Themis: A Secure Decentralized Framework for Microservice Interaction in Serverless Computing

Angeliki Aktypi
Department of Computer Science
University of Oxford
Oxford, U.K
angeliki.aktypi@cs.ox.ac.uk

Nikos Vasilakis
CSAIL
MIT
Cambridge, U.S.
nikos@vasilak.is

Kasper B. Rasmussen
Department of Computer Science
University of Oxford
Oxford, U.K.
kasper.rasmussen@cs.ox.ac.uk

ABSTRACT

In serverless computing, applications are composed of stand-alone microservices that are invoked and scale up independently. Peer-to-peer protocols can be used to enable decentralized communication among the services that compose each application. This paper presents Themis, a framework for secure service-to-service interaction targeting these environments and the underlying service mesh architectures. Themis builds on a notion of decentralized identity management to allow confidential and authenticated service-to-service interaction without the need for a centralized certificate authority. Themis adopts a layered architecture. Its lower layer forms a core communication protocol pair that offers strong security guarantees without depending on a centralized point of authority. Building on this pair, an upper layer provides a series of actions related to communication and identifier management—*e.g.*, `store`, `find`, and `join`. This paper analyzes the security properties of Themis's protocol suite and shows how it provides a decentralized and flexible communication platform. The evaluation of our Themis prototype targeting serverless applications written in JavaScript shows that these security benefits come with small runtime latency and throughput overheads, and modest startup overheads.

KEYWORDS

serverless, security, decentralized, service mesh, data-control plane

1 INTRODUCTION

Serverless computing [20, 21] is a recent approach to cloud computing that simplifies the development and use of cloud resources. Serverless applications are comprised of stand-alone microservices that interact with each other in a peer-to-peer (P2P) fashion: each microservice is responsible for only a small fraction of the application functionality, and can thus be invoked, be passed parameters, and scale-up independently. A *service mesh* [15, 24] is a dedicated infrastructure layer that allows communication among microservices, which can belong to different deployment clusters of container platforms (e.g., Kubernetes [5]).

Existing service meshes enable microservice communication by adopting a centralized architecture—microservices can authenticate and discover each other by communicating with central registries. Relying on central registries to administrate the microservice communication provides a clear and straightforward administration. However, it makes it challenging to support open cloud platforms creating vendor lock-in issues. Allowing microservices to communicate in an ad-hoc manner can facilitate the incorporation of machines/services that belong to different federated domains, whereas the lack of dependency on a central communication link can allow the deployment of serverless applications in dynamic edge (*e.g.*, IoT) and volatile (*e.g.*, disaster zones) environments [25].

This work presents Themis, a secure general-purpose abstraction overlay suitable to any application that demands point-to-point communication but particularly tailored to the service mesh infrastructures underpinning today's serverless environments. Themis is a framework built on a notion of decentralized identity management to allow secure service-to-service interaction when the connection to a central registry might not always be available, addressing underlined design flaws in the security of state-of-art service meshes [19].

Themis's design achieves security, extensibility, and service discovery in microservice communication featuring a two-layer protocol architecture. Its lower layer consists of two protocols that provide authenticity, confidentiality and integrity to the communicated messages. Its upper layer builds on the previous protocol pairs to provide a series of P2P communication and identifier management actions. We prove the low-level protocols secure in a strong adversary model, and discuss the resulting security properties of the upper level protocols. We further describe how Themis can serve as a building platform for a service mesh application.

Our Themis prototype leverages the ubiquity of JavaScript powered runtime environments in serverless platforms, and is available to modern applications as an open-source library. We evaluate Themis on eight end-to-end serverless applications, as well as a number of microbenchmarks targeting standalone serverless operations. The key take-away from the evaluation is that the performance overhead of Themis (compared to an insecure baseline implementation) is minimal, while still providing all the security benefits described by our design. Our contributions can be summarized as follows:

- *Security Protocols*: We propose two novel protocols for key agreement and subsequent secure communication that leverage the self-certifying identities of the participating nodes. This way we do not depend on a centralized root of trust like a traditional certificate authority.
- *Model and Proofs*: We provide a detailed security analysis of the security guarantees of our protocols. Specifically, we prove that they achieve authentication, confidentiality and integrity for all the exchanged messages.
- *High-level Operations*: We specify five key operations, *i.e.*, `find`, `store`, `join`, `update` and `leave` that nodes execute to maintain a structured overlay organization. Leveraging these building blocks Themis achieves service discovery and extensibility in a fully decentralized manner. We also elaborate on the security

properties that hold against several common classes of attacks that target this P2P structure.

- *Open-source Implementation*: We implement Themis in about 3.3K lines of JavaScript, as a pluggable application library called `Themis` for building serverless applications. The `Themis` library takes care of initialization and communication across a number of common protocol configurations—*e.g.*, TCP and HTTP. We provide an open-source release of Themis's implementation, the example code and benchmarks presented, as well as evaluation scripts at github.com/xxx/themis and npmjs.org/xxx/themis.[1]
- *Empirical Evaluation*: We evaluate Themis's characteristics across eight serverless applications as well as targeting microbenchmarks scaling between 1–1,000 nodes. Themis's security benefits come at a small-to-imperceptible cost in terms of runtime performance and only modest cost in terms of lines of code changed.

2 RELATED WORK

Applying a P2P architecture to enable secure service-to-service communication in serverless environments is a relatively new area of research. Existing service meshes [1, 2, 4] follow a centralized architecture, where dedicated registries forming the control plane are responsible for coordinating the microservice proxies of the data plane. Consul [16] adopts a more decentralized approach by using the Raft consensus protocol to distribute cluster state that is centrally maintained by the quorum leader node. The work in [25] also relies on service registry to store data regarding registered services; the registry acts as a DNS server resolving human-readable names to services hashes that can be queried on top of a Distributed Hash Table (DHT) structure. Themis provides a holistic decentralized design where the information concerning the nodes and the application-specific data, *e.g.*, microservices, they provide are stored on the DHT nodes themselves obliviating the need for a service registry. However, rather than having the DHT table store the data itself such as IPFS [8], it only stores the pointer(s) (*i.e.*, node identifier(s)) of the node(s) that claims to be holder(s) for that data. As such, Themis avoids several challenges at the cost of an additional indirection—*e.g.*, value staleness, continuous or infinite streams, and redistribution over heterogeneous deployments.

