which is then executed as independent and self-contained functions executing within stateless computing services. Microservice is a decentralized design pattern that involves the separation of an application into a number of separate functions, also known as services, which are able to collaborate and interact with one another using application programming interfaces (APIs). Every single microservice is equipped with its very own database, libraries, and templates, and it is also tested independently from their counterparts. There is a common comparison made between it and monolithic architecture. All of the features are unified in the latter, as they are tightly related to one another and operate as a single function.[1] Microservices are characterized by the fact that each component is more or less autonomous, and it is not necessary to execute a complete application in order to access single features. Until the mail event is triggered at midnight, there is no server operating to service the mail activity in serverless computing. This is because there is no server running. Following the execution of the code, the server is subsequently decommissioned after it has been allocated. In order to complete tasks that are typically handled by servers, serverless applications frequently make considerable use of services provided by third parties. [2-3] It is possible that these services are either single services that seek to give a turnkey set of capabilities, such as Parse or Firebase, or they could be rich ecosystems of services that are able to communicate with one another, such as Amazon AWS and Azure. Both infrastructure and higher-level abstractions, such as federated identification, role and capability management, and search, could be offered by these services. Examples of the former include message queues, databases, and edge caching. By controlling the request-response cycle, a general-purpose web application that is built on a server is able to fulfill one of its key functions. Additionally, controllers on the server side are responsible for processing input, invoking the relevant application behavior, and constructing dynamic responses, generally with the assistance of a templating engine. The client-side control flow and dynamic content creation take the role of the server-side controllers in a serverless application, which is characterized by the utilization of third-party services to weave together the program's behavior. Utilizing API calls and client-side user interface frameworks to generate dynamic content, rich JavaScript apps, mobile applications, and increasingly, TV or embedded Internet of Things applications, are responsible for coordinating the interaction between the various services [4].

Work that takes place between the controller and the infrastructure, also known as the business logic, is the most important component of a web application that is hosted on a server. For the duration of the application's existence, a server with a lengthy lifespan will host the code that implements this logic and carry out the necessary processing. A lifecycle that is significantly shorter and more comparable to the timing of a single HTTP request/response cycle is possessed by custom code components in serverless apps. Whenever a request is received, the code becomes active, processes the request, and then goes into a dormant state as soon as the activity level decreases sufficiently. A managed environment, such as Amazon Lambda, Azure Function, or Google Cloud Functions, is typically where this code is stored. This environment is responsible for the administration of the code's lifecycle as well as scaling it. The term "Function as a Service" (FaaS) is sometimes used to refer to this particular form of software organization. As a result of the short per-request lifecycle, a per-request price model is also available, which gives certain teams the opportunity to realize significant cost savings. It is possible to fit and separate the business logic in each REST API with its own function. [5] The structures, automation, and optimization are already established. As a consequence, a comprehensive agile infrastructure that is prepared to be deployed in a very short amount of time has been produced.

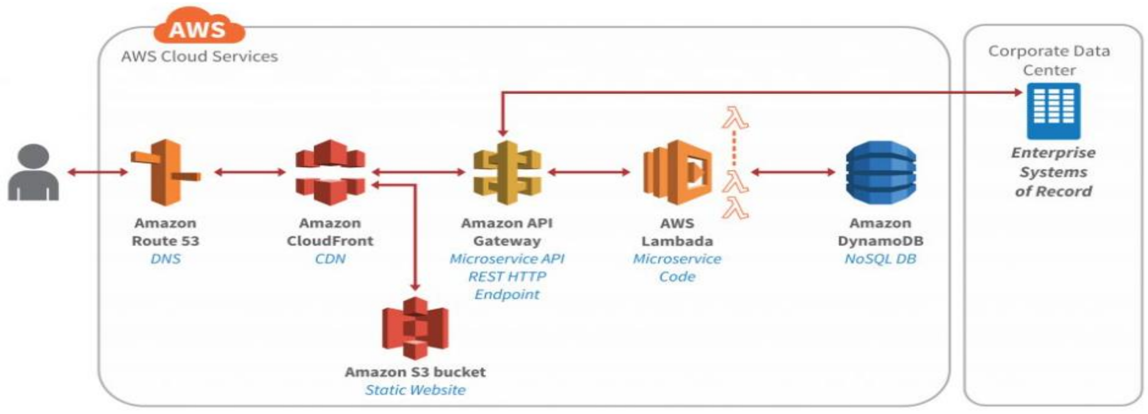

Figure 1: Amazon Tech Stack

An example of a serverless architecture that is based on microservices and is using an Amazon Tech Stack is shown in the image above.

The essential component of the system is Amazon Web Services Lamda, which obtains its routing information from the Amazon API gateway and then executes the capabilities that have been defined for it. Creating serverless microservices with an HTTP front end is one use case for API Gateway + FaaS. This allows for all of the benefits that come with FaaS functionalities, including scaling, management, and other advantages. There is a lot of value in putting technology in front of an end user as soon as possible in order to get early feedback, and the reduced time-to-market that comes with serverless fits right in with this philosophy. [6] The most important benefit, in my opinion, is the reduced feedback loop that is required to create new implementation components for an application. On the basis of our contributions, the following can be summarized:

Security Protocols: By using the self-certifying identities of the nodes involved, they present two innovative methods for secure key agreement and communication. These methods harness the nature of the network. We are able to avoid relying on a centralized source of confidence, which is the case with traditional certificate authorities.

Model and Proofs: These protocols' security assurances are examined in depth in our security study. In particular, we demonstrate that they successfully provide authentication, confidentiality, and message integrity for every single message that is exchanged.

High-level Operations: To ensure that a structured overlay organization is maintained, we identify five essential actions that nodes must carry out. These operations are located, store, join, update, and leave. Through the utilization of these fundamental components, Themis is able to accomplish service discovery and extensibility in a manner that is completely decentralized. Furthermore, they provide further elaboration on the security aspects that are effective against a variety of common types of attacks that are directed at this P2P framework.

