RATLS: Integrating Transport Layer Security with Remote Attestation

Robert Walther[1], Carsten Weinhold[2], and Michael Roitzsch[2]

[1] Technische Universität Dresden, Dresden, Germany
robert.walther@mailbox.tu-dresden.de
[2] Barkhausen Institut, Dresden, Germany
{firstname.lastname}@barkhauseninstitut.org

Abstract. We present RATLS, a companion library for OpenSSL that integrates the Trusted Computing concept of Remote Attestation into Transport Layer Security (TLS). RATLS builds upon handshake extensions that are specified in version 1.3 of the TLS standard. It therefore does not require any changes to the TLS protocol or the OpenSSL library, which offers a suitable API for handshake extensions. RATLS supports remote attestation as part of a complete TLS handshake for new connections and it augments session resumption by binding session tickets to the platform state of TLS peers. We demonstrate that RATLS enables both client and server to attest their respective software stacks using widely-used Trusted Platform Modules. Our evaluation shows that the number of round trips during handshake is the same as for traditional TLS and that session resumption can reduce cryptographic overhead caused by remote attestation for frequently communicating peers.

Keywords: TLS · TPM · Remote Attestation · Trusted Computing

1 Introduction

Transport Layer Security (TLS) [10] is the state-of-the-art protocol for securing communication channels between two computers. It uses encryption and message authentication codes (MACs) to ensure confidentiality and integrity for all information that is transmitted over the communication channel. TLS also provides authentication to ensure that only the "right" communication partners can successfully establish a TLS connection. The authentication method used by TLS requires users to trust that the party who operates the remote computer acting as a TLS peer will keep this computer secure. Typically, if Alice wants to exchange data over TLS with a computer operated by Bob, she has to make two assumptions: 1) Bob keeps the cryptographic keys needed for TLS authentication secret, and 2) the software running on Bob's computer does what he claims it does (e.g., not leak data received from Alice).

The Need for Verifiable Trust Unfortunately, TLS on its own cannot provide a verifiable proof that assumptions 1) and 2) actually hold. In certain highly-critical use cases, such a proof is desirable, though. For example, Alice might

want an assurance that her valuable scientific data will only be processed by a certain, trusted analysis program running on the cloud server that Bob rented to her. And in an Internet-of-Things (IoT) scenario, lives might be at stake if an attacker manages to manipulate the firmware of an IoT device. The risk that TLS keys (assumption 1) and software integrity (assumption 2) are compromised are much greater for a connected device that must be installed in a public place, compared to a server behind the walls of a guarded data center. Thus, technical measures are needed to reduce the trust in the operator of a remote computer or the environment that surrounds it.

Trusted Computing *Remote Attestation* is a cryptographic protocol that can complement TLS by solving the two trust problems described above. First, it is built on top of hardware support that is designed to protect cryptographic secrets. Second, it provides one computer, the *challenger*, with a verifiable proof that software running on another computer, the *attester*, is in a known-good state. It works as follows:

1. **Identifiability:** The attester has a root of trust integrated into its hardware that includes a cryptographic identity that cannot be forged. Through this identity, the challenger can know what the attester device is and what its capabilities are.
2. **Integrity:** The root of trust can create a digital signature over the code of the software that has been started on the attester. Through this signature, the challenger can know, if the software currently running on the attester will be behave as required. Typically, the signature covers both system-level code and applications, including the TLS protocol implementation that protects the communication channel.

Roots of trust are much harder to compromise than pure software solutions because the attacker has to manipulate tamper-resistant hardware. They are available in various forms for many different platforms. A well-known implementation is the Trusted Platform Module (TPM) [4], which is nowadays built into most desktops, laptops, and many servers.

TLS with Remote Attestation Despite the clear security advantages, Remote Attestation is complicated to deploy for application developers. No standardized protocol suite exists, but different root-of-trust implementations come with their own protocol and software development kit. RATLS intends to simplify the use of attestation by integrating it into the widely-used TLS protocol. The integration leverages a feature of the TLS v1.3 standard [10], where applications can append user-defined extensions to TLS handshake messages. Using this mechanism, RATLS is able to piggyback attestation-related messages onto TLS handshake messages. The RATLS approach can be applied to any TLS implementation that supports handshake extensions. We built RATLS as a companion library for the widely-used OpenSSL [2] implementation, which provides a suitable extensions API. The design of RATLS is also agnostic to the underlying hardware root of trust. In this paper, we describe an RATLS plugin that works with the widely-used TPM v2.0.

Contribution Our work on RATLS makes the following contributions:

- RATLS integrates remote attestation into TLS, thereby enabling verification of the identity and software integrity of peers communicating via TLS.
- RATLS does not require changes to the TLS protocol nor modifications to OpenSSL, thereby demonstrating that the approach is non-invasive.
- RATLS supports both full handshakes for new connections and TLS session resumption for efficient reconnects.
- We evaluate API usability, security properties, and performance of an RATLS prototype implementation using TPM v2.0-based roots of trust.

In the following Section 2, we describe relevant background. Sections 3 and 4 present the design and implementation, respectively, and in Section 5, we evaluate RATLS. We discuss related work in Section 6 before concluding paper in the Section 7.

2 Background

In this section, we describe the basics of TLS. We highlight those features of the standard that are important for RATLS. Furthermore, we give an overview over trusted computing concepts that enable remote attestation and secure storage.

