Chuchotage: In-line Software Network Protocol Translation for (D)TLS

Nikbakht Bideh, Pegah; Paladi, Nicolae

Published in:
Proceedings of the 24th International Conference on Information and Communications Security (ICICS'22)

DOI:
10.1007/978-3-031-15777-6_32

2022

Document Version:
Peer reviewed version (aka post-print)

Link to publication

Citation for published version (APA):
Nikbakht Bideh, P., & Paladi, N. (2022). Chuchotage: In-line Software Network Protocol Translation for (D)TLS. In *Proceedings of the 24th International Conference on Information and Communications Security (ICICS'22)* (pp. 589-607) https://doi.org/10.1007/978-3-031-15777-6_32

Total number of authors:
2

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Chuchotage: In-line Software Network Protocol Translation for (D)TLS

Pegah Nikbakht Bideh$^{1[0000-0001-7756-3723]}$ and Nicolae Paladi$^{1,2[0000-0003-0132-857X]}$

Lund University, Lund, Sweden[1]
CanaryBit, Sweden[2]
{pegah.nikbakht_bideh, nicolae.paladi}@eit.lth.se

Abstract. The growing diversity of connected devices leads to complex network deployments, often made up of endpoints that implement incompatible network application protocols. Communication between heterogeneous network protocols was traditionally enabled by hardware translators or gateways. However, such solutions are increasingly unfit to address the security, scalability, and latency requirements of modern software-driven deployments. To address these shortcomings we propose Chuchotage, a protocol translation architecture for secure and scalable machine-to-machine communication. Chuchotage enables in-line TLS interception and confidential protocol translation for software-defined networks. Translation is done in ephemeral, flow-specific Trusted Execution Environments and scales with the number of network flows. Our evaluation of Chuchotage implementing an HTTP to CoAP translation indicates a minimal transmission and translation overhead, allowing its integration with legacy or outdated deployments.

Keywords: Protocol conversion · IoT · Application layer protocols · Software Defined Networking · TLS· Cross-Layer Optimization

1 Introduction

Despite efforts towards standardization and interoperability, many applications use proprietary protocols and incompatible data models for information exchange [25]. This is particularly acute to address in growing density of connected embedded devices or "things". Such devices are increasingly expected to communicate in a machine-to-machine (M2M) pattern. Communication among devices, or between devices and back-end systems that use incompatible protocols can be enabled through *protocol translation*. This is commonly realized either with hardware translators, virtual gateways[1], or distributed software applications [34]. Existing approaches for protocol translation are unfit to address the scalability, latency, and security requirements of current and emerging deployment topologies [7]. Such solutions display at least one of the following limitations.

[1] Communication servers including a virtual gateway to perform protocol translation.

1. in-line translation solutions do not support encrypted network traffic;
2. solutions to circumvent limitation (1) rely on deploying trusted certificates to unprotected devices on the network path and increase the attack surface;
3. cloud-based protocol translation solutions support translation over secure network communication by terminating TLS connections in a single centralized component. This increases communication latency between network endpoints and introduces a single point of failure.

Addressing the above challenges is a prerequisite to enable wide-scale device connectivity. This requires support for secure and fast in-line software network protocol translation of encrypted traffic; support for communication over several application layer protocols while maintaining latency requirements; and finally support for distributed protocol translation. Our goal is to enable secure massive M2M communication using protocol translation capable of dynamically adapting to new devices and network topologies. Our **contributions** are as follows:

- we introduce Chuchotage[2], an efficient and secure protocol translator architecture addressing scalability, latency, and security requirements of large-scale networks; Chuchotage builds on earlier work in Software Defined Networking, Trusted Execution Environments, and TLS interception;
- Chuchotage performs in-line protocol translation while supporting secure distributed network communication throughout the network fabric, avoiding translation in a logically or physically centralized back-end;
- we introduce flow-specific, on-demand translator boxes created by software switches on the network path for TLS interception and protocol translation.
- we integrate secure protocol translation in OpenFlow [22] by reusing and extending its signaling. This allows to maintain backward compatibility.

Our solution relies on three principles: (i) secure TLS interception with the use of TEEs; (ii) high-performance confidential protocol translation, and (iii) fault-tolerant distributed architecture with the help of SDN networking. A TEE provides confidentiality and integrity with the use of an isolated execution environment. The loaded code and data to the TEE can be protected from various attacks. In our architecture, we use TEEs to securely decrypt, translate data, and re-encrypt it with a high level of confidentiality and integrity.