Open-source Implementation: Themis is an application library that may be easily added to and removed from existing projects. It is based on QuickJS, a small and embeddable JavaScript engine. About 3,300 lines of JavaScript are required to implement Themis. Themis handles object initialization, communication, and serialization by utilizing a JavaScript implementation of the NaCl Networking and Cryptography package.

Empirical Evaluation: As part of the review of Themis, they target micro benchmarks that scale between one and one thousand nodes and examine its properties across eight serverless applications. Both in terms of runtime performance and in terms of the number of lines of code that are altered, Themis's security gains come at a cost that is negligible to insignificant. [7]

2. Serverless Microservices Architecture: Framework

Combining serverless computing with microservices is a powerful approach to building scalable and flexible applications. Let's briefly look at how serverless microservices work.

After a developer writes the code for a microservices application, its functions are deployed to a serverless computing platform, such as AWS Lambda. Microservices communicate with each other through well-defined APIs and events, and this event-driven communication allows Microservices to operate independently.

On the other hand, the serverless platform automatically scales resources to accommodate changes in workload. Serverless platforms also provide the tools for monitoring and observability. Developers can track performance metrics, diagnose issues, and optimize the system through logging, tracing, and analytics. For example, by leveraging AWS Cloud Watch Logs or CloudWatch metrics, organizations can achieve critical log information, performance metrics, and application insights.

Benefits of Serverless Microservices
In addition to the commonly known benefits of serverless architecture, such as cost efficiency and flexibility, serverless microservices offer several advantages, especially in the case of complex and evolving applications. Let's understand this better with the AWS computing platform services.

Granular scaling for running microservices
AWS' serverless computing service, AWS Lambda, allows you to define and scale individual functions independently within a specific microservice based on event-driven triggers. This ensures efficient resource utilization and cost savings, particularly in scenarios where different microservices have varying demand levels.

Seamless developer experience in building applications
Serverless microservices offer seamless development and deployment experiences facilitated by various tools and practices. For example, AWS offers services such as the Serverless Application Model and the AWS Serverless Application Repository to simplify the process of building, testing, and deploying serverless microservices applications.

Unparalleled flexibility in data handling
When combined with serverless databases, serverless microservices represent a transformational approach to scalable applications. AWS offers serverless databases like Amazon Aurora Serverless and Amazon Document DB with serverless scaling. These database services complement serverless microservices, providing scalable and cost-efficient data storage solutions that automatically adjust to application needs.

Room for experimentation and prototyping
Serverless platforms provide a low-cost environment for experimentation compared to traditional server-based models. As a result, developers can quickly deploy serverless microservices without worrying about upfront infrastructure costs. Consequently, this enables rapid exploration and testing of new ideas or features.

Scalable backend for mobile and IoT
Serverless microservices are well-suited for scalable backends in mobile and Internet of Things (IoT) applications. They can efficiently handle sporadic requests from mobile and IoT devices without constantly maintaining a persistent server infrastructure.

3. Themis Architecture

In this section, the design of Themis is investigated in better detail, and a high-level explanation of its architecture is provided. [8-10] The Themis architecture is structured with a layered approach, consisting of core communication protocols and upper layers for actions related to communication and identifier management. Here is an overview of the Themis architecture:

Core Communication Protocol Layer:
- This layer forms the foundation of the Themis framework, providing a pair of communication protocols that offer strong security guarantees.
- The protocols are designed to facilitate secure service-to-service interaction without relying on a centralized point of authority.
- Security features such as confidentiality, authentication, and integrity are enforced at this layer to ensure the integrity and privacy of communication.

Upper Layer for Actions:
- Building upon the core communication protocols, the upper layer provides a series of actions related to communication and identifier management.
- Actions include functionalities such as storing, finding, and joining services within the network.
- These actions enable decentralized and flexible communication among services while maintaining security and integrity.

Design Goals Themis is a peer-to-peer (P2P) communication system that is designed to be suited for the deployment of a large-scale, multi-cloud, and open service mesh. It does this by achieving the following actions during its development.

Security: Because a service mesh is multi-tenant, allowing multiple applications to share the same machine's resources simultaneously, it is required to offer fine-grained security guarantees. In order to prevent services that are sharing a network from listening in on one another's conversations, the security mechanism that is in place must be able to provide for confidentiality. This means that the data that is being sent between two parties must remain hidden.[11] It is necessary to alleviate the difficulties of setting up the joining procedure, such as connecting with a central authority registry, in order to establish an open service mesh in which other providers can participate. Nevertheless, decreasing the complexity of the joining process makes it possible for both dishonest and trustworthy nodes to coexist on the same network. Therefore, in order to be able to attribute malicious conduct, the service mesh needs to give a high level of accountability for the messages that are being sent. There must be a guarantee of both authentication and integrity in order to establish accountability. In the context of information exchange, authentication refers to the fact that the individuals exchanging data are who they claim to be, whereas integrity refers to the fact that the material that is being sent has not been altered or otherwise falsified.

In order to construct other security primitives, such as authorization, on top of these, they can serve as the fundamental building blocks.

Extensibility: In a serverless application, each service is started and terminated separately, according to the demand made by the program's users. It is necessary for a service mesh to have the ability to scale on its own in order to accommodate the high duplication of services. Service meshes need to be open, which means that instances can be added and deleted in a flexible and quick manner, and they should be hosted on both commercial clouds and client premises where they are physically located. This is necessary in order to prevent difficulties related to vendor lock-in and privacy concerns.