2.1 Transport Layer Security

TLS enables two computers to communicate securely over an untrusted network. The protocol is based on cryptography and it guarantees confidentiality and integrity for all user data that is transmitted through the communication channel. Two computers that wish to communicate over TLS must authenticate themselves to each other. In the most common scenario, one peer, the *server*, proves its identity through possession of the private key of an asymmetric signature key pair. The other communication partner, the *client*, can validate the identity of the server using a publicly-known *server certificate* that contains the public part of said key pair. Optionally, the client can also authenticate itself to the server using its own private key and a corresponding *client certificate*. In so-called zero-trust communication scenarios using *mutual TLS (mTLS)*, certificate-based authentication is mandatory for both peers.

Security Assumptions For all these variants of TLS, the attacker model assumes that the private key used by a TLS peer is exclusively known to the party that operates this peer. For example, if Bob operates a TLS-protected server, then only Bob shall know the private key used by his server. Likewise, only Alice shall be in possession of the private key used by her TLS client. A computer operated by a third party like Eve can impersonate neither Alice's nor Bob's machines, because she does not know their respective private keys. The encryption and signature algorithms as well as the hash-based MAC schemes in TLS v1.3 are state of the art and considered secure.

TLS Handshake and Extensions One TLS peer, the client, initiates the secure communication by sending a `ClientHello`. The server replies with a `ServerHello` message, which the client acknowledges in a third message. As part of this three-phase handshake, the client and server present each other their certificates and they negotiate the symmetric *session keys* for encrypting and integrity-protecting the payload data that is transmitted after the channel has been established. In TLS v1.3, an application program can request the TLS implementation to append user-defined extensions to certain TLS messages, including handshake messages[3]. Extensions can contain up 64 KiB of arbitrary data. They must follow a request–response scheme, where a specific extension can only be appended to a reply message, if the previous message also contained an extension of the same user-defined type.

TLS Session Resumption Creating and validating signatures and exchanging session keys incurs computational overhead. Furthermore, during the three-phase TLS handshake, both client and server must wait for replies to arrive over a potentially slow network, before they can continue with the protocol. To speed up connection establishment between frequently communicating peers, TLS v1.3 supports *session resumption*. Using this optimization, a client can reopen a previously closed TLS session by sending a `ClientHello` message with a *session ticket* that the server issued to the client while the original connection was active. If the server recognizes the session ticket, only one network round trip is needed instead of two round trips for the complete handshake. With session resumption, both client and server skip the certificate exchange and key negotiation, as they can reuse the session keys from the original connection.

2.2 Trusted Computing

The core idea behind *trusted computing* is to verify through technical means that a computer and the software running on it conform to certain security properties. This verification can be performed remotely from a second, already trusted device in the case of *remote attestation*. But given the right hardware and system-software support, trusted-computing concepts can also be used to verify software that is running locally, like in the case of *sealed memory*.

Remote Attestation In the first use case, a trusted *challenger* device requests an *attestation report* from a remote computer called the *attester*. Like TLS, the underlying cryptographic protocol uses the private part of a signature key pair[4] to prove its identity. However, in contrast to TLS, which is pure software, an implementation of remote attestation typically requires a hardware root of trust that is integrated into the hardware of the attester device.[5] This root of trust

[3] TLS v1.2 supports extensions, too, but on fewer message types than TLS v1.3.

[4] Some implementations use symmetric keys or a physically unclonable function (PUF) instead, but the general concept is the same.

[5] There are implementations of remote attestation that are software only, but they assume a weaker attacker model.

hides the private key in hardware to make it more difficult to forge. The root of trust signs the executable code that is currently in control of the attester. If the challenger recognizes that the signature has been created by a root of trust that it deems trustworthy, it will know what kind of device the attester is and what software is running on it. Based on the information in the attestation report (e.g., a hash of the executable code on the attester), the challenger can then decide whether to trust the attester's software for the purpose it is interested in.

Sealed Memory An attestation can also be done locally by the root of trust on the attester device. Some trusted-computing platforms use local attestation to protect confidentiality of user data by encrypting it with a storage key that is also hidden in the hardware root of trust. The root of trust will only release or *unseal* the plaintext copy of the data to the currently running software, if the identity of this software is the one that has been specified as the "owner" when the data had been *sealed*. Thus, it is possible to *bind* (i.e., restrict access to) user data to a specific, *authorized* software configuration.

Trusted Platform Modules A widely-used and thoroughly standardized root of trust implementation is the Trusted Platform Module (TPM) [4]. TPMs can create remote attestation reports (called *quotes*) and they support sealed memory. Quotes are computed over a set of *Platform Configuration Registers (PCRs)*, which store hashes of the software stack that has been started (ranging from firmware over bootloader and OS to application programs). Like all roots of trust, TPMs require operating-system support in terms of a device driver and other system-level integration. This support includes the so-called *TPM Software Stack (TSS)* [3] through which applications can interact with the TPM.

In the following, we will refer to the combination of root of trust and its system-level support software as the *Attestation Provider*. We will also use *Quote* as a synonym for "attestation report", as this term is commonly used for many root-of-trust implementations. RATLS integrates remote attestation into the TLS handshake and it uses sealed memory to bind TLS session tickets to the software configuration that was valid at the time of the initial handshake. Sealed memory can also be used to protect TLS private keys, in addition to the identity keys of the root of trust that are already hidden in hardware. Our prototype implementation is based on a TPM v2.0-based attestation provider.

3 Design

TLS already provides confidentiality and integrity for all data sent through the communication channel. Our main goal in improving it is twofold: 1) to enhance authentication of TLS endpoints through additional identity checking, and 2) to provide technical means for verifying the software integrity on these endpoints. Thus, our aim for RATLS is to augment TLS with remote attestation such that it provides additional security guarantees.

3.1 Design Goals

In the following paragraphs, we define security goals as well as functional and non-functional goals for the design of RATLS.

