Mechanisms for Mutual Attested Microservice Communication

Kevin Walsh
College of the Holy Cross
Worcester, Massachusetts

John Manferdelli
Northeastern University
Boston, Massachusetts

ABSTRACT

For systems composed of many rapidly-deployed microservices that cross networks and span trust domains, strong authentication between microservices is a prerequisite for overall system trustworthiness. We examine standard authentication mechanisms in this context, and we introduce new comprehensive, automated, and fine-grained mutual authentication mechanisms that rely on attestation, with particular attention to provisioning and managing secrets. Prototype implementations and benchmark results indicate that mutual attestation introduces only modest overheads and can be made to meet or exceed the performance of common but weaker authentication mechanisms in many scenarios.

1 INTRODUCTION

Microservices, small and focused services deployed on independent containers or virtual machines (VMs), are an increasingly common way to structure large systems. For such an ensemble to be trustworthy, it's not enough to secure microservices individually. Communications channels must also be *authenticated.* Authentication—the reliable determination of the endpoints of a channel—is prerequisite for reliable logging, for example, and it provides a basis for authorization or enforcement of other security policies.

Ad-hoc and offline authentication mechanisms are sometimes appropriate, such as when microservices are few in number, mainly static, exist within a single administrative domain, and communicate over an isolated network of fully trusted machines. But increasingly, systems are composed of many microservices that are deployed both rapidly and dynamically, spanning multiple administrative domains and across networks with varying levels of trust. Comprehensive, automated, and fine-grained mechanisms are needed to authenticate microservices in these cases.

Authentication of end-users to cloud services is well-studied (e.g., [10, 12]), but authentication between microservices less so. In practice, when authentication within the cloud is used at all, systems often rely on a patchwork of coarse-grained, ad-hoc mechanisms that require extensive manual intervention [18]. For example, one common practice relies on HTTPS to authenticate one endpoint and HMACs with secret API keys to authenticate the other. This combination is ill-suited for microservices. HTTPS is unnecessarily dependent on a historically vulnerable public key infrastructure (PKI), run by untrustworthy third parties [5, 13], which only weakly authenticate microservices, e.g., via DNS. API key protocols vary widely due to tight coupling with specific REST-oriented protocols, and API key provisioning usually happens manually out-of-band or requires additional authentication mechanisms. Further, due to PKI rate limits and cumbersome API key management, secrets for both are long lived and must be kept in some secure, durable storage service, requiring yet other authentication mechanisms.

Reliance on ad-hoc key management moves the locus of trust from a small number of underlying, cryptographically sound protocols, to a wide variety of weak and unspecified mechanism, and it makes implicit tradeoffs between performance and security. Worse, the system is left unnecessarily vulnerable to insider attacks. The resulting authentication is at best *indirect*—endpoints do not learn true unique identifiers or any actual properties of their peers, but learn only the roles or identifiers an administrator has assigned.

We propose several new mutual authentication mechanisms for microservices using the attestation services provided by Tao [11], a platform we are developing for trustworthy cloud computing. By coupling standard mutual TLS protocols with platform attestations, e.g. from a trusted platform module (TPM) [16], and by deploying per-domain key servers, certificate authorities, and attestation servers, the mechanisms we propose implement a form of *direct authentication*, with endpoints securely exchanging globally unique and semantically meaningful identifiers in the form of platform attestations. In the next section, we describe these mechanisms in detail, with specific attention to how all necessary secrets are provisioned and managed. In Section 3, we describe prototype implementations and compare performance characteristics of all the mechanisms. We discuss related work and conclude in Section 4.

2 AUTHENTICATION AND ATTESTATION

We consider the problem of mutual authentication between two principals: A, acting as a client; and B, acting as a server. These might be individual processes, perhaps executing within a container, on an operating system (OS) that itself executes on a VM or directly on hardware. More generally, a principal could be smaller or larger than a single process, e.g., a single thread, or a service account executing many processes across a cluster. The underlying system is responsible for providing sufficient isolation between distrusting principals. To ensure that an authentication mechanism does not merely move the fundamental problem of assurance elsewhere, e.g. to some unspecified key distribution or storage mechanism, we explicitly account for the generation and provisioning of any secrets held by A, B, and other participants.

