Integrating Remote Attestation with Transport Layer Security

Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, Mona Vij
Intel Labs
firstname.lastname@intel.com

ABSTRACT

Intel® Software Guard Extensions (Intel® SGX) is a promising technology to securely process information in otherwise untrusted environments. An important aspect of Intel SGX is the ability to perform remote attestation to assess the endpoint's trustworthiness. Ultimately, remote attestation will result in an attested secure channel to provision secrets to the enclave.

We seamlessly combine Intel SGX remote attestation with the establishment of a standard Transport Layer Security (TLS) connection. Remote attestation is performed during the connection setup. To achieve this, we neither change the TLS protocol, nor do we modify existing protocol implementations.

We have prototype implementations for three widely used open-source TLS libraries – OpenSSL, wolfSSL and mbedTLS. We describe the requirements, design and implementation details to seamlessly bind attested TLS endpoints to Intel SGX enclaves.

1. INTRODUCTION

Intel SGX is a recent (2015) Intel processor extension available since the 6th Gen Intel® Core™ processors. Intel SGX enables application developers to construct trusted execution environments – called enclaves – to perform computation on commodity CPUs while maintaining previously untenable security protections. Even highly privileged software running concurrently on the same hardware, say, the operating system, and virtual machine monitor, cannot observe an enclave's data in clear text. This is possible thanks to changes in the microarchitecture that essentially prevent access to enclave memory by anyone except the enclave to which the memory belongs.

An integral part of the Intel SGX architecture is the ability to perform attestation. The *attester* wants to convince the *challenger* that it is a genuine Intel SGX enclave running on an up-to-date platform. At the end of the attestation process the enclave has convinced the challenger that it is genuine. Based on the enclave's attested attributes, the challenger decides whether to trust the enclave or not.

Previous work has shown that remote attestation and secure channel establishment must be integrated to protect against man-in-the-middle attacks [1]. At a minimum, the remote attestation protocol should result in a shared secret that can function as the basis for a secure channel. The current Intel SGX SDK provides an instance of this concept: remote attestation is performed using a modified Sigma [2] protocol. After a successful protocol instance the attester and challenger share a secret.

However, a shared secret only partially solves the problem of secure communication. Bootstrapping a secure channel based on a shared secret is possible, but inefficient since it duplicates work. Instead, we want to seamlessly integrate attestation with the establishment of a standard secure channel. The end result is an attested secure channel through which the participants can communicate.

In this white paper we describe our approach to combine Intel SGX remote attestation seamlessly into the existing Transport Layer Security (TLS) secure channel protocol. Most importantly we do leave the TLS protocol unchanged, allowing us to reuse existing implementations. Additionally, developers are already familiar with TLS's secure channel API, which eases adoption.

We first provide some background on remote attestation and TLS in Section 2. Section 3 presents our approach to incorporate remote attestation into the TLS protocol. Section 4 gives implementation details and describes the API. In Section 5 we discuss future extensions and challenges. Section 6 discusses related work before concluding in Section 7.

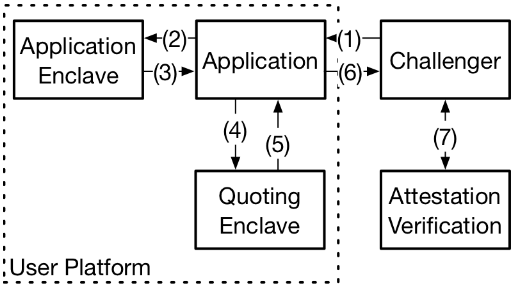

Figure 1: Remote Attestation Example. The challenger is off-platform with respect to the attester.

2. BACKGROUND

In this section we provide background information on SGX remote attestation (Sec 2.1) and the Transport Layer Security (TLS) protocol (Sec 2.2). Readers intimately familiar with either may want to skip the respective sub-sections.

2.1. REMOTE ATTESTATION (RA)

The concept of attestation was previously explored in the context of Trusted Computing [3]. A secure co-processor, called Trusted Platform Module (TPM), performed cryptographic operations, such as key generation, signing and storing of keys, in a secure environment. In addition, a TPM can also attest the overall platform's state and configuration to other interested parties [4]. The assessing party is called *challenger* while the assessed party is the *attester*. Based on the attestation, the challenger can decide whether to trust the platform by comparing the attester's state to a reference value.