In SDN networking, network intelligence is logically centralized, thus abstracting the network infrastructure from network applications [16]. In SDN, the controller has a global view and can decide what suits best for the network. The OpenFlow protocol is usually used in SDN to link the controller and other components, e.g. switches, and routers. OpenFlow is compatible with both hardware and software switches. In Chuchotage, the software switch (Open vSwitch [31] in our implementation) makes informed decisions on application layer protocol translation to provide a high-performance and fault tolerant architecture. To the

[2] The term *chuchotage* is a form of interpreting where the linguist is near a small target audience and whispers a simultaneous interpretation of what is being said.

best of our knowledge, this is the first work that integrates datapath flow matching with secure protocol translation. To improve the performance, we introduce a cross-layer optimization for switch actions described in Section 4.

The rest of this paper is structured as follows: in Section 2 we introduce the relevant background and problem, followed by a review of the related work in Section 3. We describe the design of Chuchotage in Section 4. We discuss in Section 5 the design choices of the Chuchotage implementation, followed by performance and security evaluation in Section 6. We conclude in Section 7.

2 Background

We define interoperability in IoT networks as the capability of heterogeneous devices and applications to communicate and exchange data or services. Tolk et al. presented interoperability as a layered model with two main layers: *technical* and *semantic* interoperability [30]. *Technical* interoperability enables compatibility of heterogeneous devices through common communication protocols and standards. *Semantic* interoperability enables heterogeneous services and applications to exchange information in a meaningful way [1].

Data or information models used by heterogeneous IoT devices are often incompatible, thus limiting semantic interoperability. Semantic protocol translators are a possible solution; they are able to convert information formats, allowing communication between heterogeneous endpoints. Such translators ingest a standardized way of representing vocabularies of processes or messages. However, despite ongoing efforts for IoT semantic translation, we are yet to see a unified secure platform compatible with most common IoT protocols. We next briefly introduce several interoperability solutions.

Physical gateways: A traditional way of interoperability is the use of hardware gateways that act as an intermediate component between endpoint devices [25]. Hardware gateways can translate protocols with different standards and specifications, they are commonly one-to-one protocol translators that do not scale (new protocols require adding new hardware); moreover, they require special hardware connectors, thus increasing both the overhead and complexity.

Protocol translators: Protocol translators replace traditional interoperability solutions, such as gateways; they are intermediate components that perform direct protocol to protocol translation. Depending on where the translation is done, protocol translators are either: a) cloud back-end translators or b) middleboxes. In the first case, the traffic is re-routed to the cloud back-end for translation. In the second case, a middlebox is a hardware component or software network function placed on the communication path between the endpoints.

We review existing protocol translators in Section 3.1. These translators either do not consider security or do not scale. Some perform the translation below the application layer, thus adding further network complexity.

For further information about common IoT protocols and different interoperability solutions at different protocol layers refer to Appendix A.1. We propose the Chuchotage architecture to enable protocol interoperability on the application layer. We target the application layer as it has the highest impact on application performance [12].

2.1 Threat Model

Our threat model considers two aspects - security of the network communication, and security of protocol translation. We assume the Dolev-Yao model [9], with an adversary capable of intercepting, and synthesizing any message, being only limited by the constraints of the cryptographic methods used. Considering protocol translation, we assume limited physical access to the platform, akin to the tasks of a legitimate third party user, and excluding physically modifying, probing, or monitoring the system. The adversary is capable of exploiting software vulnerabilities in the host operating system and software network components (network switch and network functions), reloading the switch binary, accessing the host memory, and starting arbitrary processes on the host. The attacker may modify any firmware of software component on the network platforms, including the hypervisor for virtualized set-ups. This threat model is aligned with the threat models of both process-based trusted execution environments (such as Intel SGX [3] or Keystone [27]) as well as virtualization-based trusted execution environments (AMD SEV-SNP [2], Intel TDX [33] and IBM PEF [14]). The Chuchotage architecture may be tuned to use other TEE implementations. Considering the growing diversity of TEE implementations [33] and their various approaches to defending or preventing side-channel attacks, we exclude side-channel attacks. Likewise, we exclude attacks on control-plane components of SDN deployments (such as the SDN controller) or ancillary components (such as the Certificate Authority); these components are trusted and attacks on them can be prevented using best-practice operational security. Translator boxes are not trusted and translation cannot be done securely without a TEE.