The mechanisms described below rely on TLS as an underlying transport protocol, with either asymmetric or shared symmetric keys. This dependence is not fundamental, but serves to clarify our focus on the distribution and maintenance of secrets and the implicit and explicit trust assumptions needed by different mechanisms.

UCC'17 Companion, December 5–8, 2017, Austin, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5195-9/17/12…$15.00
https://doi.org/10.1145/3147234.3148102

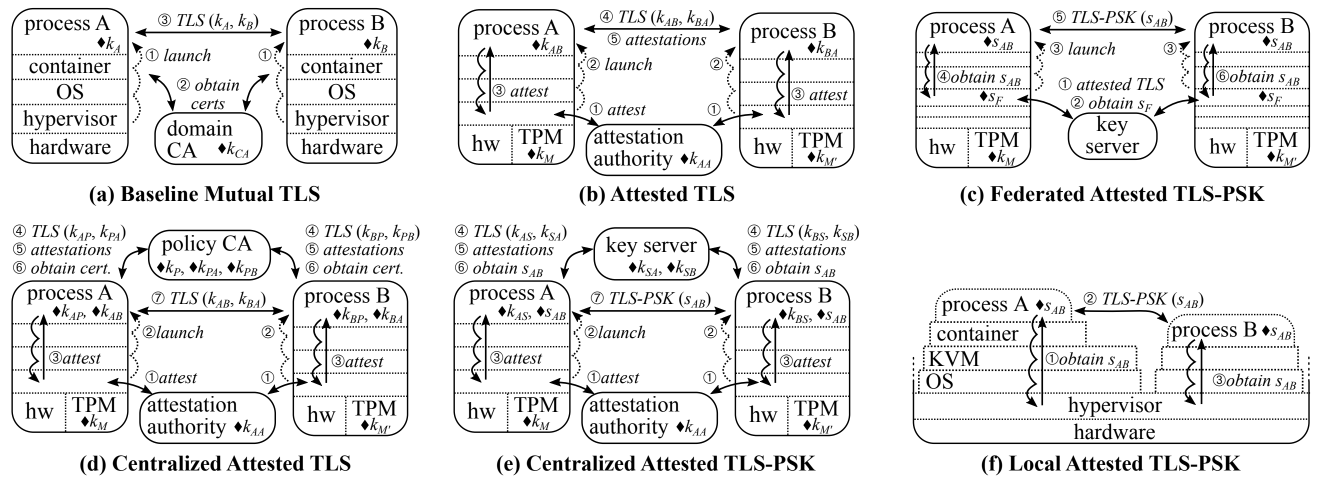

Figure 1: Mutual authentication mechanisms using TLS and attestation, with components holding secrets (♦) as indicated.

2.1 Baseline Mutual TLS

As a baseline, consider TLS with ephemeral keys and certificates, as illustrated in Figure 1 (a). At deployment, A and B each independently generate an asymmetric key, k_A and k_B, then obtain signed X.509 certificate chains for these keys from a domain-specific certificate authority (CA) over an unauthenticated TCP channel. A and B next perform a TLS handshake, exchanging certificates for mutual authentication. Each microservice holds its key for the lifetime of the instance and uses it for all communication, but the keys are not persistent. Instead, fresh keys are created on each deployment.

This baseline mechanism is widely used, though it provides no explicit authentication between microservices and the CA—in practice, manual or out-of-band mechanisms would be used to ensure the CA only issues certificates for the keys chosen by A and B, or conversely, that only A and B have access to keys associated with issued certificates. The latter approach is essentially that of Istio [9], a new and widely publicized container and microservice orchestration framework, except that Istio provisions only long-lived keys, and it does so only at a coarse, service-account granularity, rather than for individual microservice instances.

Baseline mutual TLS has significant advantages over HTTPS and API keys. First, endpoints need not store long-term secrets. And second, it is application-agnostic, in the sense that this single mechanism can accommodate a variety of application-layer protocols on top of it. However, it still requires the CA be capable of storing a long-lived private key. Further, it provides only indirect authentication, since the semantics of names contained within certificates is entirely outside the scope of the mechanism. Instead, naming depends on the unspecified authentication between endpoints and the CA and on how keys are generated by (or distributed to) endpoints.