A previous generation P2P architectures addressed scalability problems outside the serverless computing domain—*e.g.*, Chord [36], Kademlia [28]. The distributed nature of these architectures exposes them inherently to important attacks [38, 39], and thus many extensions provide authentication in distributed systems using a single root of trust [13, 14] or identity-based cryptography [12, 26]. Other works try to overcome single points of failure by using threshold cryptography [6, 34], consensus protocols [31, 32] and reputation mechanisms [11, 18]. Themis focuses on decentralized identity management, where nodes can directly authenticate themselves on their own without being previously paired with another entity; thus, Themis does not necessitate the support of a public key infrastructure (PKI) as done in QUIC [27] and mTLS [23]—the protocol adopted by the majority of service meshes.

To provide authentication Themis uses self-certifying identities [29] by constructing the identity that peers have in the overlay from their public key. This obliviates the need for a certificate to associate the identity that a peer has with a specific public key. The certificate can be issued by a Certificate Authority (CA) or the peer itself (i.e. self-signed certificates) as done in [40] and in [30]—that embeds in the initial handshake a signed identity payload to authenticate the static public key of the Noise_XX [22] protocol. Self-certifying identities have been used in the past in P2P architectures [7, 33] to prevent eclipse and sybil attacks. Themis focuses on different problems—confidentiality, message integrity, and message authentication/linkability—complementing these works rather than competing against them. In SECIO [9], an earlier secure transport protocol for IPFS and libp2p [3], peers use self-certifying identities without relying on certificates. Themis provides stronger security guarantees than SECIO, *e.g.*, allowing key confirmation [17] and resisting against identity-misbinding attacks [35].

3 THEMIS ARCHITECTURE

This section elaborates on Themis's design and gives a high-level description of its architecture.

3.1 Design Goals

Themis is a secure P2P communication scheme, designed to suit the implementation of a large-scale, multi-cloud, and open service mesh, by achieving:

Security: The multi-tenant nature of a service mesh demands fine-grained security guarantees. The security mechanism in place must allow for *confidentiality*, *i.e.*, that the data being transferred among two parties remains hidden, so as services coexisting on the same network cannot eavesdrop on others communication. To realize an open service mesh in which different providers can take part, the joining process must be relaxed from setup burdens such as communicating with a central authority registry. However, easing the joining procedure allows both malicious and honest nodes to coexist on the same network. For this reason, the service mesh must provide strong *accountability* for the exchanged messages so as attribution of malicious behavior to be performed. To achieve accountability, both *authentication*, *i.e.*, the parties exchanging information are who they claim to be and *integrity*, *i.e.*, the data being transferred has not been forged or tampered with, must be guaranteed. These can be the basic underneath blocks for other security primitives to be built on top (e.g., authorization).

Extensibility: Each service in a serverless application is created and killed independently, based on the usage demand. A service mesh needs to inherently provide *scalability* to serve the high replication of services. To avoid vendors lock-in issues and privacy concerns, service meshes must be *open*, *i.e.*, allowing instances to be added and removed flexibly and quickly, and be hosted on both commercial clouds and client in-house premises.

Service Discovery: Service instances must be able to discover each other based on the business logic they implement. Instances can implement different parts of the workflow of the same application or can provide *observability* functionalities to the serverless infrastructure by collecting metrics of the internal state of the system. Deploying the discovery mechanism in a centralized way limits the resilience of the serverless infrastructure against a region outage and restricts the nature of serverless applications (*e.g.*, disaster

[1] URLs blinded for submission.

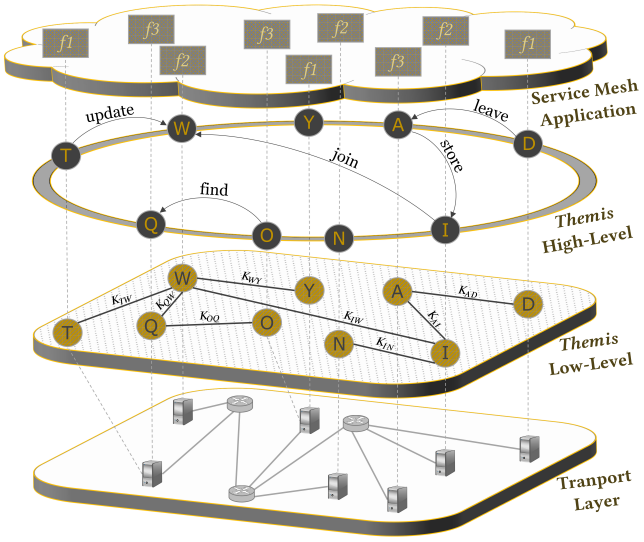

Figure 1: *Themis* sits on top of the transport layer, organizing peers on a ring topology.

management). A decentralized service discovering mechanism must allow inherently *load balancing* among the available instances that implement the same service, *fault tolerance* by allowing instances to redirect their request to a service instance with a healthy state, and *canary release* for testing new versions of services.

3.2 Overview

Themis's design features a two-layer protocol architecture. The first, lower layer described in Section 5, forms a core communication protocol pair that offers confidentiality, integrity, and authentication without depending on a centralized point of authority. This layer is comprised of two protocols: an authenticated key agreement protocol for setting up a secure communication channel between two nodes, and a protocol for direct communication between the two authenticated nodes. The key agreement protocol is somewhat comparable to the mTLS handshake protocol; the communication protocol leverages symmetric cryptographic primitives to ensure secure communication. Combined, the two protocols provide specific security guarantees to the communication channel established between two nodes, which we prove in Section 5.2. They effectively bind a symmetric key to a self-certifying identity so that any message sent over the secure channel that the key enables can be cryptography bound to an identity on the network.

The second, upper layer described in Section 6, builds on the previous protocol pair to provide a series of actions related to identifier management. Examples of actions include `join`, `store`, and `find` allow nodes to associate and manage the mapping between identifiers—both ones corresponding to nodes and ones corresponding to objects. This layer leverages the guarantees provided by the lower layer to enhance security properties on nodes' P2P communication underpinning a fully decentralized serverless infrastructure; we analyze these properties in Section 6.2. This layer also incorporates several tunable parameters that depend on deployment specifics. For example, a redundancy factor allows several copies of a single identifier-to-node mapping; a freshness factor allows the network to self-calibrate mapping staleness. We present how these parameters can be used to provide a complete service mesh architecture in Section 7.

4 SYSTEM & ADVERSARY MODEL

In this section, we provide the system and adversary model of Themis. While Themis design was motivated by the service mesh application, it is equally applicable to any decentralized application.

4.1 System Model

Our system consists of a set of nodes that interact to exchange application-specific data. Each node can represent a machine or a service that needs to interact, e.g., to build a serverless application. Each node has an identifier that uniquely identifies them in the system, and they can each store (and search for) data-objects that represent a specific capability or piece of data that they wish to make public to other members of the network. We assume that nodes know the name of the data they are looking for. Themis is agnostic to how data is named. Nodes may be physically located (and controlled by) different operators, however any node can communicate with any other node.