Freshness of Attestation Reports To prevent replay attacks, it is essential that pre-generation of attestation reports is not possible. Otherwise, an attacker could intercept a valid attestation report and send it again when trying to spoof an identity in an impersonation attack. To prevent replay attacks, RATLS must include a *nonce*, which is generated by the challenger, in each attestation report issued by the attester.

Mutual Attestation In many distributed-computing use cases, both parties need to trust each other. For example, in so-called zero-trust scenarios in cloud environments, multiple services communicate with each other over TLS and both client and server must authenticate themselves. To fully support such mutual TLS (mTLS) connections, RATLS should enable *mutual attestation* as well. Thus, both the client and the server shall be able to request an attestation report from their respective peer.

Minimal Number of Handshake Messages Round-trip messages require both client and server to wait, which is particularly costly in case of high-latency networks. Therefore, RATLS should not increase the number of handshake messages that need to be sent and received during handshake. We consider both one-sided and mutual remote attestation for this design goal.

Session Resumption In environments where connections are opened and closed frequently between the same peers, the TLS standard allows a client to resume a recently-closed session instead of performing the complete TLS handshake with the server again. To keep the benefits of this optimization, RATLS shall support TLS session resumption in a way that minimizes the cryptography-related costs for creating and validating attestation reports.

"Don't Roll Your Own Crypto" The TLS standard and its implementations have been subject to extensive review by a huge number of experts in the fields of computer security and distributed-systems engineering. Therefore, we must avoid changes to the TLS protocol and, ideally, RATLS should even be compatible with an existing TLS implementation without further modifications. These two sub-goals minimize the risk of RATLS introducing new security vulnerabilities. They also reduce maintenance overhead and make it easier to keep RATLS up to date with future TLS standards and implementations thereof.

Low-Barrier Adoption The API offered by RATLS should be as simple as possible and closely follow the API of the TLS implementation it improves upon. This simplicity will make it easy for application developers to upgrade their communication from traditional TLS to TLS with remote attestation.

Separation of Concept and Realization Remote attestation is a Trusted-Computing concept, but to use it in practice, it needs to be implemented for a

concrete computer platform with a suitable hardware root of trust. Therefore, we aim to integrate the platform-independent concept of remote attestation in RATLS, but keep separate the support for specific attestation providers as "plugins" that extend the RALTS implementation.

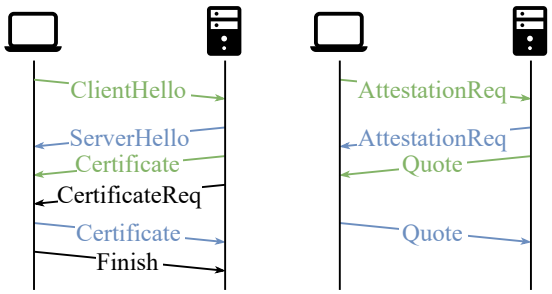

Fig. 1. Simplified visualization of the TLS v1.3 handshake (left) and protocol steps of remote attestation (right)

3.2 High-Level Design

The two sequence diagrams in Fig. 1 show simplified visualizations of the TLS v1.3 handshake (left) and the steps of the remote attestation protocol (right). Groups of arrows pointing in the same direction represent protocol information that can be transmitted in a single batched message. For example, in the TLS handshake, the server can send the `ServerHello`, `Certificate`, and `CertificateReq` messages in a single reply to the `ClientHello` message. The green and blue colors highlight conceptual similarities between establishing a TLS connection and performing a remote attestation. They give an idea of how the two protocols could be folded into one combined handshake, which establishes a mutually-authenticated TLS connection with mutual remote attestation between client and server running in parallel.

Combined RATLS Handshake RATLS builds upon handshake extensions, which have been standardized in TLS v1.3 [10]. They allow an application to append arbitrary information to the TLS messages that are exchanged during the three-phase TLS handshake. The combined handshake works as follows:

1. At the beginning of the TLS handshake, the client and the server send the `ClientHello` and `ServerHello` messages, respectively. RATLS appends to these messages the attestation requests of both peers. An `AttestationReq` message carries a nonce that the sender picked randomly in its role as a remote-attestation challenger.

2. In their role as the attester in the remote-attestation protocol, the client and server request a quote from the attestation provider of their respective devices. Each quote includes the nonce received from the respective challenger and the recorded state of the software on the attester. Each attester must also include the public counterpart to its TLS private key in the attestation report. By including the public key, both parts of the combined handshake are cryptographically linked.
3. In the final step, TLS validates the certificates. With RATLS, both parties also check the validity of the attestation reports. The corresponding quotes are piggybacked as extensions on the `Certificate` messages. At this stage of the TLS v1.3 handshake, all messages and their extensions are already encrypted. Thus, the quotes are never transmitted as plaintext.

The combined validation of the certificates and the attestation reports is more complex than in pure TLS. In the final step of the handshake, the challenger verifies that 1) the nonce is the one sent in step 1, and 2) the attester is indeed in possession of the private key that signs information in the TLS part of the handshake. To do that, it compares the public keys (embedded in the client and server certificates) from the `Certificate` message to those in the quote. If the public keys match, the connection is indeed end-to-end encrypted between the client and server. In case of a mismatch of either the nonce or the public key, a man-in-the-middle attack has been attempted and the handshake must be aborted to prevent an insecure (i.e., not end-to-end encrypted) connection.

Properties of the RATLS Handshake By piggybacking attestation-related information on the three batches of TLS messages, we can integrate remote attestation into TLS without additional network round trips. The combination of both protocols is also convenient from application's point of view. Once the combined RATLS handshake completed, both client-side and server-side applications can be certain that the hardware identity and software integrity of the remote peer have been verified and found to be trustworthy. If one of the peers does not need an attestation report from the other party, it can just omit the `AttestationReq` extension in its `ClientHello` or `ServerHello` message.