3 Related work

3.1 Protocol Translation

An early work on protocol conversion was presented in 1988 by Lam [17], proposing a formal model to achieve interoperability between processes with different protocols. Its' limitation was that it needs to be implemented as a process or as a low layer protocol in the physical layer, thus adding complexity and overhead to the network.

In [7], the authors proposed a protocol translator for industrial IoT protocols. They proposed the use of an intermediate format in order to translate more than three protocols rather than direct protocol-to-protocol translation. The solution satisfies interoperability features including transparency, scalability, reporting, verifiability, and QoS, however without addressing any security aspects, which Chuchotage explicitly addresses.

Muppet [32] is an edge-based multi-protocol switching architecture that can be used for IoT service automation. Muppet is a P4-based switch which can communicate with IoT devices using different protocols, where switches are managed by an SDN controller. Muppet was designed for translation between Zigbee [26] and Bluetooth low energy (BLE) [11] protocols or translation between BLE/Zigbee and IP protocols and is therefore complementary to Chuchotage, which works at the application layer. However, similar to [7], Muppet does not support protocol translation over TLS communication.

3.2 TLS Interception

HTTPS interception is implemented for purposes such as content filtering, malware detection, DDoS mitigation, load balancing, etc [5], and despite the relative maturity of the topic, research on TLS interception proxies gained further attention in recent years. The ME-TLS protocol [20] supports TLS 1.3 and enables endpoints to introduce middleboxes into a session given the consent of both parties. Endpoints can control middlebox access permissions on traffic data, and verify the middlebox service chain. The protocol is based on monitoring handshake messages passively without modifying the handshake of TLS 1.3. An implicit version negotiation mechanism in the ME-TLS handshake protocol enables it to interoperate with TLS endpoints seamlessly. However, ME-TLS requires deploying the Boneh–Franklin identity-based encryption (BF-IBE) [15] instead of the widely adopted Public Key Infrastructure (PKI) approach.

maTLS is an extension to TLS that allows middlebox visibility and auditability by enabling a client to authenticate all middleboxes through a dedicated *middlebox certificate*. The use of middlebox certificates eliminates the insecure practice of installing custom root certificates or servers sharing their private keys with third parties. Furthermore, the middlebox-aware TLS (maTLS) protocol enables auditing the security behaviors of middleboxes [19].

IA2-TLS [4] is an encryption-based approach to enable in-line packet inspection. IA2-TLS is based on binding an *inspection key* to the random nonces that are generated by the endpoints during a TLS handshake. The advantage of this approach is the capacity to introspect traffic both inline and offline, at any location along the network path. This approach requires modifying the client and server TLS implementation. Similar to many other TLS interception approaches, it is not practical considering the lack of backward compatibility.

Considering the properties and backward compatibility of the ME-TLS protocol, we use it for the remainder of this paper as the reference TLS interception protocol. Other approaches to TLS interception are complementary.

4 Chuchotage Protocol Translator

4.1 Architecture

Figure 1 illustrates the Chuchotage architecture, relying on principles introduced in Section 1: (i) secure and protocol-compliant TLS interception; (ii) efficient confidential protocol translation; and (iii) fault-tolerant distributed architecture. The proposed architecture assumes that network switches are configured 🅐 with an action to translate network flows between endpoints that use incompatible application layer network protocols 🅑 (we use OpenvSwitch for implementation). When invoked, the action triggers the creation of a translator box 🅒 in a trusted execution environment (Intel SGX in our implementation). The translator box is subsequently attested by a verifier network function and provisioned with credentials for TLS interception 🅓. The translator box is network-flow specific, translates subsequent communication between the endpoints 🅔 and terminates once the network flow is cleared from the switch flow table, as described next.

Dynamic Translator Box Creation We use TEEs to run *translator boxes* that decrypt the TLS traffic on the respective flow, use application protocol translators to convert it to the target protocol, and re-encrypt it before forwarding. A translator box is instantiated whenever the `translation` action is triggered by a new network flow matching the flow table rule. Depending on the implementation, translator boxes are instantiated either as a child process of the switch daemon (in-switch) or external to the switch. In-switch translator boxes are instantiated by the ovs-vswitch daemon, while external translator boxes are instantiated by the network controller. Translator boxes are deployed in TEEs to ensure execution isolation, confidentiality, and integrity of packet data.