Themis follows a layered architecture depicted in Figure 1, which consists of a lower and an upper layer described in Sections 5 and 6, respectively. In Themis nodes can be grouped into different networks distinguished by a network identifier. To join a network, nodes need to communicate with a member of that network in order to bootstrap communication. Each node generates and stores a cryptographic key-pair which represents the identity of that node within Themis. This is needed to allow nodes to be authenticated, and to link messages coming from the same node. We assume that each node has sufficient storage space to dedicate to keeping state for the overlay network.

4.2 Adversary Model

Themis considers a Dolev-Yao attacker who fully controls the communication channel but he does not have physical access to the machines, nor he can break any of the security properties of the underlying cryptographic schemes (*i.e.*, hashing, signature, mac) used by the protocols. His goals are to break the confidentiality, the integrity and the authentication of the messages that services exchanged over Themis. Like other transport layer architectures (*e.g.*, mTLS), Themis does not address attacks that aim to disrupt the communication between the nodes (*e.g.*, denial of service (DoS), jamming). Themis allows machines to control multiple identities on the overlay; this is a design choice to allow applications to use services that are hosted by the same physical machine. Themis does not specify a specific authorization mechanism; its goal is to establish the service identities and enable them to establish a secure channel for future communication. Its strong authentication, integrity, and confidentiality guarantees allow programmers to build additional security properties (*e.g.*, access control policies) on top based on each application's needs. Every service is rendered accountable for its activity. In such a way, malicious or faulty services can be identified and removed.

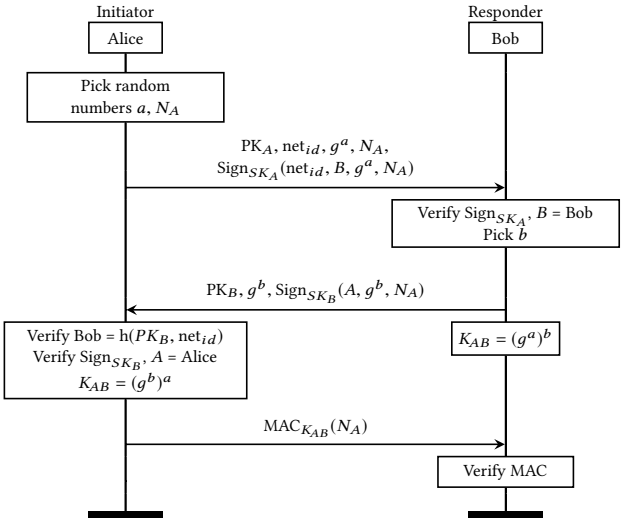

Figure 2: *Authenticated Key Agreement Protocol.* Alice and Bob establish a shared symmetric key to be used for subsequent communication.

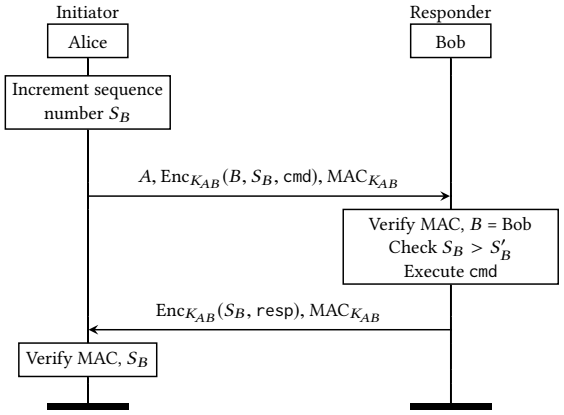

Figure 3: *Secure Communication Protocol.* Alice and Bob communicate securely using a symmetric key established with the Authenticated Key Agreement Protocol.

5 THEMIS'S LOW-LEVEL ARCHITECTURE

In this section we initially describe the protocols that constitute Themis's low layer and we further analyze the security guarantees they provide.

5.1 Low-Level Protocols

All messages between nodes in the network are sent over secure channels established by our two custom protocols described in this section. The first protocol provides authenticated key agreement between any two network identities without relying on a centralized PKI. The second protocol uses the established symmetric key to provide message integrity, confidentiality, and authentication.

5.1.1 Authenticated Key Agreement. Authentication in this context means that all messages can be attributed to exactly one identity. The identity of a node is the hash of its public key and the name of the network. That means that Alice can freely pick a public/private key-pair (PK_A, SK_A) but the hash of the public key determines Alice's identity on the network net_{id}, i.e., $Alice = \text{h}(PK_A, \text{net}_{id})$.

The protocol is shown in Figure 2. Alice picks a Diffie–Hellman exponent a and a fresh nonce N_A. She sends g^a and N_A to Bob along with PK_A and net_{id}. Everything, including Bob's identifier B, is signed with SK_A. When receiving a new message, Bob first verifies that the signature is valid. He then inspects the identifier B that is signed by Alice to check that he was the intended recipient. Bob then picks his own Diffie–Hellman exponent b and sends g^b back to Alice along with his own public key, signed by the corresponding private key. Bob includes N_A from the first message to allow Alice to confirm freshness, and Alice's identifier A to allow her to verify that the message was meant for her. Bob then computes the key K_{AB}. When Alice receives message 2 she verifies that the hash of the public key PK_B corresponds to the identity she was intending to communicate with. If that is the case she checks that the signature is valid and computes the new shared symmetric key $K_{AB} = (g^b)^a$. To prove to Bob that she knows the key, Alice sends him a MAC of N_A created with K_{AB}.

If the protocol terminates without errors, **it guarantees that Alice and Bob share the same secret key**. The resulting secret key will be used in subsequent communication between the devices, enabling them to authenticate each other. Both Alice and Bob store K_{AB} together with the identifier of the other party. Hence, the number of keys that each node has to store is proportional to the number of identifiers with which it chooses to communicate.

5.1.2 Secure Communication. The Secure Communication Protocol is used for all communication between nodes in the network (except for key establishment). Specifically, all the procedures used to maintain the P2P network (i.e., `join`, `update`, `leave`), as well as the messages exchanged to execute a `find` or `store` operation, use this protocol to provide confidentiality, integrity and message authentication.