Attestation is also an integral feature of the Intel SGX architecture although the implementation details differ compared to TPMs. Intel SGX provides two attestation mechanisms [5] depending on where the challenger and attester reside. *Local attestation* allows two parties on the same platform to attest each other. The enclaves perform a local message exchange to assess their respective trustworthiness. With *remote attestation* the challenger is off-platform and typically communicates with Intel's Attestation Service (IAS) during the attestation procedure.

Intel SGX attestation covers the identity of the software running in the enclave (e.g., MRENCLAVE and MRSIGNER), non-measurable state, such as the enclave mode (e.g., debug vs. production), additional data the enclave wants to associate with itself (e.g., a manifest describing the software's configuration), and a cryptographic binding to the platform TCB. This information is bundled into a data structure called quote that is signed by an attestation key.

Figure 1 illustrates a possible remote attestation flow [5] using a modified Sigma protocol. The challenger sends a nonce to the application to guarantee freshness (Step 1). The application creates a manifest that includes a response to the challenge as well as an ephemeral key to encrypt future communication with the challenger (Steps 2 and 3). The application computes a hash of the manifest and includes this hash as user-defined data when creating the report. Including a hash of the manifest into the report is crucial, as this binds the secret key to this enclave. Binding the key to the specific enclave instance is crucial to avoid masquerading attacks [6] [1] [7].

Next, the enclave generates a report that summarizes the enclave and platform state. The report includes information on the platform (security version number), enclave attributes, enclave measurement, software version, software vendor security version number, and additional user-provided data [8]. The quoting enclave verifies and signs the report (Step 4). The signed report, now called a quote, is returned to the application (Step 5) which passes it on to the challenger (Step 6). The Intel Attestation Service (IAS) verifies the quote (Step 7).

The Attestation Service will reply with an *attestation verification report*, confirming or denying the authenticity of the quote and the enclave it originates from [9]. The interaction with IAS is necessary to preserve the attested platform's privacy. In v2 of the IAS specification the attestation verification reply includes the original quote. The reply can thus be verified by 3rd parties other than the original challenger.

The Intel SGX SDK provides the necessary APIs and primitives to do remote attestation using a modified Sigma protocol [10]. Note though, that the Sigma protocol only results in a shared secret, not a secure channel. The application is left with the task to "extend" the shared secret into a secure channel on its own. We want to bridge this gap and integrate remote attestation seamlessly with a standard secure channel. This will make remote attestation easier to use in practice and facilitate the wider uptake of Intel SGX.

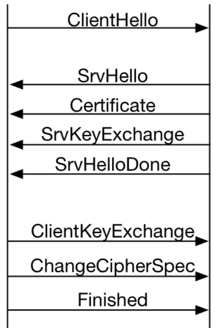

Figure 2 TLS Handshake Messages

2.2. TRANSPORT LAYER SECURITY

Transport Layer Security (TLS) is the de-facto industry standard to secure communication. It has been developed since 1995 and is still evolving, with TLS version 1.3 currently being drafted. Even with this long history, subtle deficiencies are still discovered occasionally [11]. Thus our strong motivation to build on a standard secure channel protocol instead of custom solutions.

Authentication. TLS not only protects the integrity and confidentiality of data but also allows endpoint authentication through the exchange of X.509 certificates. In a typical scenario where a browser (client) connects to a web site (server), only the server sends a certificate. The client uses the certificate to confirm that it indeed connected to the intended web site.

TLS also allows both endpoints to authenticate each other. Mutual authentication is frequently used with known client populations, for example, in enterprise settings, where only previously registered clients are allowed to connect. Again, trust in the endpoint is conveyed through the exchange of certificates and their verification through a certificate chain linking back to a trusted root certificate.

X.509 Certificates. The use of certificates for the purpose of identification is based on a Public Key Infrastructure (PKI). A set of trusted root certificate authorities either create leaf certificates directly or delegate the responsibility to an intermediate certificate authority (CA). The intermediate CA can issue leaf certificates "on behalf of" the root CA. In the case of securing web sites with TLS, the certificate represents a cryptographic binding of the web sites domain name to the public key (subject key) referenced in the certificate. When connecting to the HTTPS web site, the client verifies the certificate chain as well as that the distinguished name in the certificate matches the domain it intended to connect to.

Besides the distinguished name, X.509 certificates can be used to bind arbitrary data to the key identity in the form of X.509 extensions. Each extension is identified by an Object Identifier (OID). For example, the OID for the well-known extension Subject Alternative Name (SAN) is 2.5.29.17. SAN allows the certificate to list multiple domain names for which the certificate is valid. X.509 extensions are not regulated. Companies are free to introduce new extensions in combination with the products they offer. To ensure interoperability, extensions can be marked critical. If a client encounters a critical extension it is unaware of, it aborts the handshake. The client is free to ignore unknown non-critical extensions.