To instantiate a translator box, the parent process first invokes the creation of a TEE and deploys the translation logic configured for the pair of application layer protocols in the respective network flow. Next, a *verifier* network function attests the integrity and authenticity of the translator box [6]. Following a successful attestation, a trusted certificate authority network function provisions the cryptographic artifacts necessary for intercepting the TLS communication between endpoints. The exact artifacts depend on the approach for TLS interception, as described next in Section 4.1. The parent process of the translator box terminates it once the respective flow is evicted from the datapath cache.

In our current implementation, we used Intel SGX enclaves to create TEEs. SGX enclaves rely on a trusted computing base of code and data loaded at enclave creation time. Program execution within an enclave is transparent to the underlying operating system and other mutually distrusting enclaves running on the platform. The CPU is an enclave's root of trust; it prevents access to the enclave's memory by the operating system and other enclaves. Library operating systems were used in this context to facilitate both the portability and performance of legacy applications in SGX enclaves [27].

TLS Interception We focus on the TLS v1.2 [8] and v1.3 [10] for transport security due to their wide adoption. We further use the ME-TLS [20] protocol extension for TLS interception in protocol translator boxes. The use of ME-TLS allows delivering session key materials to translator boxes in-band and does not require additional TLS connections or round-trips. Moreover, this allows retaining backward compatibility with TLS 1.3 [10] through implicit protocol

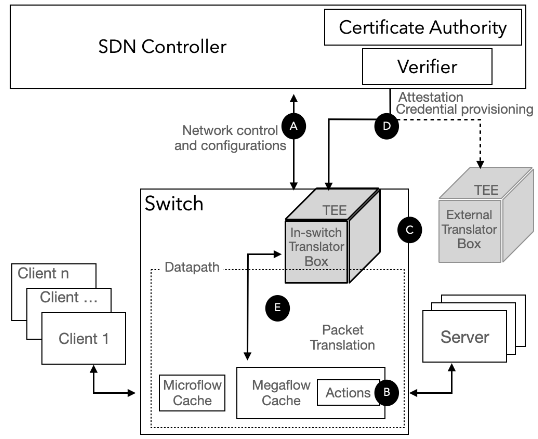


Fig. 1: Conceptual illustration of the Chuchotage architecture

version negotiation. In case one of the endpoints does not support ME-TLS, communication remains encrypted but without protocol translation.

Following the TLS1.3 specification [10], ME-TLS reuses the TLS 1.3 `Finished` message to achieve two additional goals, endpoint authentication and translator box negotiation (agreement between client and server about the translator boxes to be used). For middlebox negotiation, the `ClientFinished` and `ServerFinished` messages each contain two middlebox lists specifying the translator involved in each direction of the network path. Once both client and server endpoints complete the translator box negotiation by including the list of chosen translator boxes to the `ClientFinished` and `ServerFinished` messages, they distribute the necessary session key materials to selected translator boxes. ME-TLS achieves this through an additional `SessionKeyDistribution` message sent by the endpoints to the translator boxes on the communication path. The `SessionKeyDistribution` message is an application data message (not a handshake message); the record field of the message contains a byte sequence, which is an HMAC generated from the shared secret between the client and server (ss_{cs}^{ibe}) and a string constant to differentiate from other application data records, followed by encrypted session key materials for the translator boxes. The ME-TLS protocol uses a property of the BF-IBE scheme [15] that allows endpoints and translator boxes to establish a shared secret between each other through zero-round secret negotiation. In BF-IBE, a trusted authority called a private key generator (PKG) generates private keys for endpoints and translator boxes using their identities and a master key. The endpoints (client and server) can then use the shared secret to encrypt the session key materials communicated to the translator box instances.