2.2 Attested TLS

To provide for direct authentication, microservices can be executed on platforms capable of remote attestation, e.g., with a trusted platform module (TPM) [16]. Much like a certificate from a CA, remote attestation binds a public key to a name. But names in attestations are computed from properties of the principal holding the private key and are usually globally unique and have well-defined semantics. As such, authorization and other security policies can be enforced directly based on information conveyed by attestations. By contrast, names carried in certificates are rarely unique and often represent the outcome of policy decisions, e.g., that a principal is authorized to act on behalf of some named service account or role.

Figure 1 (b) outlines one way to leverage attestation for authentication. Here, embedded within TPMs are static keys, k_M and $k_{M'}$. A central cloud attestation authority uses a separate key, k_{AA}, to issue a signed attestation for k_M and $k_{M'}$ detailing properties of the corresponding hardware platform. As for the CA key used with baseline mutual TLS, the attestation authority key is long-lived and must be known to all participants. To establish a channel to B, A generates an ephemeral key, k_{AB}, and makes an attestation request for that key to the underlying container or OS. In the simplest case, this request is passed down the software stack, eventually to the TPM, which signs an attestation for k_{AB} containing platform details gathered at each layer of the software stack, including a hash of all boot and hypervisor code and configuration, along with VM image, OS kernel, container, and process code and configuration hashes. B similarly generates k_{BA} and obtains a matching attestation from its platform. A and B use their keys to establish a TLS channel. Then, to achieve direct authentication, they each exchange: their TPM attestations, bound to their TLS keys and signed by platform keys K_M and $K_{M'}$; and the central authority attestations, bound to platform keys and signed by K_{AA}.

In the mechanism outlined above, microservices use different keys for different peers, but reuse keys for connections to the same peer. Alternatively, a microservice could be configured to generate new ephemeral keys and attestations at any time, e.g. to rotate keys or to account for changes in software or platform state. Note that, in all cases, the choice of behavior is reflected in the attestations.

Generating attestations with a TPM is prohibitively expensive. The cost of generating lower-layer attestations can be amortized over multiple higher-layer instances by having intermediate layers of the software stack generate a key and obtain an attestation for it from lower layers, then use that key for signing later attestation requests from layers above. For example, in Tao the TPM only signs one attestation for each OS and VM, and the OS signs attestations for the containers and processes it hosts. The resulting attestation chain is still rooted in the central cloud attestation authority.

Microservices using attested TLS can make authorization decisions directly based on attestation chains exchanged during channel setup. Such direct authentication provides more opportunities than traditional certificates for enforcing non-trivial policies, since endpoints have full visibility into the platform details of their peers and complete control of policy enforcement. A microservice could reject a connection if the peer's specific attestation details do not appear on an access control list. Or, it could evaluate a custom predicate on peer attestations to determine what level of access to grant.

2.3 Centralized Attested TLS

Placing enforcement at the endpoints of communications channels has advantages but is not always appropriate. For example, relying on distributed policy enforcement with attested TLS can be brittle, since attestations encode more information than is necessarily relevant for any particular policy, and attestation details may change in difficult to predict ways. Similarly, centralized enforcement can more easily facilitate auditing, dynamic policies, and revocation.

Figure 1 (d) illustrates a mechanism incorporating a central policy authority that acts as a CA, issuing certificates for microservice ephemeral keys. Unlike baseline mutual TLS, authentication between the microservices and the policy CA is explicitly authenticated using attested TLS. In effect, the policy CA is used to trade attestations chains for a simpler policy-specific certificates. When a policy CA is used by a microservice to obtain certificates for many ephemeral keys, there is an opportunity to amortize costs by reusing keys and attestations when re-authenticating to the policy CA. As before, the policy CA key is long-lived and well known.

Centralization adds latency during connection setup, to obtain certificates from the policy CA. But it also reduces verification costs for endpoints compared to attested TLS, because microservices need not exchange or verify attestation chains. Also note that only the policy CA needs to know which cloud attestation authorities should be trusted, rather than provisioning each endpoint with this information, as would be required for attested TLS. However, the policy CA can be a single point of failure and a potential bottleneck, as it communicates with every microservice.

2.4 TLS Session Resumption

Generating and using short-lived asymmetric keys is computationally expensive, and their size brings transmission overheads. Using session tokens, TLS session resumption amortizes costs due to asymmetric keys across multiple connections between two endpoints, with the handshaking for the second and subsequent connections using only symmetric cryptography. Enabling session resumption may make the initial handshake slightly more costly, however, as the endpoints must generate and exchange a session token.