The protocol is shown in Figure 3. Alice, who wants to send the command `cmd` to Bob, first increments the sequence number S_B she maintains for her communication with Bob. She then encrypts using the symmetric key she shares with Bob the command along with Bob's identity and the sequence number. In her message, she also includes her identity A to allow Bob to retrieve the correct symmetric key and a MAC of everything. When receiving the message Bob will verify the MAC using the already established key, and if the MAC is valid he decrypts the message. Bob ensures that the identity in the encrypted message is his, and that the sequence number S_B is higher than the previous sequence number he received from Alice. If so, he can process the command. When a response `resp` is ready, Bob encrypts it with K_{AB} together with S_B, he calculates the MAC of the encrypted message and send them back to Alice. When Alice receives message 2, she verifies that the MAC is valid and that the sequence number corresponds to the one she sent in message 1.

If the protocol terminates without errors, **it guarantees confidentiality and integrity of the command and response**.

5.2 Security Analysis

In this section we prove the security guarantees provided by our protocols presented in the previous section (Sec. 5.1). For our analysis we assume an attacker under the threat model as described in Section 4.2.

5.2.1 *Authenticated Key Agreement.*

Guarantee 1. If the Authenticated Key Agreement Protocol terminates without errors and the decisional Diffie–Hellman (DDH) assumption holds in the underlying group, the key K_{AB} is known only to Alice and Bob.

Proof. The only terms in the Authenticated Key Agreement protocol that are related to the key are the terms $a, b, g^a, g^b, MAC_{K_{AB}}$ and of course the key itself. Of these a and b are never visible to the adversary as they are picked fresh during the protocol, and remain internal to Alice and Bob, respectively. By the assumption that all the underlying cryptographic primitives are secure the adversary cannot obtain the key from the MAC. That leaves g^a and g^b. If the adversary has a method of getting K_{AB} from these values then he can use that method to break the Diffie-Hellman assumption, which is a contradiction. □

Guarantee 2. If the Authenticated Key Agreement Protocol terminates without errors both Alice and Bob will be in possession of the same key.

Proof. We prove this for Alice and Bob individually, starting with Alice. If the protocol completes successfully, Alice knows that she shares K_{AB} with Bob. For an adversary, Eve, to break this guarantee and convince Alice to assign another key K_{AE} to her communication with Bob, Eve would have to successfully send message 2, as this is the only way for Alice to obtain Bob's Diffie-Hellman contribution. Eve has two options to send message 2: she can either craft the message or replay a previous captured message.

In order to craft message 2 the adversary has to find another key pair where the public key obeys the following $B = \mathrm{h}(PK_E, \mathrm{net}_{id})$. This means that Eve has to break the second preimage resistance property of the underlying cryptographic hash function, which again contradicts the adversary model. Without the ability to change the keys for Bob's signature, the adversary has to either forge Bob's signature or obtain his private key. Both again contradict the adversary model. That only leaves the option to replay a previous message. For replay to work, Eve has to make sure that the content of message 2 expected by Alice, corresponds to one of the messages available to Eve. Because the nonce N_A selected by Alice and Alice's identifier are both part of the signature in message 2, it means that Eve cannot reuse a message from a previous session or one that Bob sent to another node, to attack Alice. Eve would have to force Alice to chose a nonce that corresponds to one of Eve's captured messages; however, according to our threat model, Eve cannot influence Alice's choice of N_A.

We now prove the same for Bob. By the same argument as above the adversary cannot forge Alice's signature on message 1. However replay is a different story. Bob does not have a way to readily check if message 1 is a fresh message from Alice or indeed a replay from Eve, so Eve can initiate a protocol run. However, in order to successfully fool Bob and make him register a different symmetric key for Alice, Eve has to confirm the new key in message 3. By Guarantee 1, Eve does not know the symmetric key even if she sent the first message. That means that she has to produce a valid MAC of N_A without knowing the key. For a secure MAC scheme this is not possible, and guessing either the key or the mac-value itself is only possible with negligible probability. □

5.2.2 *Secure Communication Protocol.*

Guarantee 3. As long as the symmetric key K_{AB} remains known only to Alice and Bob, message confidentiality and integrity is preserved for any command and response sent by Alice and Bob respectively.

Proof. Both the command and response are solely sent encrypted with K_{AB}. The only way to break confidentiality of the command or response is to break the confidentiality of the encryption function. This contradicts the threat model which states that all underlying primitives are secure. In order to violate message integrity the adversary would have to recreate the MAC of the message without knowing the MAC-key. This again contradicts the threat model which states that all underlying primitives are secure. □

Guarantee 4. Any command received by Bob can be attributed to Alice and any response Alice receives can be attributed to Bob. In other words, message authentication is guaranteed.

Proof. This guarantee has to be shown for Alice and Bob individually. To break the guarantee for Alice the adversary would have to manipulate message 2. By Guarantee 3 message 2 cannot be crafted, so that only leaves replay. To successfully convince Alice that a false response came from Bob, the adversary has to replay a valid message containing the same sequence number that Alice used at the start of the protocol. The sequence number is incremented before each new transmission by Alice so two packets from different sessions will never use the same number. The only message that uses the same sequence number is the one just sent by Alice, but the encryption contains the intended receiver so it cannot be used to replay (reflect) message 2. The only message the adversary can replay is the true message sent by Bob which is not an attack.

To break the guarantee for Bob the adversary must manipulate message 1. By Guarantee 3 message 1 cannot be crafted, so that only leaves replay. To succeed the adversary must replay a message containing a sequence number that is bigger than the biggest one Bob has yet received. That means the adversary can only drop messages but never deliver them out of order. In fact the adversary cannot use any old messages that was sent prior to the last message received by Bob. □

6 THEMIS'S HIGH-LEVEL ARCHITECTURE

In Section 5.1 we presented the two protocols that construct Themis and in Section 5.2 we proved the security properties they provide. In this section, we elaborate on how Themis builds a secure structured P2P platform.

6.1 High-Level Protocols

To support a decentralized identity management, Themis organizes nodes on a ring topology that follows a clockwise order and defines five key operations, `find`, `store`, `join`, `update` and `leave`. From

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`, obj)	ack
Join	(`join`, net_{id})	id
Update	`update`	id **or** (id, $ObjTable$)
Leave	(`leave`, $ObjTable$, id)	ack

these five operations, `find` and `store` support the identifier-to-object mapping whereas `join`, `update` and `leave` support maintenance procedures for P2P organization. Every peer initially executes Themis's *Authentication Key Agreement* protocol depicted in Fig 2 with each other node it wants to communicate. This handshake allows nodes to establish a secure communication channel based on a secret symmetric key. To execute the P2P operations, nodes send messages using Themis's *Secure Communication* protocol depicted in Fig 3. The sender and the receiver assign the `cmd` and `rsp` variables according to the respective P2P operation as illustrated in Table 1. In the following paragraphs we describe the P2P operations.