Service Discovery: When it comes to the business logic that they execute, service instances must be able to find one another. By collecting metrics about the internal status of the system, instances can either provide observability functions to the serverless architecture or perform discrete portions of the workflow of the same application. These two choices are both feasible. Since serverless apps like disaster management have their limitations, centralizing the deployment of the discovery technique weakens the serverless infrastructure's resistance to a regional outage. An essential feature of any decentralized service discovery mechanism should be the ability to automatically distribute requests across all available instances of the service, the ability to detect and repair service instances in the event of a failure, and the ability to test new versions of services with a canary release. [12]

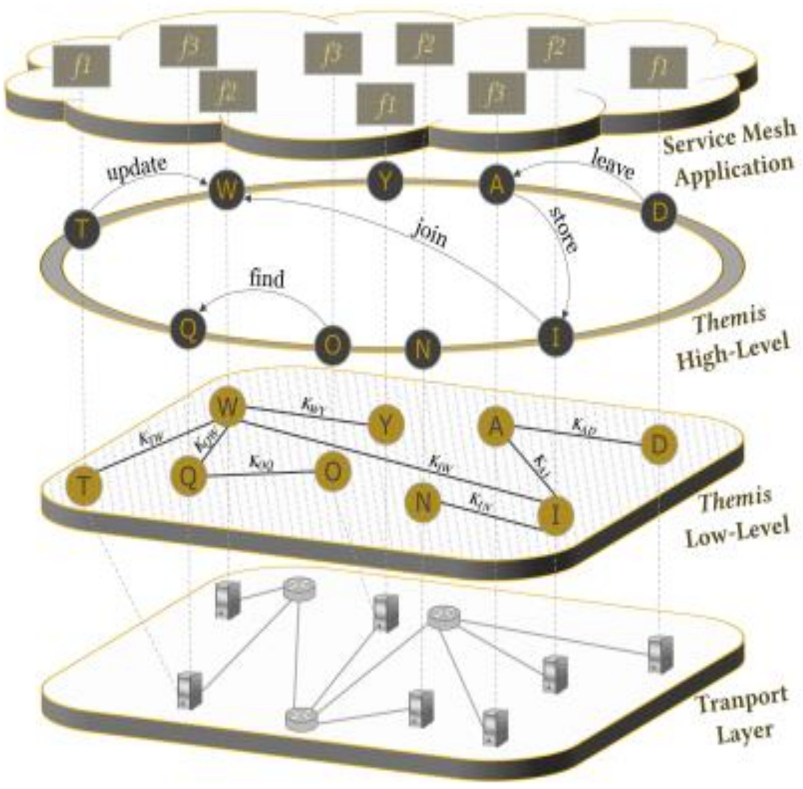

Figure 2: layer architecture

Overview: A two-layer protocol framework is part of Themis's design. An essential communication protocol combination that offers decentralized privacy, security, and authentication is the first, lower layer, in this layer, there are two protocols. A secure channel for communication between nodes can be established using the first protocol, which is an authenticated key agreement protocol. The second one is a protocol that allows the two verified nodes to communicate directly with each other. The key agreement protocol and the mTLS handshake protocol share several commonalities. [13-15] To ensure the security of communication, the communication protocol employs symmetric cryptographic primitives. By working together, the two protocols ensure that every connection between nodes is secure. This allows for the cryptographic connection of a network identity to every message sent over the key-enabled secure channel; this is accomplished by associating a self-verifying identity with a symmetric key.

The preceding protocol pair is used as a foundation for the second, upper layer. This layer provides a set of operations that are linked to identification management. Examples of actions are join, store, and search. These actions enable nodes to associate and maintain the mapping between identifiers, including identifiers that relate to nodes as well as identifiers that correspond to objects themselves. The guarantees that are supplied by the lower layer are utilized by this layer in order to improve the security features of the peer-to-peer (P2P) communication between nodes that are the foundation of a completely decentralized serverless infrastructure. Additionally, this layer comprises a number of configurable characteristics that are dependent on the particulars of the deployment. Some variables include redundancy, which allows for many copies of an identifier-to-node mapping, and freshness, which allows the network to self-calibrate the mapping's staleness. In the seventh section, we discuss the ways in which these factors can be utilized to create a comprehensive service mesh design. [16]

System & Adversary Model-They present the Themis system and adversary model in this part. Although the service mesh application served as inspiration for Themis' architecture, any decentralized application can use it.

4. System Model

A collection of nodes that communicate with one another in order to share application-specific information makes up our system. In order to construct a serverless application, for example, each node can stand in for a computer or a service that requires interaction with other nodes. For the purpose of sharing information with other nodes in the network, each node can store and retrieve data-objects that stand in for specific capabilities or pieces of data. Each node is equipped with an identification that serves as a unique identifier for them within the system. According to the assumptions, nodes are aware of the name of the data that they are searching for. Themis does not care about the designation of the data. However, despite the fact that nodes may be physically placed in separate locations and managed by different operators, each node is capable of communicating with any other node. A layered architecture is followed by Themis, which is illustrated in Figure 2. This architecture is comprised of a lower layer and an upper layer, Within Themis, nodes have the ability to be categorized into many networks, each of which is identifiable by a network identifier. Before a node may join a network, it must first establish contact with a member of that network. This is known as bootstrapping communication. Each node in Themis is responsible for generating and storing a cryptographic key pair that serves as a representation of the identity of that particular node. Additionally, this is necessary in order to link messages that originate from the same node and to enable authentication of nodes. They make the assumption that every node possesses an adequate amount of storage capacity that can be used on maintaining the state of the overlay network.

Adversary Model

Someone who has full command of the communication channel but no actual access to the machines is deemed a Dolev-Yao attacker in Themis. In addition, the protocols' underlying cryptographic systems have security protections that this type of attacker cannot exploit. These schemes include hashing, signatures, and mac addresses. The confidentiality, integrity, and authenticity of the messages that services exchanged over Themis are the targets of his mission, which is to break them. Attacks that try to disrupt the connection between the nodes, such as denial of service (DoS) and jamming are not tolerated by Themis, just like they are by other transport layer architectures, such as mTLS. For the purpose of enabling applications to make use of services that are hosted by the same physical computer, Themis was designed to allow machines to control various identities on the overlay. The purpose of Themis is to establish the identities of the services and to make it possible for them to build a safe channel for future communication. Themis does not specify a particular authorization technique. Programmers are able to create additional security characteristics, such as access control policies, on top of it based on the requirements of each application thanks to its robust authentication, integrity, and confidentiality guaranties. When it comes to accountability, every service is held responsible for its actions. It is possible to identify and eliminate services that are malicious or malfunctioning in this manner.