Session resumption is compatible with all of the mechanisms described thus far. For centralized attested TLS, session resumption can be used between microservices and also between a microservice and policy CA. False start and fast open, available in TLS 1.3, could be used as well and would likely provide similar benefits.

2.5 Centralized Attested TLS-PSK

Where TLS session resumption is not appropriate, TLS pre-shared key (TLS-PSK) mode can eliminate costs associated with asymmetric keys and certificates, relying on shared symmetric keys instead. The main challenge for microservices is in automatically and securely distributing shared keys only to the appropriate endpoints.

Figure 1 (e) illustrates a mechanism using a central key server that generates and distributes shared symmetric keys, on demand, to microservices over attested TLS channels. First, A creates an attested TLS channel to the key server and sends it a hash, H_B, of B's attestation details. The key server obtains A's verified attestation details from the TLS channel and locally computes their hash, H_A. The key server then generates a random secret key, s_{AB}, and associates it with a *tag* consisting of the unordered set $\{H_A, H_B\}$. Key s_{AB} is returned to A, which initiates a TLS-PSK handshake with B. B similarly sends H_A to the key server using attested TLS, which looks up and returns the same key, s_{AB}, previously associated with tag $\{H_A, H_B\}$. B now completes its handshake with A. This mutually and directly authenticates A and B, since their handshake succeeds only if both use the same shared key, which only occurs if both invoke the key server with matching attestation details.

As with central attested TLS, the key server here can centrally enforce policies, e.g. by issuing keys only for certain tags and thereby allowing only certain microservices to mutually authenticate.

For n microservices, the key server may need to manage $O(n^2)$ keys—one for each of pair of microservices—or more if keys are rotated. One strategy is to generate a random key upon first encountering a tag, storing the key for only a short time. Alternatively, the key server can derive keys based on the tags and a long-lived master key. This requires maintaining a long-lived secret, but eliminates the need to hold shared secrets and allows microservices to obtain keys proactively, well before initiating a connection to a peer.

It is natural to consider distributing the work of the key server to address the potential bottleneck it introduces. In fact, several novel schemes are described in the literature for distributed generation of pairwise shared secrets among a set of participants (e.g. [2, 4]). We do not evaluate such schemes as they rely on non-standard cryptographic primitives or assumptions.

2.6 Local Attested TLS-PSK

When microservices are co-located on a common platform, reliance on remote third-parties can be avoided. Instead, we can leverage the local platform for attestation, with some common, shared layer of the software stack acting in place of a central key server. The platform and shared layers of the software stack are typically fully trusted already, as they underlie both channel endpoints, so relying on them for mutual attestation adds no additional trust assumptions. It may also be more efficient than using remote servers.

With local attested TLS-PSK, shown in Figure 1 (f), A and B each invoke a parent software layer to request a shared key. These requests propagate down the software stack until they reach a common ancestor, such as a shared hypervisor. Each request carries a hash of its peer's attestation details, and each layer adds attestation details for the principal making the request. The common ancestor can then create the appropriate tag and secret s_{AB}, then return s_{AB} up the stack so A and B can complete their TLS-PSK handshake.

Heterogeneous software stacks present a challenge for local attested PSK. If microservices are simple processes on a common OS, each can simply request a shared key using a system call. More typically, microservices execute in different containers, on different VMs, or even on nested VMs. Discovering which layers might serve as a common ancestor for an arbitrary pair of microservices may not be trivial, and all intervening layers must implement a common API to pass requests, attestations, and keys between layers.

2.7 Federated Attested TLS-PSK

For microservices hosted on a set of federated, mutually trusting platforms, the platforms together can act as a distributed key server, as shown in Figure 1 (c). The goal of federation is for some common software layer on each platform to establish a shared master secret with corresponding layers on all other platforms. Each can then use this shared secret to generate TLS-PSK keys for microservices hosted above that layer. Federation can be implemented easily using essentially the same central key server as for centralized attested TLS-PSK, but here the secrets are distributed to lower layers, rather than microservices, and used as master key-generating secrets rather than TLS-PSK keys. Here, tags contain hashes of attestation details for platforms and layers, rather than for microservice endpoints. The cost of using a central key server is amortized across all microservices on each platform.