Trust Root. With TLS and the use of X.509 certificates to identify the endpoint, the trust root lies in the root certificate authorities. PKI users trust the root CAs to follow best practices and procedures before issuing certificates to service providers. Alas, not all CAs and their delegates are created equal and some may be more trustworthy than others [12]. While browsers list hundreds of trusted CAs, scenarios other than HTTPS on the internet may use a much smaller set of trusted roots.

TLS Handshake. At the beginning of a TLS connection, the client and server engage in a 3-way handshake[1]. As part of the handshake, the two parties agree on the specifics of the cipher suite and the server authenticates itself to the client by sending a certificate. While the client initially encrypts messages to the server using the server's public key obtained from the certificate, the two parties establish a mutual session key to use after the handshake.

Figure 2 illustrates a typical handshake. The client initiates the handshake by sending a ClientHello message. The ClientHello includes a nonce, specifies the client's preferred cipher suites and any supported TLS extensions. The server replies with a sequence of three messages: ServerHello, Certificate, and KeyExchange. After the server sends its ServerHelloDone message, the client continues the

[1] We are focusing on TLS v1.2 here.

exchange with its KeyExchange and ChangeCipherSpec messages. After the handshake finishes successfully, the parties will have established a session key to authenticate and en-/decrypt future messages. A detailed description of each message and their meaning is available in the official TLS 1.2 standard [13]

As part of the handshake, the server sends an X.509 certificate to the client. The certificate states the server's public key, the certificate's validity period, and domain name(s) the certificate is valid for, among other things. The client verifies this information and performs path validation to ensure the certificate chain terminates in a trusted root certificate. Additionally, the client should also check if the certificate has been revoked. In practice, this is only done for Extended Validation (EV) certificates, a special class of certificates used by high-value web sites such as banks and insurance companies. If the client's certificate validation logic indicates a problem with the certificate, the handshake is terminated.

During the handshake, the server and client negotiate a symmetric **session key** to use after completing the handshake. The specifics of the key exchange and the authentication depend on the chosen cipher suite. For example, a cipher suite employing the Diffie-Hellman Ephemeral key exchange, results in a new key for each session. A unique session key provides forward secrecy, that is even if the server's key is compromised, an attacker cannot decrypt previously recorded sessions since each session used an ephemeral key independent of the server's key.

3. DESIGN

In contrast to the PKI used in the modern web where the trust root is a list of root certificate authorities maintained by a handful of entities such as Google, Mozilla and Microsoft, we want to use Intel SGX as a root of trust. To this end, we propose to include additional information into the X.509 certificate exchanged during a TLS handshake. In the following, we explain which information we include in the certificate and how this information is validated.

We stay with the common scenario where a central server allows arbitrary clients to connect and the clients want to verify the server's identity during the connection setup. In the terminology of remote attestation, the server is the attester and the client the challenger. We focus on the classic client/server scenario to ease the explanation but our scheme also covers mutual authentication/attestation. With mutual attestation, the client and server assume the roles of attester and challenger at the same time.

The server runs in an Intel SGX enclave and terminates the TLS connection inside the enclave. The server can either be constructed using the Intel SGX SDK or any other framework/runtime to execute legacy applications on Intel SGX [14] [15] [16] [17]. The server obtains its Intel SGX identity and executes the steps necessary to obtain a report and a quote from the platform's quoting enclave. This is required to successfully complete the remote attestation workflow.

To achieve our goal of seamless remote attestation, we have to address two objectives: First, we need to tie the server's (authentication) key to a specific instance of an Intel SGX enclave. Second, the server/attester must provide additional information during the TLS connection setup to convince the client/challenger that it is indeed connected to a veritable Intel SGX enclave.

3.1. OBJECTIVE 1: BINDING KEY TO ENCLAVE

The enclave generates a new public-private key pair at startup. Unlike with traditional PKI, the key does not reflect the enclave's identity. It merely serves as a vehicle to establish a secure channel through the standard TLS protocol. Since the key is not used for identification, there is also no need to persist it across enclave restarts. This gives the strong assurance, that the key is never exposed outside the enclave. Interested parties can inspect the source code to convince themselves that this property holds.