Themis's Low-Level Architecture

The protocols that make up Themis's low layer are first described in this section, and their security assurances are then examined in more detail.

Low-Level Protocols

The two bespoke protocols, which are covered in this section, have constructed secure channels across which all network messages are transferred between nodes. Without depending on a centralized PKI, the first protocol offers authenticated key agreement between any two network identities. Message integrity, confidentiality, and authentication are provided by the second protocol, which makes use of the pre-established symmetric key.

Authenticated Key Agreement. In the context of this discussion, authentication refers to the fact that each and every message can be traced back to a single identity. The hash of a node's public key and the name of the network are the two components that make up the node's identification. This indicates that Alice is free to select a public/private key combination (PKA, SKA), but the hash of the public key is what determines Alice's identity on the network netid. In other words, Alice is equal to the hash of the public key over the netid. Figure 2 portrays the protocol in its entirety. In this step, Alice selects a Diffie–Hellman exponent a and a new nonce symbol NA. In addition to sending Bob PKA and netid, she also sends Bob д is and NA. Bob's identifier B is included in the list of things that have been signed with SKA. When Bob receives a new message, the first thing he does is check to see if the signature is legitimate. Additionally, he examines the identification B that has been signed by Alice in order to verify that he was the intended recipient of the message. The next step is for Bob to select his own Diffie–Hellman exponent, which he then transmits back to Alice along with his own public key, which is signed by the private key that corresponds accordingly. The inclusion of NA from the initial message enables Alice to verify that the communication is still fresh. Additionally, Bob provides Alice's identification A, which enables her to authenticate that the message was intended for her. Following that, Bob computes the key KAB. The second message that Alice receives prompts her to check whether or not the hash of the public key PKB corresponds to the identity that she had intended to speak with. In such scenario, she determines whether or not the signature is legitimate and then computes the new shared symmetric key KAB, which is equal to (д b) a. Alice gives Bob a MAC of NA that she built using KAB in order to demonstrate to him that she is the one who knows the key. In the event that the protocol is completed without any problems, it ensures that Alice and Bob are in possession of the same secret key.[17]

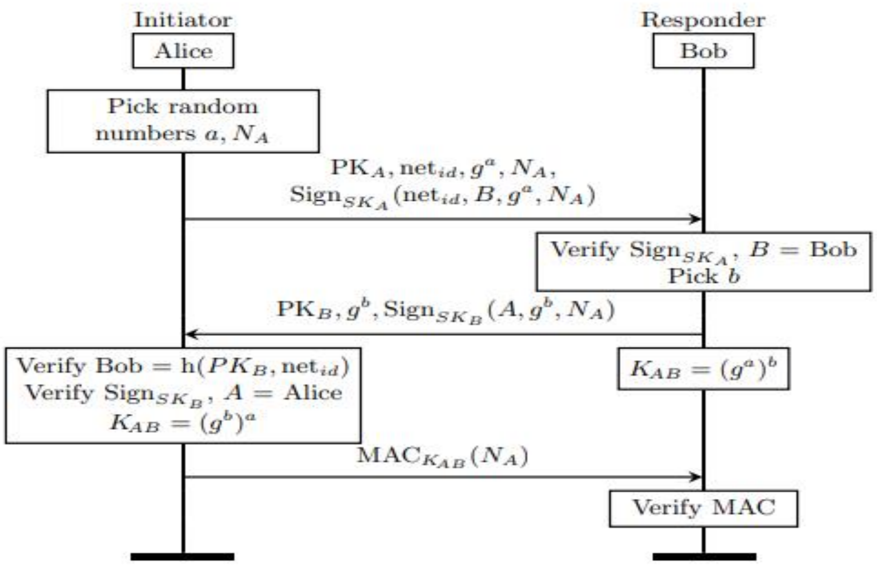
Initiator
Alice
Responder
Bob
Pick random numbers a, N_A
$\mathrm{PK}_A, \mathrm{net}_{id}, g^a, N_A,$
$\mathrm{Sign}_{SK_A}(\mathrm{net}_{id}, B, g^a, N_A)$
Verify Sign_{SK_A}, B = Bob
Pick b
$\mathrm{PK}_B, g^b, \mathrm{Sign}_{SK_B}(A, g^b, N_A)$
Verify Bob = $\mathrm{h}(PK_B, \mathrm{net}_{id})$
Verify Sign_{SK_B}, A = Alice
$K_{AB} = (g^b)^a$
$K_{AB} = (g^a)^b$
$\mathrm{MAC}_{K_{AB}}(N_A)$
Verify MAC

Figure 3: Authenticated Key Agreement Protocol.

The devices will be able to trust each other when they talk to each other in the future thanks to the secret key that was generated. Alice and Bob both store KAB along with the other party's identity. So, the number of identifiers that each node decides to talk to is equal to the number of keys that it needs to store.