One drawback to federation is the potential management complexity of establishing federation at the appropriate layers of the software stack. In our prototype implementation, described in the next section, each software layer can federate with only one group of peer platforms, and the configuration is managed manually.

3 IMPLEMENTATION AND PERFORMANCE

We implemented prototypes for all the mechanisms described above and used micro-benchmarks to compare costs. The implementations are freely available, as is the Tao platform upon which they build[1]. Benchmark microservices were implemented as individual processes, running variously within Docker containers, on CoreOS, on KVM virtual machines, and on Debian GNU/Linux 9 instances on either a hypervisor in the Google Compute Engine (GCE) cloud or on dedicated TPM-capable hardware.

All mechanisms were implemented in Go with TLS 1.2, using third-party[2] TLS-PSK support. Elliptic-Curve Diffie-Hellman key exchange is used, with NIST P-256 or 256-bit TLS-PSK keys, 128-bit AES-GCM, and SHA256. While we have attempted to avoid implementation vulnerabilities, the prototypes have not yet undergone formal security evaluation; they are intended to investigate mechanism performance rather than serve as a basis for deployment.

3.1 Tao APIs for Attested Authentication

Tao was used to augment each layer of the software stack—Docker, CoreOS, Linux, and KVM—to provide trustworthy computing services, i.e., generating shared keys and attesting to software hosted by each layer. Here, we sketch core features of Tao and describe extensions we made to implement attested authentication mechanisms. Further details on Tao can be found in prior work [17].

[1]https://github.com/kevinawalsh/taoca/ and https://github.com/jlmucb/cloudproxy/
[2]https://github.com/mordyovits/golang-crypto-tls/

The first Tao API needed for attested authentication is invoked by microservices to obtain an *n*-byte secret key for TLS-PSK.

$$GetKey(tag, i, n) \rightarrow [\,]bytes$$

Here, *tag* is a set of Tao *principal names*, each a globally unique, hierarchically-structured description of the software stack implementing some principal. These are the principals that are intended to share the resulting secret, i.e., the microservices at the endpoints of a TLS-PSK channel. In simple cases, the Tao layer underlying the endpoint first checks that the caller's attestation details match one of those in *tag*. Tao then generates and returns $s_{(tag,i)}$, a secret of *n* bytes that depends on both *tag* and *i*. Crucially, the same secret will be returned if *GetKey*() is later invoked with the same parameters from other microservices, but only if those later callers are in *tag*. Parameter *i* is an arbitrary identifier to allow key rotation.

To support local microservices on heterogeneous software stacks, Tao examines all principal names in *tag*. If the principal name representing the Tao layer itself is a prefix of those names, then the Tao layer is a common ancestor, and $s_{(tag,i)}$ is derived using an ephemeral master secret chosen at random during instantiation of that Tao-enabled layer. Otherwise, the Tao layer invokes *GetKey*() on the next lower layer of the software stack. Ultimately, the request will reach a suitable ancestor of the principals in *tag*, if one exists.

We extend *GetKey*() to support federation by having some Tao layers open attested TLS channels with a central key server to obtain a master secret. This shared secret is used to derive $s_{(tag,i)}$ when the names in *tag* have no common prefix, i.e. when microservices use federated attested TLS-PSK from distinct platforms.

A second Tao API is used as part of our implementation of attested TLS and all mechanisms that rely on it.

$$Attest(stmt) \rightarrow att$$

Encoded in *stmt* is an assertion in a formal logic defined by Tao, and *att* is a signed message conveying, essentially, *P says stmt*, where *P* is the caller's principal name. For attested TLS, after creating an ephemeral TLS key, each endpoint invokes *Attest*() with *stmt* containing a hash of the public half of its ephemeral key, thereby linking that key to the Tao principal name for that endpoint. Depending on the Tao layers involved, *att* usually contains a chain of signatures with each layer attesting to details of the next: the TPM attests to hypervisor details, the hypervisor attests to details of the OS, etc. Because public clouds do not generally provide access to a hardware TPM, we configured bottom-most Tao layers in GCE benchmarks to use software-based attestation instead.