Translator Box Integration with OvS Translator boxes are created following the *translate* action in the flow rules and are instantiated during the transport layer protocol handshake between two communicating endpoints, regardless of the application layer protocol they use. An incoming packet to the switch is first matched against the available rules (see Appendix A.2). A match against a rule that contains the *translate* action on the datapath triggers the creation of a flow-specific translator box. The translator box can be created either on the datapath in kernel space or user space, depending on the TEE implementation. When using Intel SGX, translator boxes are created in user space enclaves, since SGX enclaves can only run as user processes. While this may affect their performance (due to IO penalties inherent to the Intel SGX model), recent work indicates that modifying software network components deployed in TEEs can help to improve their IO performance [29]. Next, a verifier network function of the network controller attests the enclave to make sure it is trustworthy, then the enclave receives the key shares through key provisioning that allows it to compute session key materials and decrypt the TLS communication between the endpoints in the respective flow Figure 1. Attestation and key provisioning are done *in parallel* with the ongoing transport layer protocol handshake. All subsequent packets in the respective flow will be processed by the translator box.

Protocol to Protocol Translation Once a translator box inside the enclave receives a packet from the respective flow, it first decrypts the packet using the session key materials computed from the key shared received from the network controller. Next, the translator box parses the decrypted packet, extracts the application data, and formats it into the destination protocol format. Finally, the formatted packet is re-encrypted and returned to the switch data path to be forwarded to its destination.

4.2 Challenges

The design of Chuchotage addresses several important challenges, namely enabling distributed protocol translation and combining TLS interception with attestation primitives of the trusted execution environments. We address distribution and scalability by introducing the concept of ephemeral, flow-specific, on-demand translator boxes created by software switches on the network path. To achieve scalability in high density networks, multiple switches, and SDN controllers can be used in the network depending on the network topology and available resources. Chuchotage combines the ME-TLS protocol for TLS interception [20] with the SGX attestation protocol to provide an uninterrupted chain of trust that includes the communicating endpoints, the translator box, and the certificate authority by the communicating parties.

4.3 Operating flow

In the following operating flow description, we assume that a network administrator uses a deployment blueprint to define flow rules for the endpoints included in the topology. For the types of devices and communication protocols known beforehand, the network administrator specifies a `translate` action for the flows that require translation. Note that two distinct translation policies will be specified for each source-destination pair in a flow where endpoints implement distinct application layer protocols. In the following operating flow description, we assume the latest version of TLS, version 1.3; while other TLS versions can be made compatible with this operating flow, this requires additional adjustments.

In-line operating flow The sequence diagram in Figure 2 illustrates how translator boxes instantiated by the switch obtain the session keys negotiated between two endpoints, client and server:

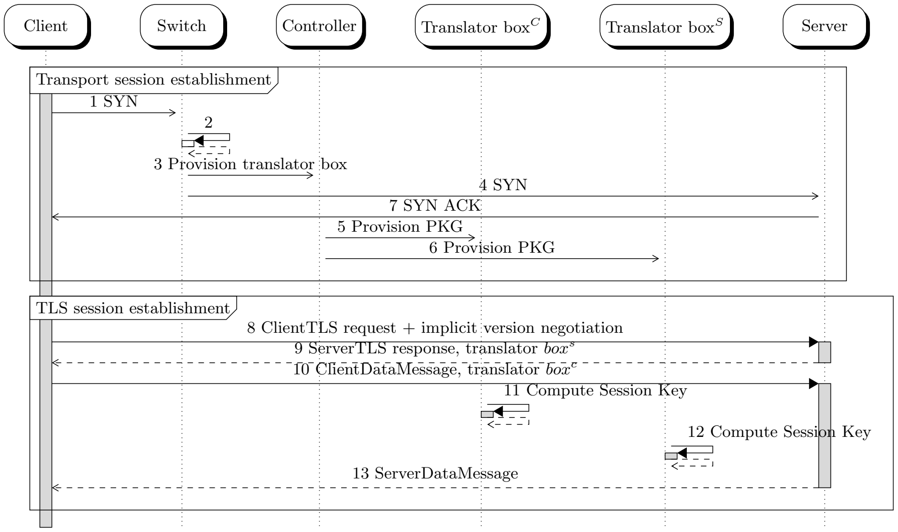

Client
Switch
Controller
Translator box^C
Translator box^S
Server
Transport session establishment
1 SYN
2
3 Provision translator box
4 SYN
7 SYN ACK
5 Provision PKG
6 Provision PKG
TLS session establishment
8 ClientTLS request + implicit version negotiation
9 ServerTLS response, translator *box^s*
10 ClientDataMessage, translator *box^c*
11 Compute Session Key
12 Compute Session Key
13 ServerDataMessage

Fig. 2: Chuchotage operating flow

- The Client initiates a communication session by sending a TCP `SYN` packet to Server (step 1).
- A Switch on the network path matches the `SYN` packet against entries in its Microflow cache. Since the Client did not communicate with the Server earlier, the search continues in the Megaflow cache and ultimately in the OpenFlow flow tables, where it matches the translation policy defined by the network administrator (step 2). The results of Megaflow cache lookup will be cached in Microflow cache. The switch triggers the controller to instantiate the translator boxes (step 3).
- The `SYN` packet is immediately forwarded to the destination; this avoids introducing additional latency (step 4).