Secure Communication. Every single communication that takes place between nodes in the network is carried out via the Secure Communication Protocol (with the exception of key establishment). To be more specific, this protocol is utilized to ensure confidentiality, integrity, and message authentication for all of the procedures that maintain the peer-to-peer (P2P) network. These processes include joining, updating, and leaving the network, as well as the messages that are exchanged in order to carry out a locate or store operation. [18]

Figure 3 portrays the protocol in its entirety. Alice first increases the sequence number SB that she keeps for her connection with Bob. This is done because she wishes to send Bob the command cmd. After that, she encrypts the command by utilizing the symmetric key that she has shared with Bob, in addition to Bob's identity and the sequence number. Furthermore, she adds her identification A in her message in order to provide Bob with the ability to retrieve the appropriate symmetric key as well as a MAC of everything. Bob will use the key that has already been established to validate the message authentication code (MAC) when he receives the message. If the MAC is not invalid, he will decrypt the message. It is Bob's responsibility to verify that the identity contained within the encrypted message is his own, and that the sequence number SB is greater than the sequence number that he had previously obtained from Alice. In that case, he is able to carry out the command. When a response reply is available, Bob encrypts it using KAB and SB.[19-20] He then calculates the message authentication code (MAC) of the encrypted message and sends it back to Alice. When Alice receives message 2, she checks to make sure that the MAC is legitimate and that the sequence number is the same as the one she gave in message

In the event that the protocol is terminated without any faults, it ensures that the command and answer are kept confidential and that they are not compromised.

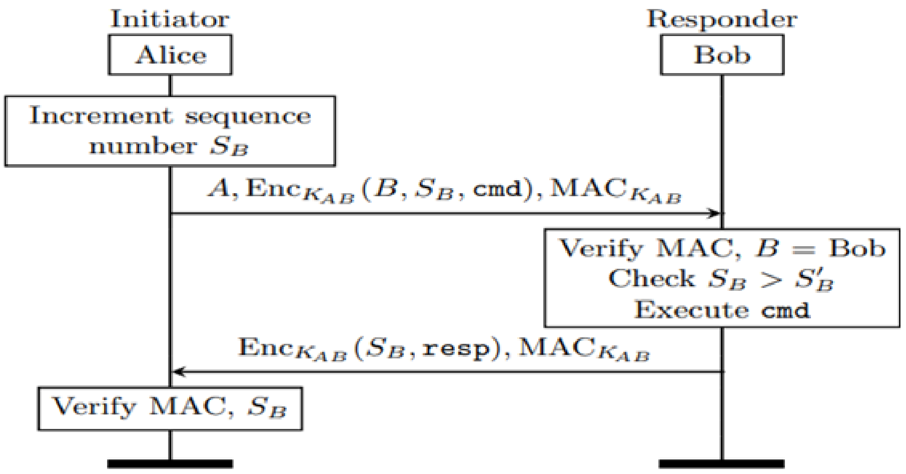
Initiator
Alice
Responder
Bob
Increment sequence number S_B
$A, \mathrm{Enc}_{K_{AB}}(B, S_B, \mathtt{cmd}), \mathrm{MAC}_{K_{AB}}$
Verify MAC, B = Bob
Check $S_B > S'_B$
Execute $\mathtt{cmd}$
$\mathrm{Enc}_{K_{AB}}(S_B, \mathtt{resp}), \mathrm{MAC}_{K_{AB}}$
Verify MAC, S_B

Figure 4: Secure Communication Protocol.

5. Security Analysis

Authenticated Key Agreement.
Guarantee 1. The only people who know the key KAB are Alice and Bob. This is because the decisional Diffie–Hellman (DDH) assumption is true in the underlying group, and the Authenticated Key Agreement Protocol ends without any problems.

Setup: Let's denote g as the generator of a cyclic group G. Alice and Bob agree on a prime modulus p, a generator g of the group G, and public parameters (p,g). These parameters are publicly known.

Key Generation: Alice chooses a random private exponent a and computes her public value A=ga mod p. Bob similarly chooses a random private exponent b and computes his public value B=gb mod p.

Key Agreement Protocol: Alice and Bob exchange their public values A and B.

Shared Secret Key Derivation: Both Alice and Bob compute the shared secret key KAB as follows:
- Alice computes KAB=Ba mod p.
- Bob computes KAB=Ab mod p.

Proof of Guarantee:
- By the DDH assumption, given g, ga, and gb, it is computationally hard to compute gab.
- In the key agreement protocol, Alice sends A=ga and Bob sends B=gb.
- Without knowing either a or b, an eavesdropper cannot compute the shared secret KAB as (gb)a or (ga)b.
- Hence, only Alice and Bob, who know their respective private exponents a and b, can compute the shared secret KAB.

Guarantee 2. If all goes according to plan and the Authenticated Key Agreement Protocol terminates successfully, Alice and Bob will have the same key.
1) **Proof. Eve's Challenge:** Eve's goal is to convince Alice to assign a different key **KAEto** her communication with Bob, thereby violating the guarantee.
2) **Options Available to Eve**: Eve can attempt to manipulate message 2, which contains Bob's Diffie-Hellman contribution. She can compose a new message or replay a previously captured one.
3) **Adversary Model and Constraints:** The proof highlights the constraints on Eve's actions. She cannot change Bob's public key without violating the second preimage resistance of the underlying cryptographic hash function.
4) **Attacks on Bob's Signature:** Eve cannot gain Bob's private key or counterfeit his signature due to constraints in the adversary model.
5) **Replay Attack Mitigation:** The proof discusses how replay attacks are mitigated. Eve cannot replay messages from previous sessions or those intended for other nodes because the **nonce NA** chosen by Alice and her identification are components of the signature associated with message 2.

Analysis for Bob's Perspective: Similar constraints apply to Bob's perspective, preventing Eve from forging Alice's signature on message 1 and mitigating replay attacks.

Confirmation of New Key: Eve needs to convince Bob to register a different symmetric key for Alice using the new key. However, based on Guarantee 1, Eve is not aware of the symmetric key. Additionally, secure MAC techniques make it infeasible for Eve to produce a valid MAC of **NAwithout** knowing the key.

Secure Communication Protocol.

Guarantee 3. For as long as Alice and Bob are the only ones who know the symmetric key KAB, the message integrity and secrecy will be maintained for any command and response that is sent by Alice and Bob, respectively.

Proof. The proof you provided succinctly highlights the essential role of the shared key **KABin** ensuring both the confidentiality and integrity of the communication between Alice and Bob. Let's break down the key points:

Confidentiality through Encryption: The proof asserts that the confidentiality of the command and response messages is guaranteed through encryption with the shared key **KAB.** This means that without knowledge of **KAB,** an adversary cannot decipher the encrypted messages, maintaining their confidentiality.