Find: This operation allows nodes to reach specific identifiers, *i.e.*, addresses on the ring topology. The identifier that is specified each time by the initiator can refer to a node ($node_{id}$) or to an application-specific data ($object_{id}$), which is generated by applying a *hash function* $h(m)$ on the data descriptor. When nodes receive a `find` request, will try to resolve it by examining if this identifier is included in their routing table (RT_{id} – the table where nodes store pointers to other nodes around the ring) or in their object table (OT_{id} – the table where nodes store mappings between $node_{id}$ and $object_{id}$ for the objects they are responsible for). The `find` operation for a $node_{id}$ terminates to a node that has this $node_{id}$ in its RT_{id}, returning the associated communication address of the specified $node_{id}$. In the case of an $object_{id}$, `find` terminates to the responsible for this object node, who will return the associated values $\{node_{id}\}$. In case the receiver of the request does not have this identifier neither in its RT_{id} or its OT_{id}, it will return the $node_{id}$ (and its communication address) from its routing table that is closer to the sought-after identifier. To allow the initiator to monitor how her request is progressively resolving, we adopt an iterative routing where all the traffic is handled by the requesting node; the responder will return the closest node to the initiator, who will then initiate a new operation with this new node.

Store: This operation provides the possibility for each node to associate its $node_{id}$ with a specific $object_{id}$. The node who wants to be associated initially starts a `find` operation for this $object_{id}$; this will provide the possibility to the initiator to retrieve the responsible node for this object. As soon as the node has this information, it will contact the responsible node specifying the object with which it wants to be associated. The responder will then amend the list for this object with the $node_{id}$ of the initiator. Themis achieves fault tolerance by adopting a replication mechanism that maps application-specific data to multiple objects. In particular, every name that is defined by the representation mechanism in use is hashed together with a counter number $object_{id} = h(name_{id}||r)$ where $r \in [0, k]$; thus, for a single application-specific data nodes create and store $k + 1$ objects. The value k is a network constant agreed among the participant nodes, which determines an upper bound of the redundant objects that will be stored on the network. However, nodes are free to decide the value k they will use for every application-specific data. The responsible nodes drop the values they store in between specific time intervals t. For this reason, peers contact the responsible nodes periodically, to make sure that their associations are maintained registered on the network.

Join: To learn the first node who succeeds them in the network, nodes execute the *join* operation with a bootstrapping node, a node which they know and which is already a member of this network. The bootstrapping node will initiate a `find` operation specifying the newcomer's $node_{id}$ as the sought-after identifier. Based on the routing procedure explained above, the `find` operation will return at the end the first successor of this identifier in the network. The responder will return to the initiator its first successor who upon receiving its response will save it as its first successor in its RT_{id}.

Update: To maintain a consistent mapping between nodes and objects, nodes periodically execute the *update* operation in specific time intervals. During this operation every node will contact its first successor to check if it is actually the node who succeeds it in this network. The responder, upon receiving the initiator's request will retrieve his predecessor and check if his predecessor is also a predecessor of the initiator. If this is the case, the responder saves the initiator as his predecessor and checks to see if the $node_{id}$ of the initiator is closer than his identifier to any of the objects for which he is responsible. If there are such objects, he will pass them with his response to the initiator who will now become the responsible for these objects node. If the predecessor of the responder is between the responder and the initiator identifiers, the responder will send back to the initiator his predecessor $node_{id}$ as this is her correct first successor who she needs to communicate.

Leave: In Themis nodes maintain in their OT_{id} the associated values for all the objects for which they are responsible. The *Leave* operation allows for node departures from a network to occur without any loss of associations. The initiator who wants to leave the network communicates her first successor to pass the information related to the objects for which she is responsible together with the $node_{id}$ of her predecessor. The responder will store and become now responsible for these objects. He will also update his predecessor so as to point to the initiator's predecessor. In Themis nodes are free to leave and join at any time. Nodes can abandon any network in which they participate without notifying other nodes, in this case any node who will try to contact them will receive a timeout and the node will be presumed dead, and all the associated values for the objects for which they were responsible will be lost. The rest nodes of the network can retrieve the lost associations by initiating a find request for one of the k remaining objects.

6.2 P2P Attacks

In this section we elaborate on the security properties of the P2P operations against common attacks of DHT networks [38].

Sybil attack: The self-certifying identifiers that Themis provides allow every device to participate in the network without the need for any initial communication. They also allow devices to create as many identities as they want, known as a Sybil attack, which can be problematic in certain applications. Themis does not consider this behavior an attack since the properties of message linkability and authentication still apply for every identity. Some applications might want to use different identities to provide different services—even though these are hosted by the same device. Our solution makes that possible. For applications where Sybil attacks are problematic, Themis provides the application layer with enough information, *i.e.*, the communication address of each identity, to implement checks for which identities are allowed to run on which devices.

Eclipse (Routing Table Poisoning) attacks: In an Eclipse attack an attacker tries to isolate a node from honest peers by placing malicious nodes as its neighbors. It is possible for an attacker to insert incorrect information into the routing table of a device if that device happens to contact an adversary. However, because of the circular structure of the addresses, it is always possible to check if a `find` request is making progress towards its goal. If progress is not being made the device can make sure not to contact the same identity again. As long as there are still honest nodes present in the routing table, the device will eventually ask one of them and get useful information. A coordinated attack against a specific device could result in DoS, but that falls outside of our threat model (Sec. 4.2).

Storage attacks: Themis enables nodes to store an association between an application-specific data (*e.g.*, a filename, a capability, a service, etc.) object and its identity in the network. Specifically the association is kept by $k+1$ different nodes, and some of those nodes might decide to drop the association, or make up an association that does not exist (ghost objects). If that happens other devices that search for an object will either not find it or find a ghost object that matches their search. This is a nuisance but will never result in a violation of any of our security guarantees. Any object is stored in the network $k+1$ times so if an association is drooped by a few nodes the other objects will still be available. Furthermore, it is the responsibility of any node that stores data in the network to regularly check and re-store the data if objects are missing, so such an attack has limited effects. If a ghost object is returned from a `find` operation, it will result in a connection attempt to the identity pointed to by the ghost object. That identity can either signal that the association is unknown, in which case the attack resulted in a single unnecessary connection and nothing else. If the ghost object points to an attacker that then provides a service, that is not an attack on Themis. This is equivalent to an attacker announcing a service under a specific name and then providing that service. The service itself might be malicious, but that is a problem for the application layer as Themis does not know or care what data is transferred, only that the identity of the communicating partners cannot be falsified.