We bind the key to the enclave by including a hash of the public key as user-data into the Intel SGX report. Recall, that the report is generated by the attester and passed to the platform's quoting enclave. The quoting enclave signs the report, vouching that the report was indeed generated by a genuine Intel SGX enclave on the local platform. At this point, the server's public key is cryptographically bound to the Intel SGX enclave.

The server must still send the quote to IAS to obtain a quote verification report. Later, anybody can use the attestation verification report to independently verify the link between the server's public key and the server's Intel SGX identity.

Based on the key and the attestation verification report, the server creates a self-signed X.509 certificate. It includes custom X.509 extensions based on the information contained in the attestation verification report. These extensions enable the client to verify that it is connected to an Intel SGX enclave. Note that clients do not have to interact with IAS. Instead, they rely on the signed response from IAS embedded in the certificate to attest the server.

3.2. OBJECTIVE 2: TRANSPORT DATA TO CHALLENGER

To assess the server's Intel SGX identity and the link to its public key, the client needs access to the following four pieces of information:

Attestation Verification Report. This is the reply received from IAS when submitting a quote for verification [9]. It tells whether the quote was generated on a genuine Intel SGX platform by a genuine enclave. Embedded in the report is a copy of the data previously reported by the enclave, such as platform security version number, enclave identity (MRENCLAVE), etc.

Attestation Verification Report Signature. The attestation verification report is signed by IAS. The signature ensures that report is authentic and unmodified.

Attestation Report Signing [CA] Certificate. To verify the signature over the attestation report, the client must know the keys used to create the signature. This information is conveyed in additional certificates included in the response from IAS. The Attestation Report Signing CA Certificate is self-signed and trusted. Only if the entire chain verifies correctly, can the report signature be trusted.

We propose to embed this information as custom X.509 extensions into the server's certificate. For our purposes, we introduce four new extensions, each with their own unique object identifier (OID). Adding this information into the certificate is less intrusive than, for example, introducing a new TLS extension which would require invasive changes to each TLS implementation. By extending the certificate, we do not require any changes to existing TLS libraries and use readily available hooks to do the extended certificate validation.

Extending the certificate traditionally requires resigning it by a CA. However, since we propose to use Intel SGX as a trust root, we can simply self-sign the certificate. Instead of relying on the CA to bind the domain name to the server's identity/key, we rely on Intel SGX to provide the identity. If a binding between the Intel SGX identity and a domain name is desired, we discuss one proposal how to incorporate this into the attestation flow in Section 5.

3.3. EXTENDED CERTIFICATE VALIDATION

The receiver of a certificate with these extensions must perform the following steps to verify the authenticity and validity of the certificate: (1) Verify the signature on the Attestation Report Signing Certificate against the Attestation Report Signing CA Certificate (which is trusted). Only if validation passes do we have assurance that the report was signed by a trusted attestation service. (2) Verify the report signature against the report using the Attestation Report Signing Certificate to ascertain that the attestation report is genuine and unmodified. (3) Compute the SHA256 hash of the server's public key and compare this against the SGX report's user data. If they match, this is proof that the server is indeed an Intel SGX server and using this key to secure the TLS session. (4) Compare the enclave's identity (MRENCLAVE, MRSIGNER) against the expected values. They must match to ensure that the client indeed connected to the intended server. If any of the above verification steps fails, the challenger must abort the connection.

3.4. SECURITY PROPERTIES

The original modified Sigma protocol included a nonce provided by the challenger to ensure freshness. Without the nonce, an attacker could replay a previous remote attestation protocol instance – posing as the enclave – to trick the challenger to reveal secrets to an untrusted entity. The result of the modified Sigma protocol is a fresh shared secret the parties can use to secure further communication.

We achieve freshness, using different mechanisms. First, freshness of the exchanged messages is guaranteed by TLS itself: nonces at the record layer protect against replays. Freshness of the quote can be determined by the challenger. The signed attestation verification report returned by IAS includes a time stamp. Based on the time stamp, the challenger may refuse the quote as too old and terminate the connection. Since generating a new quote is low overhead, we expect the attester to obtain a new quote at reasonable intervals, e.g., once a day. This ensures that challengers always receive a fresh quote.

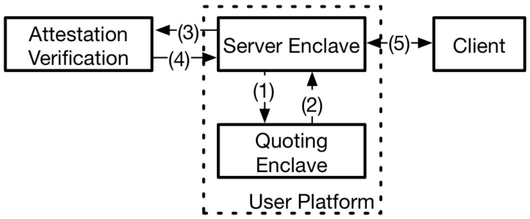

Figure 3: Message flow for remote attestation using TLS.