Session Resumption When the client resumes a previously closed TLS session, some parts of the handshake, including certificate exchange, are skipped. Instead, the client presents to the server a session ticket that includes the previously negotiated session keys. This *resume handshake* is shorter and therefore the remote-attestation protocol cannot be piggybacked on it. To provide the additional security guarantees of remote attestation also with session resumption, we borrow from TLS the idea of keeping session secrets for later use. In a nutshell, our variation of the approach in RATLS works as follows:

1. During the lifetime of a TLS session, the server sends a `NewSessionTicket` message to the client. This message carries the session ticket that the client can later use to resume the session. In RATLS, the server creates a pair of additional secrets, namely a *client secret* and a *server secret*. It appends

these secrets to the `NewSessionTicket` message and when the client receives this message, it stores the server secret in sealed memory. The server keeps a copy of these secrets, too. It seals the client secret.
2. When an RATLS-enabled client resumes the session at a later point in time, it unseals the server secret. It appends this secret to the `ClientHello` message and sends it to the server. The server does the same with the client secret, which it will send to the client in its `ServerHello` reply. Both client and server then compare the received secrets with their locally stored copies.

We introduce the client and server secrets, because OpenSSL's API does not allow RATLS seal, discard, and later unseal session tickets. By sealing the secrets, the client and server bind them to the software states that have previously been attested as part of the RATLS handshake. The capability of the client and server to unseal the secrets at a later time is used to prove that the session had originally been established between remotely attested RATLS peer.

We discuss further details about RATLS session resumption and all other parts of the implementation in the following Section 4.

4 Implementation

In this section, we describe an implementation of RATLS that is compatible with the widely-used OpenSSL library. We describe in detail how the combined RATLS handshake works, both for new connections and for session resumption. We will also describe the plugin API for attestation providers and an example implementation of such a plugin for TPMs.

4.1 Architecture

The general design of RATLS is independent of both the TLS implementation and the attestation provider that is needed for a specific computer platform. Although message extensions are part of the TLS v1.3 standard, not all TLS libraries support them. The OpenSSL library does offer an API, which is based on user-defined callbacks. We therefore built a companion library to OpenSSL that implements our RATLS prototype as a callback-driven state machine on top of this interface.

OpenSSL API for Message Extensions Whenever OpenSSL creates or consumes a TLS protocol message during the lifetime of a TLS session, it calls a function that RATLS registered for the corresponding OpenSSL session context (`SSLContext`). We refer to these two functions as `AddCallback` and `ParseCallback`, respectively. Before sending a message, OpenSSL invokes the `AddCallback` for any extensions that have been registered for that specific message. The callbacks can specify whether or not to add the extension and what data to populate it with. When receiving a message, OpenSSL calls the `ParseCallback` for all registered extensions. In this `ParseCallback`, we can

extract the extension data and also decide whether to abort or continue the handshake. There is no way in OpenSSL to define a callback for a missing extension. If the extension is not set, no callback is invoked. Also, according to the specification, TLS v1.3 does not allow applications to freely add extensions to arbitrary messages. Instead, extensions can only be appended if the same extension has already been added to the corresponding, previously received message. For example, to add an extension to the `Certificate` message on the server side, the same extension must be set in the `ClientHello` message.

RATLS State Management OpenSSL defines additional callbacks for events such as creation of a new session or certificate verification. RALTLS registers appropriate callback functions for all relevant events. When invoked by OpenSSL, each of these callbacks receives a pointer to the current SSL session object. In this object, the RATLS functions maintain an additional `RASession` that represents the attestation-related state during the lifetime of the session. The complete set of RATLS callback functions implement the generic concept of remote attestation for TLS.

Attestation Provider Callbacks Each of the RATLS functions invokes another callback function that is implemented by an attestation provider plugin. This second layer of callbacks separates the platform-specific root of trust and its system-level support software from the generic parts of RATLS. When the application initializes OpenSSL and RATLS, it must register the callbacks of the attestation provider with their generic counterparts in the RATLS library.

4.2 RATLS Handshake with Remote Attestation

The sequence diagram in Fig. 2 visualizes the complete, mutual handshake for both client and server attestation. For readability reasons, we explain in the following paragraphs only how the server attests to the client. The steps for attesting the client's identity and software state to the server are analogous and interleaved in the RATLS handshake as shown in Fig. 2.

Request Phase Each TLS handshake starts with a `ClientHello` message sent by the client to the server. If RATLS has been enabled for the specific `SSLContext`, OpenSSL invokes the `AddCallback` for the remote attestation request (`RA_REQ`) extension. The `AddCallback` invokes another callback function for generating an `RA_REQ`. This `CreateRequest` callback is provided by the attestation provider. It is called in the client's role as the challenger of the remote-attestation protocol and its main purpose is to randomly generate a nonce.