7 THEMIS SERVICE MESH

We now elaborate on how Themis can be used to assist programmers tasked with the development of serverless applications by taking care of all the low-level details of a secure and scalable communication across serverless nodes. Let's assume an example application, *AppAuth*, which consists of two microservices: an ingress point, *i.e.*, `handler` and a processing back-end, *i.e.*, `check`. When the `handler` microservice receives a web `request` carrying a `username` and `password`, uses `check` to fetch a dictionary of known users' credentials, validates the received pair, and then handles it appropriately by sending a response message and associated code.

The programmer starts by creating a mesh with a specific network identifier to group the microservices that need to communicate, specifying a bootstrapping node. In our example, microservices `joins` a service mesh with identifier `twiitr` and bootstrapping point the route `/main`. Microservices are placed on a ring topology based on each microservice's node identifier, following a key-based routing scheme. Themis allows *service discovery* by creating mappings between the node identifiers and the microservices. A node first creates a cryptographic public-private key pair and generates its identifier $B = h(PK_B, twiitr)$. It then calculates an object identifier for the microservice it implements, by hashing the service's name interface, *e.g.*, $Object_B = h(check)$. A mapping is created by initiating the `store` operation that informs the responsible node for $Object_B$ to amend the object's value with its node identifier B. Nodes that request `check`, for example node A, retrieve B by initiating a `find` operation for $Object_B$. In fact, the `find` operation returns to the requester a list of node identifiers that have executed the `store` operation specifying $Object_B$. There is one node responsible for every object identifier. Themis adopts a redundancy mechanism having one microservice to be associated with multiple object identifiers. In our example, assuming a redundancy factor $k = 2$, the `check` microservice will be associated to three sibling object identifiers, $Object_B$, $Object_{B1} = h(check, 1)$ and $Object_{B2} = h(check, 2)$, for which node B initiates three separate `store` operations. Node A can retrieve B by initiating a `find` operation for any of the three object identifiers. The returned list and the redundancy factor provide *fault tolerance*; in case of failing nodes the requester can select another node from the list or initiate a `find` operation for a sibling object, until a node with a healthy state is found. Nodes select randomly over the returned list the node to communicate; thus, the load is evenly distributed between the replicated nodes. Further to the dynamic *load balancing* property that this technique provides, it also enables *canary releases*. As nodes are evenly selected, programmers can deploy a new version of a microservice progressively on different machines based on their nodes identifiers.

In a centralized architecture configuration functionalities (*e.g.*, authentication, discovery) are implemented in whole or in part by control plane registries which data plane proxies need to communicate to operate. Themis couples the data and the control plane implementing a fully decentralized application. In our example the responsible node for $Object_B$, *e.g.*, node C, fulfils the task of a central service discovery registry redirecting A to B. Themis leverages the DHT to route messages among peers; thus, it inherits its *scalability* property to accommodate the high rate of replication services. Abolishing managerial registries provides *openness* that facilitates migration off commodity serverless platforms, particularly beneficial to enable access and processing of data hosted on edge devices. Decreasing programmers lock-in can allow them to invent

Table 2: *End-to-end performance evaluation.* For each measurement we present three values: 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
DecisionTree	19.23	0.31 / 0.26	0.20	176.25 / 175.90	0.00	0.68 / 0.68	0.91	57.59 / 57.07	0.00	1.45 / 1.45
K-Means Clustering	45.86	0.31 / 0.21	0.23	245.70 / 245.14	0.00	0.49 / 0.49	0.23	92.19 / 91.98	0.00	2.02 / 2.02
Knn	356.52	1.05 / 0.23	0.48	731.69 / 728.20	0.00	0.16 / 0.16	0.58	336.93 / 334.98	0.33	6.03 / 6.01
LinearRegression	43.48	0.33 / 0.23	7.97	169.10 / 156.62	7.79	0.71 / 0.77	12.99	53.48 / 47.33	7.75	1.39 / 1.29
NLP	12.50	0.45 / 0.40	0.95	129.19 / 127.98	1.06	0.93 / 0.94	3.94	34.30 / 33.00	0.95	1.06 / 1.05
NaiveBayes	42.86	0.30 / 0.21	0.39	124.62 / 124.14	1.03	0.96 / 0.97	2.80	31.89 / 31.02	0.00	1.02 / 1.02
RandomForest	30.00	0.26 / 0.20	0.22	120.40 / 120.13	0.00	1.00 / 1.00	2.69	29.74 / 28.96	0.00	0.99 / 0.99
Unweighted Shortest-Path	0.00	0.04 / 0.04	0.13	165.46 / 165.24	0.00	0.73 / 0.73	0.37	52.23 / 52.04	0.00	1.36 / 1.36

their own configuration services, *e.g.*, *observability* microservices exposed through special service name interfaces, which application microservices can discover following the same discovery mechanism explained above. The low-level protocols of Themis render nodes *accountable* for the messages they exchange. Using every message as a node behaviour trace, faulty or malicious activity can be identified and resolved.

8 IMPLEMENTATION

We have implemented a prototype of Themis in about 3.3K lines of JavaScript available on `npm` via `npm -i @xxx/themis`. JavaScript was chosen due to its ubiquity in serverless environments. Themis sits on top of the transport layer; our implementation is designed to operate atop a series of communication protocols, *e.g.*, TCP, UDP and HTTP, passing an initial communication object to the `start` method. The Themis implementation avoids platform and runtime-specific configuration, and thus can execute atop any JavaScript runtime environment. A thin command-line wrapper allows the construction of virtual nodes for testing and experimentation, as Unix processes running Node.js [37]. Node.js is a JavaScript runtime that bundles (i) Google's V8, a high-performance JIT compiler, (ii) libUV, asynchronous cross-platform OS wrappers, and (iii) a small set of standard libraries (*e.g.*, `crypto`).

Themis uses the built-in EcmaScript object type to maintain an in-memory mapping between strings to identifiers (along with their timeouts and debug metadata). In cases where the host environment additionally supports persistent storage, Themis stores persistent data using non-blocking I/O: data are always cached and served from memory, and persistent storage is used to facilitate service restarts. In terms of cryptographic support, our Themis prototype offloads operations to NaCl [10]. NaCl is a high-speed constant-time cryptographic library, which we compiled to JavaScript using Emscripten (adding another 2K LoC). NaCl offers high performance cryptographic primitives, avoids calls to dynamic memory allocation functions such as `malloc` and `sbrk`, and uses small amounts of stack space. Upon bootup, a service generates a new secure public-private key pair. For the communication between two services, NaCl provides primitives that combine a receiver's public key with the sender's private key to derive a common symmetrical key. This key is then used to symmetrically encrypt and authenticate plaintext messages (in Themis, serialized buffers).