The security of the key is ensured by Intel SGX. The quote binds the key to a particular enclave instance. By design, only the enclave has access to the key. An attacker would need to break the Intel SGX security model or exploit application-specific vulnerabilities to compromise the key. Users can inspect the code to ensure that it never leaves the enclave intentionally. We do not intend to persist the key across enclave restarts. Instead, a new key is generated at each startup and, possibly, periodically at runtime for long running services. Usage of the key is further minimized when a cipher suite with a Diffie-Hellman Ephemeral key exchange is used. In this case, the enclave key is only used during the handshake and a per-connection ephemeral Diffie-Hellman key protects the subsequent communication.

A classic problem with traditional PKI is the regeneration of certificates, either because the certificate expired or the key was compromised, which is time-consuming and expensive. Since we root trust in the Intel SGX hardware, we can self-sign the certificate. This allows us to have short expiration times and regenerate certificates periodically. Whenever we change the enclave key, we do need to generate a new quote to reflect this. Short certificate validity periods and fresh quotes also protect challengers from connecting to revoked platforms. If a platform is revoked, the attester will no longer be able to obtain a valid attestation report from the attestation service.

4. IMPLEMENTATION

We have implemented our proposal using the wolfSSL [18] library for the cryptographic operations and certificate generation. Our C library exposes a set of functions for use by the attester and challenger. The library is linked into the application like any other standard library.

We also demonstrate how to integrate our library with two additional TLS libraries: OpenSSL [19] and mbedtls [20]. The code clearly separates the TLS library-specific parts (hash computation, certificate generation and validation, and key generation) from the library agnostic aspects (extended validation steps). This allows us to reuse the library-agnostic parts when porting our changes to new TLS libraries.

Besides the actual TLS library, we also encapsulate the Intel SGX SDK specific functionality into separate functions. Our library uses the Intel SGX SDK to obtain the quote. If the Intel SGX SDK is not available, for example, because a different framework is used [16] [15] [17] to execute the program on Intel SGX hardware, we provide an alternative implementation of the required functionality independent of the Intel SGX SDK. Hence, integrating our library with other SGX frameworks is possible and low overhead.

We describe the programming interface in terms of the attester and the challenger. Depending on the scenario, either can be the server or client in a traditional distributed computing setting. If mutual attestation is desired, and this is supported by our library and TLS, the client and server will in fact assume the role of attester and challenger at the same time.

4.1. ATTESTER

Traditionally, the server/attester reads the key and certificate from stable storage into memory before serving it to other parties. It is the attester's responsibility to apply sufficient security measures to safeguard the key at rest.

Instead we propose to generate a new key and certificate whenever the attester enclave starts. The key and certificate are ephemeral and shall not be persisted past the attester's lifetime. The attester's API consists of one function `create_key_and_certificate` that outputs the key and the corresponding certificate. To keep the interface generic, the key and certificate are encoded in standard DER format.

Figure 3 illustrates the flow of messages sent by the server when it initializes. The server runs inside an Intel SGX enclave. Before accepting any client connections, the server generates a new key and creates a self-signed X.509 certificate including the extensions mentioned in the previous section. The TLS library is configured to use this certificate as the server certificate for new client connections.

To create the certificate, the server must go through the usual steps to perform remote attestation: create

```
01 WOLFSSL_CTX* ctx;
02 ...
03 char key[2048]; char crt[8192];
04 int key_len = sizeof(key);
05 int crt_len = sizeof(crt);
06
07 create_key_and_x509(key, &key_len,
08                     crt, &crt_len);
09
10 wolfSSL_CTX_use_certificate_buffer(ctx,
11      crt, crt_len, SSL_FILETYPE_ASN1);
12
13 wolfSSL_CTX_use_PrivateKey_buffer(ctx, der_key,
14      der_key_len, SSL_FILETYPE_ASN1);
```

Figure 4: An example, based on the wolfSSL library, using the attester API. Functions exported by the remote attestation library are in bold.

a report, pass the report to the quoting enclave, and, finally, send the quote to IAS to receive an attestation verification report. These steps must be performed at startup (and every time the server's key changes at runtime). If it is desirable to maintain the same key across server restarts, the key can be sealed to the enclave's identity.