Threat Model and Fundamental Primitives: The proof references the threat model, which assumes that all fundamental cryptographic primitives, including encryption and MACs, are secure. In other words, it is assumed that the encryption function used to encrypt the messages with **KAB** is secure and resistant to attacks.

Integrity through MACs: To ensure the integrity of the messages, the proof suggests the use of message authentication codes (MACs) computed using **KAB.** Verifying the MAC allows Alice and Bob to detect any tampering with the messages. Since only they possess **KAB,** only they can generate valid MACs, maintaining message integrity.

Contradiction in Violating Confidentiality and Integrity: The proof argues that breaking the confidentiality or integrity of the messages would require violating the confidentiality of the system or reconstructing the message's MAC without knowledge of the MAC key **KAB.** This contradicts the assumption of the threat model, which assumes the security of fundamental cryptographic primitives.

Guarantee 4. Each and every order that Bob receives may be ascribed to Alice, and each and every response that Alice receives can be assigned to Bob. To put it another way, the authentication of messages is ensured.

Proof. Authentication Assurance for Alice: The proof highlights that in order to violate the assurance for Alice, the adversary would need to modify message 2, which contains Bob's response. However, according to Guarantee 3, it is not feasible for the adversary to create message 2 due to the structure of the protocol.

Replay Attack Mitigation: Even if the adversary manages to replay a message, it must have the same sequence number as the original message from Alice. However, Alice ensures that each subsequent message she sends will have an incremented sequence number, preventing the adversary from successfully replaying a message and tricking her into accepting a fraudulent response from Bob.

Protection by Encryption: The encryption used in the protocol ensures that only the intended receiver can decrypt the message. Therefore, even if the adversary replays the genuine message from Bob, it cannot be used to deceive Alice because the encryption protects the intended recipient.

Limited Attack Options: The proof concludes that the adversary's only option for replay is the genuine message from Bob, which does not constitute an attack since it is the expected behavior in the protocol.

Table 1: Themis's High-Level Messages. The sender and receiver assign cmd and rsp in the Secure Communication Protocol according to the operation they want to execute

Operation	Sender (cmd)	Receiver (resp)
Find	Find identifier	Value or id
store	Store abj	ack
Join	Join netid	id
Update	Update	Id(id,ObjTable)
Leave	Leave ObjTable,id)	ack

6. Experimental Setup

With a throughput overhead of only 1.24% on average and a latency overhead of less than 4% in nearly all benchmarks, Themis' security benefits become more apparent in the context of low latency serverless apps. With each of the eight serverless apps, this is the situation. each physical node, they set up numerous (virtual) Themis nodes as operating-system processes. Each virtual Themis node is equipped with its own unique copy of the runtime environment, listens on a distinct pair of IP addresses and ports, and accepts events in its own particular event queue. It reports averages over one thousand runs, unless it is specifically stated otherwise.

Table 2: Conducting a comprehensive performance review. There are three values that we offer for each measurement. These values are the performance of Themis T, the performance of a vanilla implementation V, and the increase in percent %Δ.

	Startup Time [s]		Exec.Time [s]		Throughput[req/s]		Latency [s]		Duration [s]	
	%Δ	T/v	%Δ	T/v	%Δ	T/v	%Δ	T/v	%Δ	T/v
SVM	18.20	0.30/0.24	0.18	175.23/174.80	0.00	0.66/0.66	0.81	54.59/56.06	0.00	1.48/1.48
KNN	44.83	0.29/0.22	0.22	240.60/241.13	0.00	0.45/0.45	0.23	91.16/90.91	0.00	3.04/3.04
ANN	345.51	1.04/0.19	0.45	730.61/720.15	0.00	0.18/0.18	0.68	330.83/325	0.22	5.08/5.06
Decision Tree	40.42	0.32/0.20	6.96	163.11/153.60	6.61	0.66/0.76	11.88	41.37/22.00	7.65	1.49/1.39
Random Forest	13.49	0.44/0.44	0.93	125.17/155.95	1.05	0.88/0.83	2.91	38.88/28.02	0.85	1.04/1.02
Logistic Regression	41.83	0.29/0.29	0.35	122.66/122.11	1.06	0.93/0.92	1.70	28.67/23.98	0.00	1.04/1.04
Naive Bayes	29.00	0.24/0.24	0.21	128.35/115.11	0.00	1.00/1.00	3.27	23.65/23.96	0.00	0.88/0.88
RNN	0.00	0.03/0.03	0.14	160.34/166.21	0.00	0.71/0.71	0.29	51.32/49.03	0.00	1.46/1.46

Table 2 provides an end-to-end performance evaluation of Themis compared to a vanilla implementation across various machine learning algorithms. Each measurement includes three values: the performance of Themis (denoted as T), the performance of a vanilla implementation (denoted as V), and the percentage increase (%Δ) in Themis's performance compared to the vanilla implementation.