Attestation Phase Upon receiving the `ClientHello` message, the server detects the `RA_REQ` extension and invokes the `ParseCallback`. The server stores the client's nonce in the `RASession` part of OpenSSL's SSL session object, such that the next RATLS callback function can retrieve the nonce from there. The handshake continues until the server intends to send the `Certificate` message. OpenSSL calls the RATLS `AddCallback`, this time for the remote attestation response (`RA_RES`) extension. From this callback, RATLS invokes the

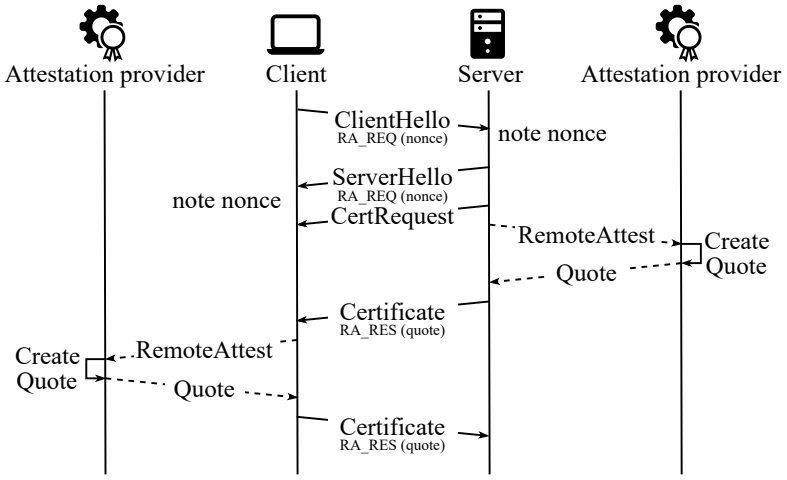

Fig. 2. Remote attested handshake

`RemoteAttest` function of the attestation provider plugin with the previously stored nonce as a parameter. In the `RemoteAttest` function, the server's attestation provider creates a quote of the client's nonce, the TLS public key, and the system state. The user-defined data that has been passed via `CreateRequest` will also be included in the quote. RATLS appends the quote to the `Certificate` message.

Verification Phase When RATLS on the client receives the `Certificate` message via the `ParseCallback`, it stores the server's quote in the client-side `RASession` for later use. Later, OpenSSL invokes the certificate-validation callback of RATLS. Here, RATLS checks if the client application originally requested an attestation. This information is expresses as a flag in the `RASession`. If the flag is true and the server ignored the request, the handshake is aborted. If the server did append an `RA_RES` extension with a quote, RATLS calls the attestation provider's `CheckQuote` function with the original nonce and the received quote as a parameter. In it's role as remote-attestation challenger, it then checks the quote's signature, compares the copies of the public key in the TLS certificate and the quote, and aborts the handshake if there is a mismatch. The `CheckQuote` function also decides whether the server's software state as reported in the quote is acceptable or not.

4.3 RATLS Handshake with Session Resumption

When the client resumes a TLS session, RATLS uses the same callback-based approach to create and inspect message extensions.

Binding Phase Session resumption requires a preparatory step while a TLS session is active. The specification allows the server to send a `NewSessionTicket`

message at any point during the lifetime of this session. When that happens, a server-side RATLS callback function creates and appends two new secrets, the *client secret* and the *server secret*, to this message. On the server, RATLS keeps the server secret in plaintext, but it invokes the `SealSecret` callback function of its attestation provider plugin to seal the client secret. It then discards the plaintext copy of the client secret. The client performs a similar procedure after receiving the two secrets in the `RA_RESUMPTION` extension of the `NewSessionTicket` message. It stores the server secret in sealed memory and keeps the client secret in plaintext. Fig. 3 shows the sequence of messages to perform this exchange.

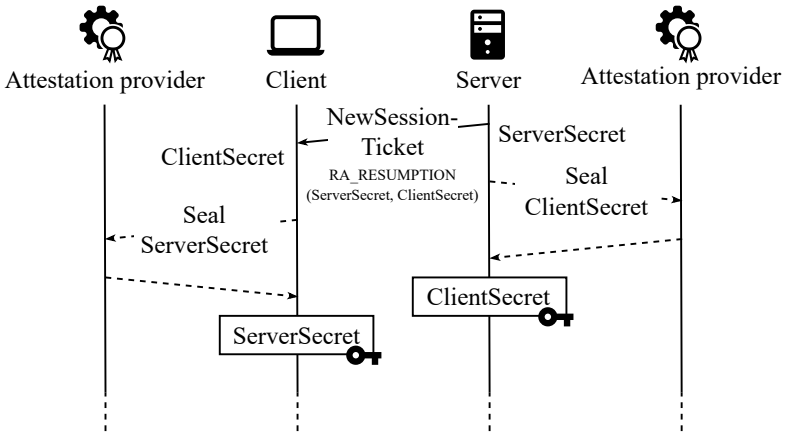

Fig. 3. Attested ticket issuing

Unseal and Resume Phase As explained at the end of Section 3.2, OpenSSL does not allow RATLS to access the session ticket. Instead the client must append the server secret to the `ClientHello` message when it wants to resume an RATLS-enabled session. Since it discarded the plaintext copy after sealing this secret, RATLS must first unseal it using the `UnsealSecret` function of its attestation provider plugin. When the server receives the session ticket and the client's copy of the server secret via the `ClientHello` message, it looks up the secret that matches the session ticket. It then compares the client's version of this secret with its own, locally found copy. It allows the session to resume, if the two copies of the client secret match and the session ticket is valid. As shown in Fig. 4, the server proves the capability to access its previously sealed copy of the client secret in the same way.