We illustrate the usage of the attester API in Figure 4. In this particular example we use the wolfSSL library to configure an endpoint accepting connections. After defining the data structures to hold the key and certificate (lines 3 – 5) we call our library function to create a new key and certificate (lines 7 – 8). Subsequently the attester calls the wolfSSL function to use the newly generated key (lines 10 – 11) and certificate (lines 13 – 14) with incoming connection requests. The simple and intuitive API enables developers to integrate our library easily into existing applications.

4.2. CHALLENGER

The challenger extends the existing certificate validation logic. To determine if a server certificate is valid, the TLS library performs a standard set of checks: it verifies the certificate's expiration status, if the certificate's content matches its signature and whether the certification chain terminates in a trusted root certificate.

All the TLS libraries we have examined allow the user to specify a custom certificate validation function. The custom validation function either extends the built-in verification logic or overrides it depending on how it is used. Using these hooks/callbacks, we implement all the required changes to the challenger without modifying the TLS library itself.

The certificate validation hook indicates the successful validation through the return value. The caller uses the hook's return value to either continue the handshake or abort it. The hook calls the function `verify_sgx_cert_extensions` exported by our library. If the verification succeeds, the function returns 0. Otherwise, the function returns 1.

```
01 int cert_verify_cb(int preverify,
02   WOLFSSL_X509_STORE_CTX* store) {
03
04   WOLFSSL_BUFFER_INFO* crt = store->certs;
05   int ret = verify_sgx_cert_extensions(
06               crt->buffer,
07               crt->length);
08   return !ret;
09 }
10
11 int main(int argc, char* argv[]) {
12   ...
13   wolfSSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,
14                          cert_verify_callback);
15   ...
16   WOLFSSL_X509* crt =
17       wolfSSL_get_peer_certificate(ssl);
18
19   int der_len;
20   const byte* der =
21       wolfSSL_X509_get_der(crt, &der_len);
22
23   sgx_quote_t quote;
24   get_quote_from_cert(der, der_len, &quote);
25   sgx_report_body_t* body = &quote.report_body;
26
27   if (0 != memcmp(body->mr_enclave.m,
28                   golden_mr_enclave,
29                   SGX_HASH_SIZE)) { exit(1) };
30   ...
31 }
```

Figure 5: An example, based on wolfSSL library, using the challenger API. Functions exported by the remote attestation library are in bold.

The application gets access to the Intel SGX identity and platform attributes through a complimentary function `get_quote_from_cert` to extract the Intel SGX quote from the X.509 certificate. The quote contains information on Intel SGX identity attributes such as MRENCLAVE and MRSIGNER as well as platform related attributes such as the CPU security version number. Challengers use the attributes to decide whether the remote end meets their security requirements.

Figure 5 illustrates how to use the challenger API in a wolfSSL-based application. First, we define a custom certificate verification function `cert_verify_cb` with the signature expected by the wolfSSL library (lines 1 – 9). This verification callback simply wraps our library function doing the extended certificate validation. Later on, we register the certificate verification callback with the wolfSSL library (lines 13 – 14). After the TLS handshake finished successfully, we obtain access to the peer's DER encoded certificate (lines 16 – 21). Next, the challenger extracts the quote from the DER encoded certificate (lines 23 – 25). The peer's MRENCLAVE value is compared against a reference value to verify the SGX identity (lines 27 – 29).

Even though the example is based on wolfSSL, the flow is similar for other TLS libraries. The data structures and function signature will be different, but the interface exposed by our library is generic to make interoperability with other TLS libraries easy.

5. LIMITATIONS AND EXTENSIONS

We discuss limitations and considerations when pursuing seamless attestation via extended X.509 certificates.

Non-standard X.509 extensions. We embed the SGX identity information in custom X.509 extensions. X.509 extensions can be marked as critical. If a client encounters an unknown critical extension, the default is to abort the connection. If the attester only expects Intel SGX-aware clients, it is sensible to mark the extensions critical. If the client includes Intel SGX-unaware legacy clients, the extensions should not be marked critical to allow backwards compatibility. A legacy client may still complain about a self-signed certificate.

Clients wishing to make use of the Intel SGX identity included in the X.509 certificate need to extract this data. To reuse existing APIs, it may make sense to fold the Intel SGX identity attributes into established X.509 fields. For example, the Common Name (CN) attribute could hold a copy of MRENCLAVE. When validating the certificate's Intel SGX extensions fields, the attester must verify that the Intel SGX values encoded in the common attributes match the ones embedded in the quote.

Certificate size. Including all the relevant additional information into the certificate increases its size significantly. A typical HTTPS certificate is around 1100 byte (DER format)[2]. A certificate extended with Intel SGX identity information is around 6200 bytes.