Themis is accessible as a software module (library), implantable into (often, pre-existing) programs using a conventional `import` statement. Upon import, the Themis library (1) checks for the existence of a public-private key pair, and if non-existent generates a fresh one; (2) loads and binds a listener of the chosen transport protocol on a pre-specified port (consecutive ports for multiple nodes running on a single physical host); (3) returns the `Themis` object, which is used to access `Themis`'s interfaces.

9 EVALUATION

In this section, we investigate the following questions: (Q1) What is the performance and scalability characteristics of serverless applications built on top of Themis, and how do they compare to Themis-less versions (Sec. 9.1)? (Q2) How do different Themis operations perform in the limit, and how do they compare with their insecure counterparts (Sec. 9.2)? To answer these questions, we perform experiments across two distinct environments. For Q1, we use as benchmarks eight serverless applications, outlined in Table 2 (alphabetical order), whereas for Q2, we use synthetic microbenchmarks that stress individual Themis operations.

Result Highlights: Across all eight serverless applications, the security benefits of Themis come at an imperceptible throughput overhead (1.24%, on average) and a small latency overhead ($< 4\%$ in almost all the benchmarks), more pronounced in the context of low-latency serverless applications. The application startup overhead introduced by Themis is 356.52% in the worst case, but remains under 1s and is an one-off cost amortized across the long execution times typical for serverless applications.

Experimental Setup: On the hardware side, for Q1 we use an 8-node distributed cluster on Microsoft's Azure Cloud. It amounts to eight DC1s v.2 machines equipped with Intel Xeon E-2288G CPUs and 4GB of memory, and running Ubuntu 18.04 LTS with kernel version 5.4.0-147. For Q2, we use a large-scale multiprocessor equipped with an 128-core Intel Xeon E7-8830 processor at 2.13GHz, 512GB of memory; it is running Debian 4.19.160-2 with kernel version 4.19.0-13. This environment is used to launch hundreds of micro-services, avoiding the non-determinism introduced by network operation and allowing us to zoom into Themis-inherrent overheads. On the software side, we use Node.js v8.9.4, bundled with V8 v6.1.534.50, LibUV v1.15.0, and `npm` version v6.4.1. We launch multiple (virtual) `Themis` nodes as operating-system processes on

each physical node. Each virtual `Themis` node has its own copy of the runtime environment, listens on a separate (ip, port) pair, and accepts events in its own event queue. Except when noted otherwise, we report averages over 1K runs.

9.1 Q1: End-to-End Performance

To understand the performance and scalability overheads introduced by Themis, we compare the Themis-augmented benchmark applications against their non-secure counterparts—*i.e.*, ones that do not incorporate Themis' security components. Table 2 presents the results of Themis's end-to-end performance evaluation across five metrics: startup time, execution time, throughput, latency and duration. Each measurement cell contains three numbers—a tuple of the form (Δ, V, T) where Δ is the percent difference between the vanilla and Themis-augmented versions of the serverless application, and V and T is the absolute measurement corresponding to the vanilla and the secure, Themis-augmented implementation, respectively.

Regarding startup time, *i.e.*, , the duration cost of reaching a stable state—registering all relevant services and creating encrypted communication channels between them—Themis introduces an average overhead 0.16*s*. The total execution time, *i.e.*, the end-to-end time to run the full load, shows minimal differences (under 1% in most of the cases). Similarly, the request throughput, *i.e.*, the number of requests a serverless application handles per second, shows identical performance for the majority of the evaluated applications. Themis has a higher impact on the application Latency, the average time to execute a single request. Themis introduces a maximum of 13% latency overhead to each request, whereas the execution overhead of each request ranges between 0–7.75%.

9.2 Q2: Individual Operator Performance

To understand Themis's performance of `find` and `store` operations across different scales, we perform two experiments where we insert and retrieve 1M 16-byte data objects at a constant rate of 50K objects per second. In the first experiment, the objects are configured to resolve to the node receiving the request to store or retrieve the object—*i.e.*, the node does not need to forward the request to other nodes. By resolving the request locally, *i.e.*, excluding multiple hops of serialization, inter-process communication, and context switching, the throughput and latency results show the best-possible performance achieved by our runtime implementation—*i.e.*, Themis's practical limits due to the implementation's runtime environment. The resulting throughput averages 10223 and 9332 operations per second for `find` and `store`, respectively; the resulting latency averages 331ms and 338ms for `find` and `store`, respectively.

In the second experiment, we focus on the throughput as a function of the number of nodes. Object identifiers are now randomly generated, and thus are expected to hit all the nodes in the network with uniform probability. Fig. 4 shows the throughput and the latency of the `find` operation as a function of the number of nodes. The overhead of adding security ranges between 2.1–31.6% and depends critically on the percentage of nodes that have already performed the key agreement protocol. For low numbers of nodes (left side of plots), the majority of nodes have performed the key agreement protocol, whereas for high numbers of nodes (right side of plots), the majority of nodes have not.

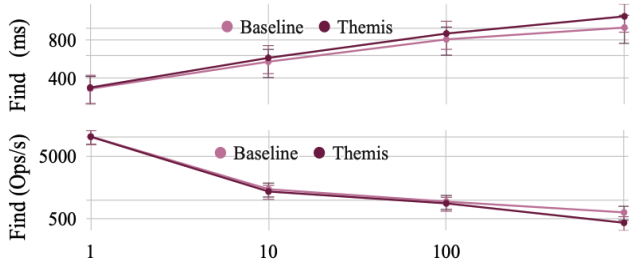

Figure 4: *Operation Find.* The plots show the throughput (bottom) and the latency (top) of the `find` operation, as a function of the number of nodes, on a constant operation workload of 50K peer-to-peer operations per second.

To understand the performance of the `join` operation, we have 500 nodes contact ten bootstrap nodes in round-robin fashion. Starting these 500 nodes sequentially takes 362.466s (an average of 720.5ms/node). If, however, we spawn 500 nodes in parallel, we get a total of 15.403s (an average of 30ms/node). A part of this overhead includes operating system overheads such as V8 process creation, which averages about 320ms per node. Other overheads arise from the identify (public-private key pair) creation and the authenticated key agreement protocol, averaging about 52ms.