Security of Session Resumption Because of the way the TLS protocol works, the `NewSessionTicket` message and its `RA_RESUMPTION` extension are end-to-end encrypted between client and server. However, when resuming, the `RA_RESUMPTION` extension attached to the `ClientHello` must be transmitted in

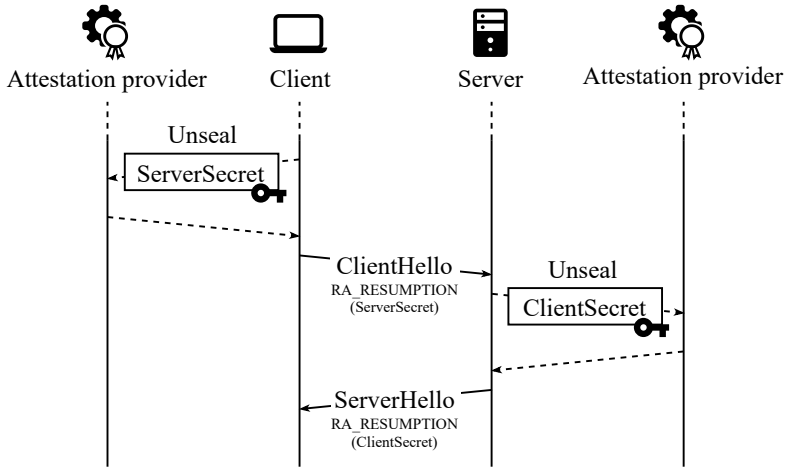

Fig. 4. Resumed attested handshake

plaintext and the server secret is potentially revealed to an observing attacker. Nevertheless, an impersonation attack cannot be mounted using just the server secret:

1. TLS Session tickets are cryptographically bound to the client and server that negotiated a TLS session. Therefore only a specific pair of client and server processes have access to the pre-shared session keys associated with the session ticket. Well-behaving TLS clients and servers do not compromise session tickets and keys.
2. If there is a client and server secret for a session ticket, this session ticket has been exchanged between two computers whose software stacks have been verified and found to use well-behaving TLS implementations based on remote attestation during the initial, non-resume handshake.
3. The fact that a server secret (or client secret) is presented during a resume handshake means that the software on the client (or server) has been in the correct state at the time of the resumption attempt. Otherwise, the attestation provider of the respective device could not have successfully unsealed the plaintext secret that the well-behaving RATLS implementation discarded before. The pre-shared session key are required to complete the handshake.

Thus, despite their name, the client and server secrets are not used for cryptographic purposes or as an authentication token. Instead, they merely serve as a hint that the session that is being resumed has been previously negotiated with a well-behaving client or server that must still be in the same software and hardware state as at the time of the attestation. An eavesdropper has no use for the secrets as the session is still cryptograhpically bound to the TLS session ticket.

4.4 Attestation provider plugins

RATLS integrates the concept of remote attestation into the TLS handshake but leaves the implementation open to a specific attestation provider. All attestation provider-specific functionality is offloaded to callbacks. This allows RATLS to be easily used with a variety of attestation providers. The API callbacks of RATLS are described below. These callbacks could map directly to the attestation provider plugins API, making them trivial to use:

- `CreateRequest` is the callback that generates the nonce for the attestation. Also user specific data could be included, which in turn then is also part of the generated attestation report.
- `RemoteAttest` is called by RATLS upon receiving an attestation request. It passes the requested nonce to the callback. This function should return a quote based on the requested nonce.
- `CheckQuote` is called, when RATLS received a quote. This quote and the expected nonce are passed as parameters to the function. The function returns true, if the quote comes from a trusted attestation provider and matches the requested nonce. Otherwise, false is returned.
- `SealSessionSecret` is called by RATLS upon receiving a session secret. This callback binds the secret to the system state and returns the encrypted session secret.
- `UnsealSessionSecret` is the inverse operation to `SealSessionSecret`. It unseals the secret and returns it as plaintext.

RATLS must register two callback functions in the OpenSSL context to work as intended. If an application registered its own callbacks for the same OpenSSL handshake events, it would disable RATLS. Therefore, RATLS provides the following replacement callbacks, which such an application can use instead:

- `CustomNewSession` mirrors the functionality of the `new_session_cb` callback in OpenSSL. It gets called when a session ticket is issued or received.
- `CustomVerifyCallback` mirrors the `verify_callback` and is invoked when OpenSSL verifies the certificate chain.

Customization Registering these two callbacks is optional. Furthermore, the behavior of RATLS can be adjusted by the following parameters:

- `maxSessionTicketsNum` specifies the maximum number of session tickets that RATLS should keep stored, making it possible to limit the memory footprint.
- `onlyAllowRemoteAttestedSessionResumption` can be specified on the client side to prevent the use of session tickets that resulted from unattested sessions.
- `forceClientRemoteAttestation` is a server-side parameter. When set to true, the server will abort the handshake, if the client has not attested itself. When false, unattested TLS connections are accepted, too.

RATLS_TPM2 RATLS comes with a sample implementation of an attestation provider plugin for TPM v2.0-based roots of trust. The RATLS_TPM2 plugin is based on Microsoft's TSS.MSR [5] library, which allows communication with both hardware and software TPMs. We use the C++ version of TSS.MSR, which we extended with a driver back-end class for accessing TPMs via the `/dev/tpm0` character device on Linux. RATLS_TPM2 provides all callback functions that RATLS needs and it performs all `TPM_Quote`, `TPM_Seal`, and `TPM_Unseal` operations that a TPM v2.0 attestation provider must use to fulfill its purpose. It also verifies quotes using TSS.MSR, as a real implementation would do, but with demonstration-only TPM storage and attestation keys.

5 Evaluation

In this section, we evaluate the usability, security, and performance of our RATLS prototype implementation.

5.1 Usability

Activating RATLS for a specific OpenSSL context is as simple as performing one initialization call. After that all handshakes in that context are attested. To use RATLS with a specific attestation provider, the application just needs to register the callback functions of the plugin library.