3.2 Channel Setup Latency

Latency for setting up communications channels is important for short-lived and rapidly deployed microservices. We measured latency for channel setup and a small, 32-byte message exchange between microservices executing on independent VMs in the same GCE zone. Central servers were deployed to the same GCE zone. A client for each mechanism was instrumented to measure latency in three scenarios, in sequence: (a) an initial connection and exchange with one peer; (b) a connection and exchange with a second peer; and (c) a final re-connection and exchange again with the first peer. Experiments for all scenarios were performed both with, and without, TLS session resumption.

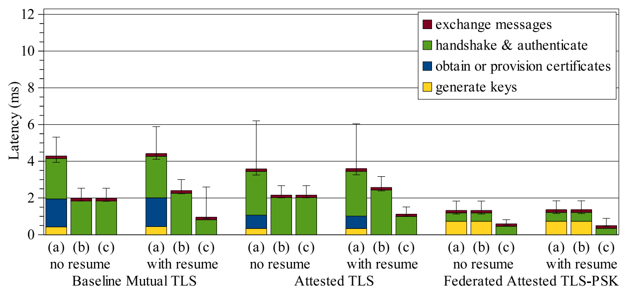

Figure 2: Channel latency for non-centralized mechanisms. Bars are medians, whiskers show 5th and 95th percentiles.

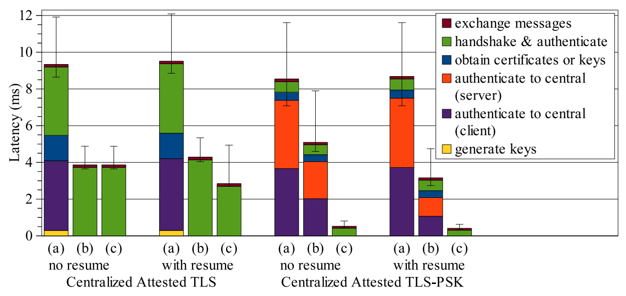

Figure 3: Channel latency for centralized mechanisms. Bars are medians, whiskers show 5th and 95th percentiles.

We report the median of at least 10,000 trials for each experiment, with variance indicated by 5th and 95th percentiles. Artifacts from VM migration and other factors caused high, unpredictable latency in fewer than 1% of trials. As a reference and lower-bound, in GCE zones where tests were performed, channel setup and message exchange using unauthenticated TCP required 0.18 ms, split evenly between TCP handshake and message exchange.

Figure 2 shows a breakdown of latency for non-centralized mechanisms, revealing that attested TLS without resumption is 16% faster than baseline mutual TLS on the initial connection, down 0.7 ms from 4.25 ms, but it also imposes a 9% penalty on subsequent connections. This reflects the fact that using *Attest*() locally is faster than obtaining a certificate from a remote CA, even over an unauthenticated TCP channel, but handshaking for attested TLS is slower, as it adds an extra round trip to exchange attestations and peers must also verify the signature chains within those attestations. We exclude costs for servers to generate keys and obtain certificates or attestations, as this is done well before or concurrently with clients.

Baseline and attested TLS benefit similarly from session resumption in scenario (c). Session resumption does impose a small but measurable cost in scenarios (a) and (b), as the server must generate a cryptographic token during the initial handshake.

Federated attested TLS-PSK has much lower latency than either baseline or attested TLS in all scenarios because of its reduced handshake, using only symmetric keys and without need to obtain or exchange certificates or attestations. The latency for generating keys is increased—0.7 ms, versus 0.4 ms for baseline and attested TLS—because the keys for TLS-PSK are generated by invoking *GetKey*() on the underlying Tao over an IPC or system call channel, rather than locally within each microservice process. Session resumption with TLS-PSK shows only marginal effects, as the handshakes are quite similar. These results do not include costs for underlying Tao layers to obtain shared keys, which itself uses centralized attested TLS, since this occurs well before microservice instantiation and the cost is amortized over multiple microservices.

Figure 3 shows results for the two centralized mechanisms. Overall, these are roughly twice as slow as their non-centralized counterparts, due mainly to the cost of establishing an initial connection to the central policy CA or key server. Both client and server use attested TLS for this, incurring that mechanism's full cost in scenario (a) (we only include client costs here). In centralized attested TLS, the policy CA is contacted only once. For centralized attested TLS-PSK, the key server is contacted by a microservice once for each destination endpoint, but here session resumption reduces the cost of authenticating to the CA, and even without session resumption the same keys and attestations can be reused.