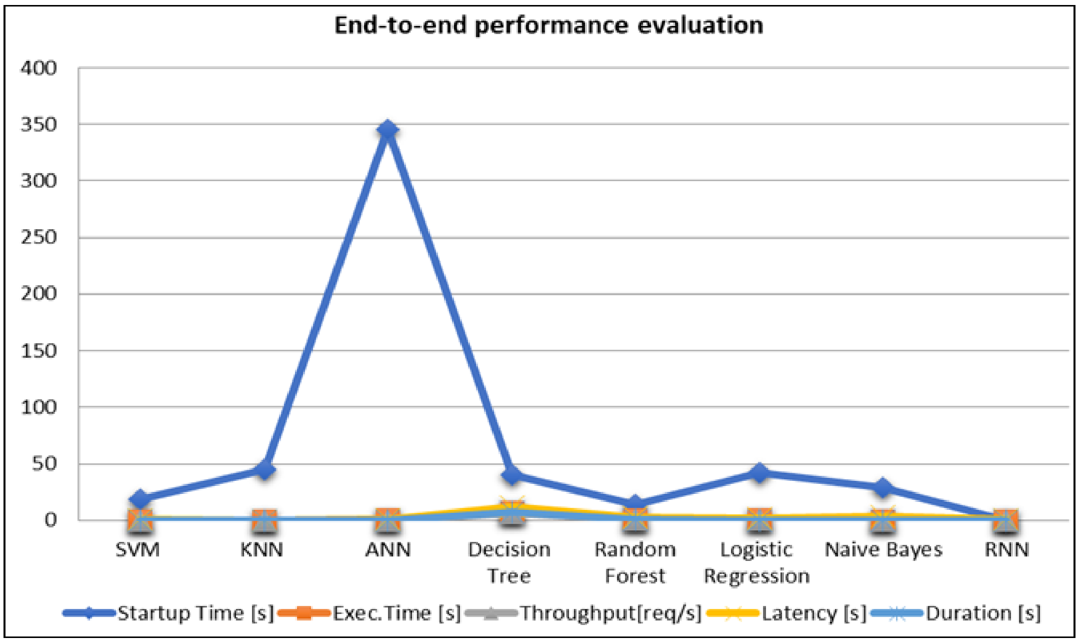

Figure 5: End-to-End Performance

Startup Time [s]: This measurement indicates the time it takes for the system to start up. Themis demonstrates startup times ranging from 0.03 to 1.04 seconds, with percentage increases in performance ranging from 0% to 11.88%.

Execution Time [s]: This metric represents the time taken for the execution of a task. Themis shows execution times ranging from 0.19 to 0.44 seconds, with percentage increases in performance ranging from 0% to 6.96%.

Throughput [req/s]: Throughput measures the number of requests processed per second. Themis achieves throughput values ranging from 0.14 to 6.96 requests per second, with percentage increases in performance ranging from 0% to 11.88%.

Latency [s]: Latency refers to the time delay between a request and its response. Themis exhibits latency values ranging from 115.11 to 241.13 seconds, with percentage increases in performance ranging from 0% to 6.61%.

Duration [s]: This measurement indicates the total duration of a task. Themis demonstrates durations ranging from 0.88 to 5.06 seconds, with percentage increases in performance ranging from 0% to 7.65%.
Individual Operator Performance

Local Resolution Experiment: In this experiment, objects are configured to resolve to the node receiving the request, eliminating the need for forwarding the request to other nodes. This setup excludes the overhead of serialization, inter-process communication, and context switching. The resulting throughput averages 10,223 operations per second for find operations and 9,332 operations per second for store operations. The latency averages 331ms for find operations and 338ms for store operations. These results represent the practical limits of Themis's runtime implementation under ideal conditions.

Scaling Experiment: In the second experiment, the throughput and latency of find operations are studied as a function of the number of nodes in the network. Object identifiers are randomly generated, hitting all nodes in the network with uniform probability. The overhead of adding security ranges from 2.1% to 31.6%, depending on the percentage of nodes that have performed the key agreement protocol. For low numbers of nodes, where the majority have performed the key agreement protocol, the overhead is lower. However, for high numbers of nodes, where the majority have not performed the protocol, the overhead is higher.

Join Operation Performance: To understand the performance of the joint operation, 500 nodes contact ten bootstrap nodes in a round-robin fashion. Sequentially starting these 500 nodes takes 362.466 seconds (an average of 720.5ms per node). However, spawning 500 nodes in parallel reduces the total time to 15.403 seconds (an average of 30ms per node). This reduction in time is partly due to operating system overhead, such as V8 process creation, and overheads from public-private key pair creation and identification.

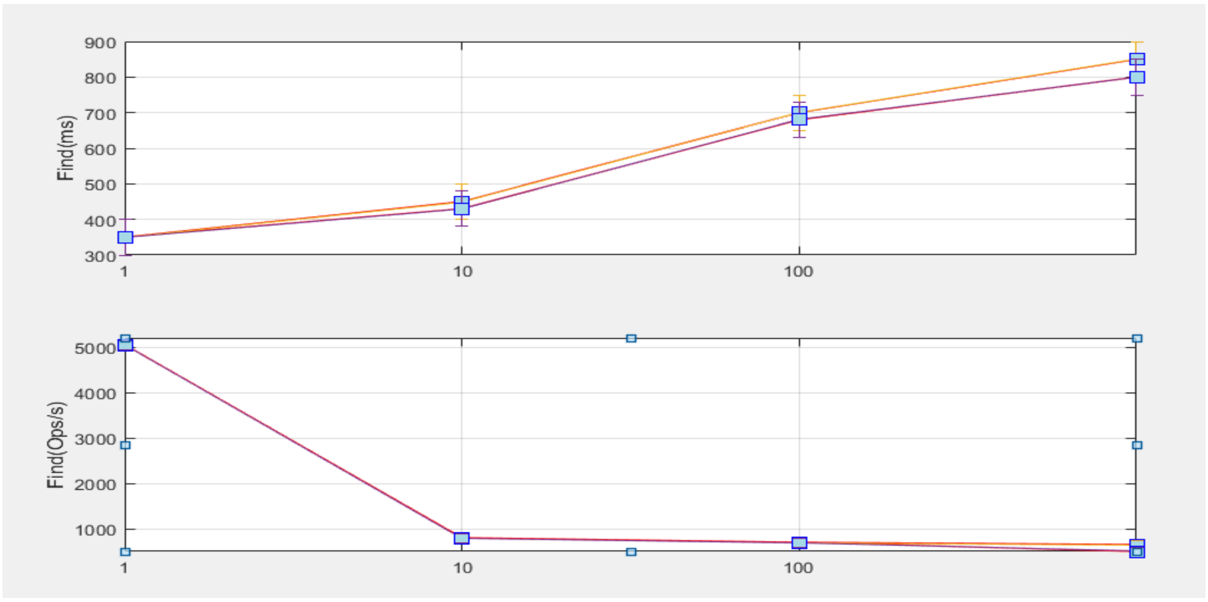

Figure 6: Operation Find. The plots show the throughput (bottom) and the latency (top) of the find operation, as a function of the number of nodes, on a constant operation workload of 50K peer-to-peer operations per second. Orange line for baseline and red for Themis