The size increase of the certificate may be relevant for attesters that establish new connections frequently since the certificate's size dominates the overall handshake-related traffic volume. Known techniques such as TLS sessions can help alleviate the problem.

Since the majority of the additional data pertains to the attestation server's certificate chain, it may help to distribute the relevant certificates independently through a different channel. We expect these certificates to be long-lived such that they do not have to be included in each and every handshake.

Self-signed certificate. The attester creates a self-signed certificate that represents its identity. This may pose problems with legacy clients. For example, in the case of HTTPS, clients expect a known trusted CA to sign the certificate. If the certificate is self-signed, the client's certificate validation logic will abort the handshake. A self-signed certificate will also lose the binding between the server's domain name and the key as vouched for by the CA.

A possible solution around this dilemma is to have a trusted CA sign the extended certificate. With protocols such as the Automated Certificate Management Environment (ACME) to streamline the issuance of certificates [21], this is one possibility to combine traditional X.509 identities with Intel SGX identities. Alternatively, if the attester has access to an intermediate CA certificate, this can be used to sign the Intel SGX-extended certificate. In this way, the leaf certificate chains back to a PKI trust root, albeit at the added burden to securely handle the intermediate CA's key.

Enhanced Privacy ID (EPID). The current implementation for remote attestation is focused on a client/server model. The server attests clients running on end-user platforms, i.e., consumer devices. In this scenario, the server is the challenger and the client is the attester. Maintaining the client's privacy is paramount, i.e., challengers should not be able to track clients across attestations.

This design, however, complicates the deployment of remote attestation in other scenarios. Currently, the challenger must register a self-signed certificate with IAS prior to verifying a quote. During the registration a software product ID (SPID) is assigned. Attestation requests using the certificate are only allowed for quotes matching the assigned SPID.

Reversing the roles, where the client becomes the challenger and the server the attester, is difficult in this scheme. Essentially, the entity requesting attestation is different from the entity providing the software. Version 2 of IAS now includes the original quote in its response. This at least allows a verifier to

[2] https://scans.io/data/rapid7/sonar.ssl/20170912/2017-09-12-1505178001-https_get_443_certs.gz -- We used this data set for the certificate related statistics.

inspect that a certain quote was deemed valid without communicating with IAS. An attestation model geared towards server environments would provide a means of verifying an attestation without registering with a centralized attestation service first.

6. RELATED WORK

The TLS protocol provides authentication, integrity, and privacy for data transmitted across untrusted communication lines. However, people have cautioned that "Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit-card information from someone living in a cardboard box to someone living on a park bench." [22]. The reason being that it is far easier to subvert the endpoint than the channel.

Leveraging novel hardware security features, like Intel SGX, makes it significantly harder to subvert the endpoint. Still, moving the endpoint into an enclave is insufficient. What is needed is a way to bind the endpoint to a particular execution context, i.e., an enclave. Otherwise, the system is susceptible to relay attacks, where a compromised endpoint claims to have certain properties, but the properties actually pertain to a third system also controlled by the attacker.

Previously work in this direction [7] falls under the general umbrella of Trusted Computing [3] and focused on Trusted Platform Modules as the ultimate trust anchor. Intel SGX enables a more performant and versatile solution compared to a resource constrained TPM. Hence, it is worth revisiting the problem of secure channel binding in the context of Intel SGX.

In [6] the authors introduce an attestation extension to TLS. The peers exchange attestation information during the handshake. They used provisions within TLS to incorporate the additional information [23]. This particular TLS extension seems to be superseded by HelloExtensions messages in more recent versions of TLS. We decided to embed the additional channel binding information into the certificate to minimize changes to the TLS library. Introducing new messages [24], would have required changes to the library itself. Extending the certificate validation logic is achieved by calling TLS library hook functions from the application. It does not require any changes to the TLS library, making it easier to maintain and port to new TLS libraries. Complex interactions between existing extensions already plague the overall security of TLS [25].

Armknecht et al. [1] also identified the channel binding problem as crucial to the value remote attestation provides. Armknecht et al. propose a generic protocol that combines key exchange and remote attestation to avoid relay/masquerading attacks. However, they do not integrate the generic protocol into any existing secure communication protocol like TLS or IPsec. We integrated our solution with three existing open-source TLS libraries, making it directly usable to applications building on those libraries.