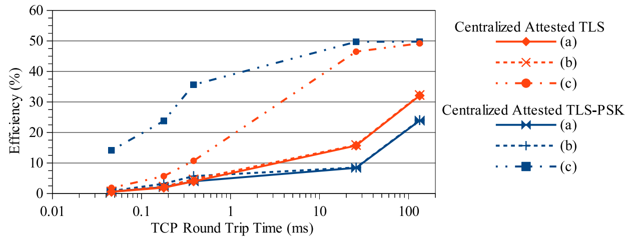

Figure 4: Mechanism efficiency (median), relative to TCP.

Centralized attested TLS-PSK is faster than its non-PSK counterpart in most scenarios. Here, we include the cost for obtaining shared keys for both client and server, because these keys are obtained as part of the TLS handshake and are tied to specific endpoint pairs. Still, the efficiency gains from TLS-PSK more than compensate for the extra cost of obtaining multiple keys from the central server. When centralized attested TLS-PSK is used to connect repeatedly to the same peer, in scenario (c), latency reduces to that of just TLS-PSK itself. This is significantly faster than any other mechanism or scenario we tested.

3.3 Sensitivity to Network Latency

To explore the extent to which the mechanisms are sensitive to network conditions, relative to fixed computational costs, we performed benchmarks with all participants on the same VM, in the same GCE zone, in disjoint GCE zones within a single region, and spread across a variety of GCE regions. Figure 4 shows how mechanism *efficiency* varies over this range of network conditions, where efficiency is defined as the time taken to perform mutually attested channel setup and message exchange relative to using unauthenticated TCP. In scenarios (a) and (b), computational costs for keys and signatures lead to relative insensitivity to network latency. But in scenario (c) much of the computational costs are avoided, resulting in higher network sensitivity. Peak efficiency is achieved when network latency is high: at 133 ms network latency, both mechanisms in scenario (c) achieve nearly 50% efficiency, meaning their latency is only about twice that of TCP.

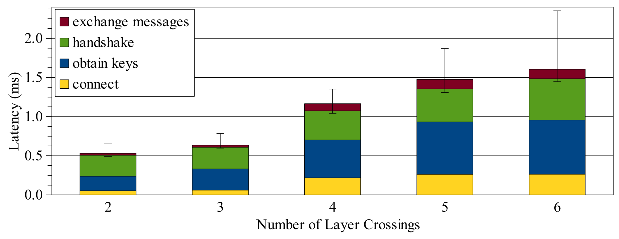

Figure 5: Channel latency for local attested TLS-PSK. Bars are medians, whiskers show 5th and 95th percentiles.

3.4 Heterogeneous Software Stacks

Local attested TLS-PSK is restricted to microservices co-located on the same platform, so it is unaffected by network conditions. Instead, performance depends on how many layers are crossed when using Tao APIs and TCP. For example, communication may involve channels from a client microservice, down to a Docker container, to CoreOS on KVM, to an underlying Linux OS, then back up to a different container, and finally up to a server microservice.

We measured latency for local attested TLS-PSK channel setup and message exchange between microservices separated by a varying number of software layers. The results, shown in Figure 5, reveal that as the number of layers increases, the cost to obtain keys grows more rapidly than other phases of channel setup. This is likely because Tao-related inter-platform communication is not as efficient or optimized as cross-layer TCP, unsurprising given the importance of TCP for existing Docker and KVM-hosted microservices. Still, local attested TLS-PSK can complete a message exchange, with full mutual authentication and attestation across six layers, in well under 2 ms, and only 43% of this cost is due to attestation.

4 RELATED WORK AND CONCLUSIONS

The attested mechanisms we describe are built on top of standard TLS channels and certificates. Prior work (e.g. [1, 7, 15]) examines the feasibility of modifying TLS to directly integrate TPM-based endpoint attestation. Stumpf [15] analyzes many such protocols.

Gasmi et al. [6] link TPM attestations to TLS channels, but further isolate keys from endpoints to enable dynamic attestation, in which channel state can be updated or revoked without endpoint participation, e.g., in case of compromise. By contrast, microservices using our attested mechanisms have direct access to key material and must participate in revocation. Alternatively, TLS and per-endpoint keys could be eliminated entirely by arranging for lower layers of the software stack to provide directly-attested, authenticated channels—perhaps federated across multiple platforms—much as an OS provides built-in IPC facilities.