5.2 Security

RATLS is about integrating the concept of remote attestation into TLS, but its security guarantees build upon the underlying attestation provider and its system-level integration. These lower layers must ensure secure startup of applications that use RATLS. They also provide the functionality that RATLS needs to request quotes that attest identity, integrity, and possession of certain secrets for the system and application using RATLS. However, the specifics of their implementation are out of scope for this paper, as RATLS does not need to alter the quote format employed by the underlying root of trust. It transmits each quote as an opaque piece of data. Furthermore, RATLS does not alter the TLS protocol or its implementation, but rather extends OpenSSL through publicly available interfaces. Therefore, we are confident that RATLS does not individually weaken the security properties of remote attestation or TLS. However, two potential issues remain.

Code Size First, RATLS itself contributes additional library code to an application using it and therefore increases code complexity. Our prototype implementation adds 1,145 and 384 lines of C++ code for RATLS and RATLS_TPM2, respectively. OpenSSL is much more complex, as the entire package consists of hundreds of thousands of lines of code. The C++ implementation of TSS.MSR comprises more than 30,000 lines of code. Weighed against the stronger cryptographic assertions offered by attestation, we consider this a worthwhile addition.

Protocol Composition Second, although we reuse attestation and TLS without modification, RATLS could introduce weaknesses at the meeting points of both protocols. We performed a manual audit and identified one critical point: Looking at Fig. 2, we observe that a malicious server could try to fake an attestation response in the server-side `RemoteAttest` step. Instead of asking its local root of trust for a quote, the server could instead become a client to a new attested handshake and pass along the quote obtained from this new connection as its own. The original client would thus receive a valid quote as part of a valid TLS connection, but from different remote machines. RATLS defends against this attack by including the public keys of the TLS certificates in the measurements that are reported by the quote. By comparing the keys in the quote and the certificate, each peer can verify that the quote it received originates from the machine terminating the TLS connection and not from a third party.

5.3 Performance

We evaluate the performance of our RATLS prototype implementation with standard, non-attested TLS as a baseline. The evaluation was carried out on two Raspberry Pi 4 single-board computers acting as client and server. Each Raspberry Pi had an Infineon Optiga SLB 9670 TPM 2.0 plugged onto the GPIO pin header of the device. Both devices were located in the same local-area network with a round-trip latency of less than half a millisecond.

Baseline We benchmarked four variants for establishing a TLS connection: 1) mutually-attested RATLS handshake, 2) RATLS session resumption using sealing, 3) standard TLS handshakes, and 4) standard TLS session resumption. Variants 3 and 4 represent the baseline, using the same OpenSSL version and parameters as RATLS. All experiments were run 100 times. Tab. 1 shows the average duration to complete the handshake for all four variants and for both server and client side. Variation was low, as indicated by the standard deviation (STDEV) figures in the table.

Table 1. Comparison of RATLS and TLS handshake duration

Benchmarks	Server		Client	
	Avg. time	STDEV	Avg. time	STDEV
RATLS initial handshake	616.06 ms	2.92 ms	525.30 ms	2.88 ms
RATLS resume handshake	156.23 ms	1.42 ms	114.28 ms	1.30 ms
TLS initial handshake	52.89 ms	2.03 ms	52.60 ms	2.04 ms
TLS resume handshake	2.97 ms	0.38 ms	2.79 ms	0.37 ms

Initial Handshake Measurements Mutually-attested RATLS handshakes are significantly slower than standard TLS handshakes without remote attestation. The observed 10x overhead is caused almost entirely by cryptographic

operations being performed inside the Optiga TPM. On average, `TPM_Quote` and `TPM_Seal` operations take 212 and 42 milliseconds, respectively. As the TLS handshake protocol forces client and server to perform their quote operations one after the other (see Fig. 2 on page 11), these costs add up for mutually-attested sessions. The measurements also include the cost for sealing the session secrets on the server. A breakdown of these costs, including the time spent on the TLS part of the protocol, is shown in Fig. 5. Note that the client receives multiple `NewSessionTicket` messages that trigger `TPM_Seal` operations. But two of these messages arrives after the handshake already completed on the client side. Hence, their costs are not captured in the client-side figures in the table, but we confirmed that the operations are performed by the client and the costs are as expected.

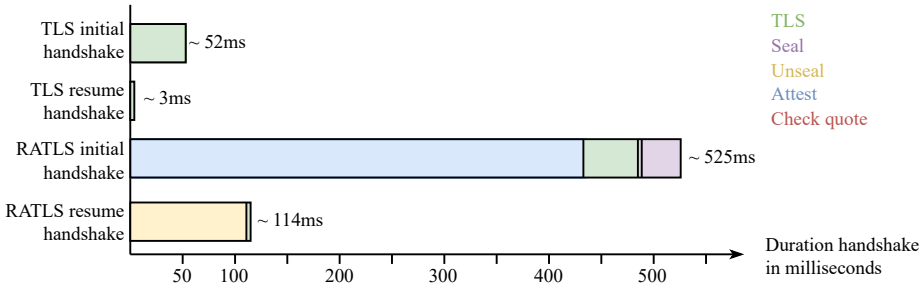

Fig. 5. Duration of client side handshakes in comparison

Resume Handshake Measurements The TLS-only bars in Fig. 5 show that TLS session resumption can speed up TLS re-connects. Fortunately, RATLS can play the same trick to reduce the attestation-related costs. RATLS session resumption is dominated by the duration of two `TPM_Unseal` operations, one performed by the client and one on the server. As unsealing is cheaper on the Optiga TPM than generating a quote, RATLS re-connects are about four times faster than a complete RATLS handshake. On the server, we measured 156 milliseconds, whereas the client finishes the resume handshake after 114 milliseconds. Like above, for the complete handshake, this difference is caused by session-ticket messages arriving after the client-side finished the handshake.