In subsequent work [26], Armknecht et al. propose alternative protocols to address performance concerns with software that frequently needs to provide attestations. Since our work is based on Intel SGX, as opposed to a TPM, these performance concerns do not apply. Attestation is cheap and its overall cost dominated by the communication latency to IAS. Since challengers do not contact the attestation service there is no attestation-related communication overhead added to the TLS handshake. The information required to assess the channel binding properties are embedded in the certificate and the challenger can verify them locally.

7. CONCLUSIONS

Intel SGX offers a unique opportunity to perform secure computation in otherwise untrusted environments. An integral part of Intel SGX is the ability to obtain an attestation on the properties of the enclave and its platform. Integrating remote attestation seamlessly with a standard secure channel protocol greatly simplifies the use of remote attestation in practice. We developed a library that conveniently encapsulates the attestation flow and verification behind a simple API. Using this interface, developers can rely on the added assurance remote attestation provides their application without having to deal with the intricacies of implementing it correctly.

A proof of concept implementation of remote attestation integrated into the TLS handshake is available at https://github.com/cloud-security-research/sgx-ra-tls

Acknowledgements

The authors thank Vincent R. Scarlata and Daniel E. Smith for comments on earlier drafts of this paper.

8. BIBLIOGRAPHY

[1] F. Stumpf, O. Tafreschi, P. Röder, C. Eckert and others, "A robust integrity reporting protocol for remote attestation," in *Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, 2006.

[2] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc'approach to authenticated Diffie-Hellman and its use in the IKE protocols," in *Annual International Cryptology Conference*, 2003.

[3] "The Trusted Computing Group," [Online]. Available: https://trustedcomputinggroup.org.

[4] *TPM Main Specification Level 2 Version 1.2, Revision 116,* Trusted Computing Group(R), 2011.

[5] I. Anati, S. Gueron, S. Johnson and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[6] F. Armknecht, Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, G. Ramunno and D. Vernizzi, "An Efficient Implementation of Trusted Channels Based on Openssl," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2008.

[7] K. Goldman, R. Perez and R. Sailer, "Linking Remote Attestation to Secure Tunnel Endpoints," in *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2006.

[8] "Intel® Software Guard Extensions Programming Reference," Oct 2014. [Online]. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[9] "Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation," [Online]. Available: https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf.

[10] "Intel® Software Guard Extensions SDK for Linux* OS," [Online]. Available: https://download.01.org/intel-sgx/linux-2.0/docs/Intel_SGX_SDK_Developer_Reference_Linux_2.0_Open_Source.pdf.

[11] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan and J. K. Zinzindohoué, "Implementing and proving the TLS 1.3 record layer," in *IEEE Symposium on Security and Privacy (SP)*, 2017.

[12] "The (Decentralized) SSL Observatory," [Online]. Available: http://www.usenix.org/events/sec11/tech/slides/eckersley.pdf.

[13] "The Transport Layer Security (TLS) Protocol Version 1.2 (RFC 5246)," [Online]. Available: https://tools.ietf.org/html/rfc5246.

[14] A. Baumann, M. Peinado and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, 2016.

[16] C.-C. Tsai, D. E. Porter and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[17] S. Shinde, D. Le Tien, S. Tople and P. Saxena, "PANOPLY: Low-TCB Linux Applications with SGX Enclaves," in *Network and Distributed System Security Symposium*, 2017.

[18] "wolfSSL," [Online]. Available: https://www.wolfssl.com/.

[19] "OpenSSL," [Online]. Available: https://www.openssl.org.

[20] "mbedTLS," [Online]. Available: https://tls.mbed.org.

[21] R. Barnes, J. Hoffman-Andrews and J. Kasten, "Automatic Certificate Management Environment (ACME), internet draft (work-in-progress)," [Online]. Available: https://tools.ietf.org/html/draft-ietf-acme-acme-08.

[22] G. Spafford, "The Risks Digest 19.37," [Online]. Available: http://catless.ncl.ac.uk/Risks/19.37.html.

[23] "TLS Handshake Message for Supplemental Data (RFC 4680)," [Online]. Available: https://tools.ietf.org/html/rfc4680.

[24] "Transport Layer Security (TLS) Extensions: Extension Definitions (RFC 6066)," [Online]. Available: https://tools.ietf.org/html/rfc6066.

[25] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti and P. Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *IEEE Symposium on Security and Privacy*, 2014.

[26] F. Stumpf, A. Fuchs, S. Katzenbeisser and C. Eckert, "Improving the Scalability of Platform Attestation," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2008.

[27] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.