We use Tao for attestation and secret key generation, and Tao in turn relies on a hardware TPM where available. Thus far, public clouds have been slow to provide APIs for hardware-based attestation, though Google's recently-announced Titan chip [8] may eventually enable such capabilities. Coker et al. [3] discuss a variety of hardware and software remote attestation primitives, including timing-based techniques, that may apply to microservices in the cloud regardless of cloud provider support.

The centralized attested TLS mechanism described here closely mirrors the scheme used in keylime [14], which also relies on TPM attestations, a central cloud CA for platforms, and a policy CA for tenant applications. That work did not consider TLS session resumption or mechanisms using TLS-PSK, however.

All of the attested communications mechanisms we describe would provide tangible trustworthiness benefits for systems composed of multiple short-lived or rapidly deployed microservices. Benchmarks results reveal tradeoffs between policy centralization, in the form of key servers and policy CAs, and latency for key provisioning and handshaking. We find that for all mechanisms, the computational and network overheads involved are modest for many typical scenarios and, in many scenarios, performance of some attested mechanisms can meet or exceed that of a common, but only weakly-authenticated, baseline TLS configuration.

ACKNOWLEDGMENTS

Tom Roeder has provided invaluable help implementing authentication mechanism prototypes and related Tao extensions.

REFERENCES

[1] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. 2008. An Efficient Implementation of Trusted Channels Based on OpenSSL. In *Proc. ACM Workshop Scalable Trusted Comput.*
[2] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Proc. Annu. Netw. and Distrib. Syst. Security Symp.*
[3] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Hanlon O'Brian, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of Remote Attestation. *International Journal of Information Security* 10, 2 (2011), 63–81.
[4] Wenliang Du, Jing Deng, Yunghsiang S. Han, Pramod K. Varshney, Jonathan Katz, and Aram Khalili. 2005. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. *ACM Trans. Inf. and Syst. Security* 8, 2 (2005), 228–258.
[5] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *Proc. Internet Measurement Conf.*
[6] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. 2007. Beyond secure channels. In *Proc. ACM Workshop Scalable Trusted Comput.*
[7] Kenneth Goldman, Ronald Perez, and Reiner Sailer. 2006. Linking remote attestation to secure tunnel endpoints. In *Proc. ACM Workshop Secure Web Services.*
[8] Urs Hölzle. 2017. Bolstering security across Google Cloud. https://www.blog.google/topics/google-cloud/bolstering-security-across-google-cloud/. (March 2017).
[9] Istio 2017. https://istio.io/. (June 2017).
[10] Hongqian Karen Lu. 2014. Keeping Your API Keys in a Safe. In *Proc. IEEE Int. Conf. Cloud Comput.*
[11] John Manferdelli, Tom Roeder, and Fred Schneider. 2013. *The CloudProxy Tao for Trusted Computing*. Technical Report UCB/EECS-2013-135. University of Califronia at Berkeley.
[12] Jisoo Oh, Jaemin Park, Sungjin Park, and Jong-Jin Won. 2016. TAaaS: Trustworthy Authentication as a Service. In *Proc. IEEE Int. Conf. Cloud Comput.*
[13] Steven B. Roosa and Stephen Schultze. 2013. Trust Darknet: Control and Compromise in the Internet's Certificate Authority Model. *IEEE Internet Comput.* 17, 3 (Feb. 2013), 18–25.
[14] Nabil Schear, Patrick T. Cable II, Thomas M. Moyer, Bryan Richard, and Robert Rudd. 2016. Bootstrapping and Maintaining Trust in the Cloud. In *Proc. Annu. Comput. Security Appl. Conf.*
[15] Frederic Stumpf. 2010. *Leveraging attestation techniques for trust establishment in distributed systems.* Ph.D. Dissertation. Technische Universität Darmstadt.
[16] Trusted Computing Group. 2011. Trusted Platform Module (TPM) Specification, version 1.2. https://www.trustedcomputinggroup.org/specs/TPM/. (1 March 2011).
[17] Kevin Walsh. 2016. TLS with Trustworthy Certificate Authorities. In *Proc. IEEE Conf. Commun. and Netw. Security.*
[18] Kevin Walsh and John Manferdelli. 2017. Intra-Cloud and Inter-Cloud Authentication. In *Proc. IEEE Int. Conf. Cloud Comput.*