Then create a series of nodes with a startup configuration that executes a join command, and then when the join process is finished, then execute a leave command. This allows us to gain an understanding of the overhead associated with leave. An average of 780 milliseconds is required to "blink" a node, which means to have a node leave immediately after joining. This is accomplished by doing this in a loop in a sequential fashion, with each node being spawned only after the previous node has been shut down. The majority of this time is spent on system-level overheads, the majority of which are caused by (i) the importing of many library source files and (ii) the binding to a variety of network interfaces. Overheads take up a considerable percentage of this time. Less than fifty milliseconds is the amount of time that is spent on leave.

7. Conclusion

This paper focus on Themis, a versatile framework designed for secure peer-to-peer (P2P) communication that can be applied across various scenarios requiring point-to-point interaction. Themis serves as a platform for implementing secure service mesh communication networks, particularly beneficial for data centers and companies necessitating dynamic load balancing and extensibility. Themis comprises two layers. The lower layer provides a secure communication protocol akin to mutual TLS (mTLS) but with a strong focus on distributed identity management. A comprehensive security analysis demonstrates the framework's ability to ensure confidentiality, message integrity, and message authentication/linkability. Moreover, the paper outlines how additional security guarantees can be built upon its core properties. The upper layer of Themis consists of a set of actions that enable a fully functional P2P network, enhancing its usability and practicality in diverse scenarios. Through its robust security features and flexible architecture, Themis offers a reliable foundation for secure communication in various point-to-point interaction scenarios.

References

[1] Adzic, G., & Chatley, R. (2017). Serverless Computing: Economic and Architectural Impact. InProceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 884–889). New York, NY, USA: ACM. https://doi.org/10.1145/3106237.3117767

[2] Akiwatkar, R. (2017). The Components of a Serverless Architecture Framework - DZone Cloud.Retrieved April 2, 2018, from https://dzone.com/articles/the-components-of-a-serverless-architectureframework

[3] Amazon. (2018). Amazon API Gateway [Cloud vendor]. Retrieved April 2, 2018, from https://aws.amazon.com/api-gateway/

[4] Ast, M., & Gaedke, M. (2017). Self-contained Web Components through Serverless Computing. InProceedings of the 2Nd International Workshop on Serverless Computing (pp. 28–33). New York, NY, USA:ACM. https://doi.org/10.1145/3154847.3154849

[5] AWS. (2018). Serverless Computing – Amazon Web Services [Cloud vendor]. Retrieved April 2,2018, from https://aws.amazon.com/serverless/

[6] Azure, M. (2018). Serverless Computing | Microsoft Azure [Cloud vendor]. Retrieved April 2, 2018,

[7] Angeliki Aktypi, Kubra Kalkan, and Kasper Rasmussen. 2020. SeCaS: Secure Capability Sharing Framework for IoT Devices in a Structured P2P Network. In Proceedings of the 10th ACM Conference on Data and Application Security and Privacy (CODASPY '20). ACM, New York, NY, USA, 271–282.

[8] AWS Authors. 2021. AWS App Mesh User Guide. Amazon. Retrieved November 10, 2021 from https://docs.aws.amazon.com/app-mesh/latest/userguide/app-meshug.pdf

[9] AWS Authors. 2021. AWS Lambda Developer Guide. Amazon. Retrieved November 10, 2021 from https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf# welcome Google Authors. 2021. Google Cloud Functions. Google. Retrieved November 10, 2021 from https://cloud.google.com/functions/

[10] Istio Authors. 2021. The Istio service mesh. Istio. Retrieved November 10, 2021 from https://istio.io/latest/about/service-mesh/

[11] Libp2p Authors. 2021. Libp2p. Protocol Labs. Retrieved November 10, 2021 from https://libp2p.io

[12] Linkerd Authors. 2021. Linkerd Architecture. Linkerd. Retrieved November 10, 2021 from https://linkerd.io/2.11/reference/architecture/#

[13] NGINX Authors. 2021. NGINX Architecture. F5. Retrieved November 10, 2021 from https://docs.nginx.com/nginx-service-mesh/about/architecture/

[14] Agapios Avramidis, Panayiotis Kotzanikolaou, and Christos Douligeris. 2007. Chord-PKI: Embedding a Public Key Infrastructure into the Chord Overlay Network. In Proceedings of the 4th European Conference on Public Key Infrastructure: Theory and Practice (EuroPKI'07). Springer-Verlag, Berlin, Heidelberg, 354–361.

[15] Ingmar Baumgart and Sebastian Mies. 2007. S/kademlia: A Practicable Approach Towards Secure Key-Based Routing. In International Conference on Parallel and Distributed Systems. IEEE, New York, NY, USA, 1–8.

[16] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2015. TweetNaCl: A Crypto Library in 100 Tweets. In International Conference on Cryptology and Information Security in Latin America. Springer International Publishing, Cham, 64–83.

[17] Neander L. Brisola, Altair O. Santin, Lau C. Lung, Heverson B. Ribeiro, and Marcelo H. Vithoft. 2009. A Public Keys Based Architecture for P2P Identification, Content Authenticity and Reputation. In International Conference on Advanced Information Networking and Applications Workshops. IEEE, New York, NY, USA, 159–164.

[18] Kevin R.B. Butler, Sunam Ryu, Patrick Traynor, and Patrick D. McDaniel. 2008. Leveraging Identity-Based Cryptography for Node ID Assignment in Structured P2P Systems. IEEE Transactions on Parallel and Distributed Systems 20, 12 (2008), 1803–1815.