To understand the overhead of `leave`, we launch a series of nodes with a startup configuration that runs a `join` followed by a `leave` command when the `join` command completes. We run this sequentially in a loop where we spawn a node only after the previous node has shutdown; on average, "blinking" a node (*i.e.*, have a node `leave` right after `join`ing) takes a total of 780ms. Much of this time is spent in system-level overheads, most of which is from (i) importing multiple library source files and (ii) binding to various network interfaces. The overhead of `leave` amounts to less than 50ms.

10 CONCLUSION

This paper presents Themis, a framework for secure P2P communication that is general enough to be usable in a variety of scenarios that demand point-to-point interaction. In this paper we have shown how Themis can serve as a platform for implementing a secure service mesh communication network for use in data centres and companies that need dynamic load balancing and extensibility. Themis consists of two layers. Its lower layer provides a secure communication protocol, similar to mTLS in many ways but with a strong emphasis on distributed identity management. We provide a thorough security analysis that proves the claimed security guarantees, *i.e.*, confidentiality, message integrity, and message authentication/linkability. We emphasize how any additional guarantees can be built on top, leveraging its core security properties. Its upper layer consists of a set of actions that offer a fully functional P2P network. We use Themis to implement a service mesh communication network in order to perform a detailed evaluation. Our open-source prototype achieves both throughput and latency on par with other (insecure) P2P networks despite the fact that it provides strong security guarantees.

ACKNOWLEDGMENTS

This research was funded partly by DARPA contract no. HR00112020013 and no. HR001120C0191. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA or other agencies.

REFERENCES

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

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

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

[4] Open Service Mesh Authors. 2021. *Open Service Mesh Docs*. Microsoft. Retrieved November 10, 2021 from https://docs.openservicemesh.io

[5] The Kubernetes Authors. 2021. *What is Kubernetes?* Kubernetes. Retrieved November 10, 2021 from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[6] 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.

[7] 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.

[8] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561 [cs.NI]

[9] Juan Benet, Bigs, and Yusef Napora. 2021. *Secio Specification*. libp2p. Retrieved November 10, 2021 from https://github.com/libp2p/specs/tree/master/secio#shared-secret-generation

[10] 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.

[11] 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.

[12] 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.

[13] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. 2002. Secure Routing for Structured Peer-to-Peer Overlay Networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 299–314.

[14] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. 2009. Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communications Magazine* 47, 12 (2009), 94–101.

[15] Amine El Malki and Uwe Zdun. 2019. Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures. In *Software Architecture*. Springer International Publishing, Cham, 3–19.

[16] Jeff Escalante and Zachary Shilton. 2021. *Consul Architecture*. HashiCorp. Retrieved November 10, 2021 from https://www.consul.io/docs/architecture

[17] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. 2016. Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy (S&P '16)*. IEEE, New York, NY, USA, 452–469.

[18] Rohit Gupta and Arun K. Somani. 2004. Reputation Management Framework and Its Use as Currency in Large-Scale Peer-to-Peer Networks. In *Fourth International Conference on Peer-to-Peer Computing*. IEEE, New York, NY, USA, 124–132.

[19] Dalton A. Hahn, Drew Davidson, and Alexandru G. Bardas. 2020. MisMesh: Security Issues and Challenges in Service Meshes. In *Security and Privacy in Communication Networks*. Springer International Publishing, Cham, 140–151.

[20] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*. USENIX Association, Denver, CO, 7 pages.

[21] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[22] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. 2019. Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols. In *European Symposium on Security and Privacy (EuroS&P '19)*. IEEE, New York, NY, USA, 356–370.

[23] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. 2013. On the Security of the TLS Protocol: A Systematic Analysis. In *Annual Cryptology Conference*. Springer, Cham, 429–448.

[24] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *International Conference on Service-Oriented System Engineering*. IEEE, New York, NY, USA, 122–1225.

[25] Thomas Lin, Weiyu Zhao, Ivan Co, Andrew Chen, Henry Xu, and Alberto Leon-Garcia. 2021. PhysarumSM: P2P Service Discovery and Allocation in Dynamic Edge Networks. In *International Symposium on Integrated Network Management*. IFIP/IEEE, Laxenburg, Austria, 304–312.

[26] Guor-Huar Lu and Zhi-Li Zhang. 2007. Wheel of Trust: A Secure Framework for Overlay-Based Services. In *International Conference on Communications*. IEEE, New York, NY, USA, 1148–1153.

[27] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. 2015. How Secure and Quick is QUIC? Provable Security and Performance Analyses. In *Symposium on Security and Privacy (S&P '15)*. IEEE, New York, NY, USA, 214–231.

[28] Petar Maymounkov and David Mazieres. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*. Springer, Cham, 53–65.

[29] David Mazieres and M Frans Kaashoek. 1998. Escaping the Evils of Centralized Control with Self-Certifying Pathnames. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. ACM, New York, NY, USA, 118–125.

[30] Yusef Napora. 2020. *Noise Specification*. libp2p. Retrieved November 10, 2021 from https://github.com/libp2p/specs/tree/master/noise

[31] Esther Palomar, Juan M. Estevez-Tapiador, Julio C. Hernandez-Castro, and Arturo Ribagorda. 2006. A P2P Content Authentication Protocol Based on Byzantine Agreement. In *International Conference on Emerging Trends in Information and Communication Security*. Springer, Cham, 60–72.

[32] Vivek Pathak and Liviu Iftode. 2006. Byzantine Fault Tolerant Public Key Authentication in Peer-to-Peer Systems. *Computer Networks* 50, 4 (2006), 579–596.

[33] Bernd Prünster, Dominik Ziegler, Chrisitan Kollmann, and Bojan Suzic. 2018. A Holistic Approach Towards Peer-to-Peer Security and Why Proof of Work Won't Do. In *International Conference on Security and Privacy in Communication Systems*. Springer, Cham, 122–138.

[34] Nitesh Saxena, Gene Tsudik, and Jeong Hyun Yi. 2007. Threshold Cryptography in P2P and MANETs: The Case of Access Control. *Computer Networks* 51, 12 (2007), 3632–3649.

[35] Mohit Sethi, Aleksi Peltonen, and Tuomas Aura. 2019. Misbinding Attacks on Secure Device Pairing and Bootstrapping. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS '19)*. ACM, New York, NY, USA, 453–464.

[36] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[37] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[38] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2011. A Survey of DHT Security Techniques. *Comput. Surveys* 43, 2 (2011), 1–49.

[39] Dan S. Wallach. 2002. A Survey of Peer-to-Peer Security Issues. In *International Symposium on Software Security*. Springer, Cham, 42–57.

[40] Paul Wouters, Hannes Tschofenig, John Gilmore, Samuel Weiler, and Tero Kivinen. 2014. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). https://doi.org/10.17487/RFC7250