Discussion We acknowledge that RATLS takes significantly more time to establish a secure connection than standard TLS. However, our benchmarks represent a worst-case scenario, because discrete TPM chips like the ones we used are among the slowest roots of trust that are available. Also, the relative performance benefits of session resumption would be greater in higher-latency networks (e.g., over the Internet); we used an Ethernet link with 0.5 ms latency. The additional costs pay for the additional security guarantees that remote attestation provides.

6 Related Work

RATLS integrates remote attestation into the TLS handshake. Other works have explored integration at levels below or above the TLS protocol layer with resulting differences in usability or generality.

SGX Remote Attestation with TLS Knauth et al. integrated attestation for Intel SGX enclaves with TLS [9]. They chose not to change or extend the TLS protocol or implementation, but included an attestation quote into the X.509 certificate used for authentication. A certificate extension is used to carry the additional information. This method of integration is fully transparent to the TLS layer and therefore works with any TLS implementation. However, a new certificate must be minted for every attestation, complicating the interaction with existing TLS certificate hierarchies. The paper therefore restricts its scope to self-signed certificates. RATLS does not alter certificates and thus can fully reuse existing certificate chains and the trust relationships they encode.

HTTPA King and Wang proposed HTTPA, the HTTPS Attestable Protocol [8]. This work integrates attestation in a protocol layer above TLS, by proposing changes to the HTTP layer. New HTTP messages are used to exchange bidirectional attestation information. Consequently, no changes to TLS implementations or certificates are needed. However, attestation is specific to HTTP and must be integrated into application-level code. By encapsulating attestation in TLS, RATLS gives developers TLS encryption with automatic remote attestation for any application-layer protocol with just a few lines of code.

DECENT Zheng and Arden published DECENT [11], which is an attestation system for decentralized applications consisting of multiple distributed components. These components mutually authenticate and attest themselves. In order to save expensive attestation operations, DECENT proposes mechanisms to perform attestation only once at component launch. TLS-based protocols like RATLS would have to re-attest components for every established connection. However, because we integrated session resumption, RATLS can keep attestation information alive and reusable, similarly saving expensive operations.

LightBox Duan et al. describe an example of how trusted execution environments can be used to protect metadata of network applications. Their network middlebox system, called LightBox [6], tightly integrates with Intel SGX [1]. Their proposed design is highly optimized for operation in SGX enclaves to avoid computational overhead when handling packet routing inside an SGX enclave. Although RATLS could use SGX as an attestation provider for an application running in an SGX enclave, its goals are orthogonal. Namely, RATLS aims to integrate the concept of remote attestation into the TLS protocol, so that it can be used in a variety of applications with minimal effort on behalf of the application developer.

Benefits of Remote Attestation Other works point out security benefits of trusted execution environments and using their roots-of-trust for remote attes-

tation. For example, Kim et al. published [7] case studies for leveraging of SGX for privacy sensitive applications. In their use cases, they introduce attestation schemes for inter-domain routing, mix relays like TOR, and other types of middle boxes. The goals and benefits of RATLS are similar to what they present in terms of establishing a secure channel between attested endpoints or middle boxes. RATLS could be used as a building block to implement such use cases and because of its modular design, it is compatible with a variety of attestation providers besides SGX. But most importantly, our work aims to be a general solution that is easy to use. Thus, by integrating remote attestation into the TLS Handshake, RATLS makes it trivial to upgrade existing TLS connections to remote attested sessions in many other application scenarios.

7 Conclusions

In this paper, we presented the design and implementation of RATLS, which integrates the concept of Remote Attestation into the Transport Layer Security (TLS) protocol. RATLS provides additional security guarantees for authentication and software integrity of TLS endpoints. Our implementation builds upon message extensions in v1.3 of the TLS standard. This approach requires no modifications to the TLS protocol or its implementation, thereby minimizing the risk of introducing new security weaknesses. Our prototype is compatible with Trusted Platform Modules (TPMs), but thanks to a modular design, other hardware roots of trust could be supported via attestation provider plugins.

Acknowledgements This research was co-financed by public funding of the state of Saxony/Germany. It has also received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 957216.

References

1. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, (Accessed on May 1, 2022)
2. OpenSSL, https://www.openssl.org/
3. The Trusted Computing Group - TPM Software Stack (TSS), https://trustedcomputinggroup.org/work-groups/software-stack/
4. The Trusted Computing Group - Trusted Platform Module (TPM), https://trustedcomputinggroup.org/work-groups/trusted-platform-module/
5. TSS.MSR, https://github.com/microsoft/TSS.MSR
6. Duan, H., Wang, C., Yuan, X., Zhou, Y., Wang, Q., Ren, K.: LightBox: Full-stack protected stateful middlebox at lightning speed p. 2351–2367 (2019). https://doi.org/10.1145/3319535.3339814, https://doi.org/10.1145/3319535.3339814
7. Kim, S., Shin, Y., Ha, J., Kim, T., Han, D.: A first step towards leveraging commodity trusted execution environments for network applications (2015). https://doi.org/10.1145/2834050.2834100, https://doi.org/10.1145/2834050.2834100

8. King, G., Wang, H.: HTTPA: HTTPS Attestable Protocol. CoRR **abs/2110.07954** (2021), https://arxiv.org/abs/2110.07954
9. Knauth, T., Steiner, M., Chakrabarti, S., Lei, L., Xing, C., Vij, M.: Integrating Remote Attestation with Transport Layer Security. CoRR **abs/1801.05863** (2018), http://arxiv.org/abs/1801.05863
10. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor (August 2018), https://www.rfc-editor.org/rfc/rfc8446.txt
11. Zheng, H., Arden, O.: Building secure distributed applications the DECENT way. CoRR **abs/2004.02020** (2020), https://arxiv.org/abs/2004.02020