- The controller instantiates translator boxes for the flows t_c (client-server, step 5) and t_s (server-client, step 6). The controller instantiates the translator box in a TEE, attests it [6, 3] and provisions key shares generated by the PKG [15].
- The server returns a `SYN ACK` reply, the transport session is established at this point (step 7).
- The TLS negotiation starts; the negotiation follows the TLS 1.3 with the ME-TLS extensions [20] (step 8). The Client TLS request includes an implicit version negotiation to check that the Server supports the ME-TLS extensions. The Server TLS response follows the TLS 1.3 specification and additionally specifies the identifier of the server translator box (step 9).
- Next the Client starts sending encrypted application data (step 10).
- The `ClientDataMessage` packet containing application data is matched in the Microflow cache of the switch and processed by translator box t_c. At this point, t_c obtains its session key material from the `SessionKeyDistribution` message and generates the key distribution bytes using the shared secret between itself and the endpoints (step 11). It derives the application traffic secrets, allowing it to derive symmetric keys to encrypt and decrypt application data on the client-server path. The session key is used for the remainder of the TLS session.
- Having decrypted the data, t_c converts the application data to Server application protocol format, re-encrypts it, and forwards the packet to the Server;
- The Server returns the application data encrypted with a TLS session key. The `ServerDataMessage` application data packet is matched in the Microflow cache of the Switch and processed by translator box t_s; t_s obtains its session key material from the `SessionKeyDistribution` message, generates the key distribution bytes using the shared secret between itself and the endpoints, and derives the application traffic secrets allowing it to derive symmetric keys to encrypt and decrypt application data on the server-client path. The session key is used for the remainder of the TLS session (step 12);
- t_s converts the decrypted application data to the client's application protocol format, re-encrypts it and forwards it to the client (step 13);
- Translation of application data continues for the remainder of the TLS session; the translator boxes are terminated once the network flow is evicted from the Switch flow cache.

In case of DTLS, the operating flow is modified such that the translator boxes are created after the `ClientHello` message.

5 Implementation

For evaluation purposes, we implemented Chuchotage with two popular IoT protocols, CoAP and HTTP. Our implementation includes the following components. A *client*, an HTTP client representing an IoT device contacts a server with a different protocol, a *Server*, A CoAP server is listening for client connections.

Open vSwitch (OvS): endpoints are connected to OvS through the same bridge and OvS is responsible for forwarding incoming client or server packets to the translator box, as well as forwarding outgoing packets from the translator box to their destinations; *SDN controller*: an SDN controller manages the network flows to improve network performance. For that we used Ryu[3], an open source controller. Whenever OvS does not find any matching entry in its flow caches to handle packets in need of translation it contacts the controller, which will trigger a translation. *Translator box*: via the translation process, the controller creates a translator box responsible for translating the traffic between client and server. In the translator box, we used an HTTP to CoAP parser/formatter library[4], capable of parsing and converting HTTP to CoAP messages and vice versa. *TEE*: to ensure execution isolation as well as confidentiality and integrity during packet translation, we ported the protocol translator to an SGX enclave using the Occlum library OS [27]. Occlum[5] is a memory-safe library OS for SGX. Note that for implementing other protocol translation (other than CoAP and HTTP), a new parser/formatter is required but the rest of the components will remain unchanged.

5.1 Implementation choices

In Chuchotage, the translator box can be instantiated either by the network controller (external) or OvS (in-switch [28]). In our prototype implementation, the SDN controller deploys an SGX enclave with the translator code and attests it, as deploying, managing, and debugging external translators is easier for network administrators. Attestation can be done locally or remotely based on the location of the appraiser and of the target enclave [6]. In our prototype implementation, the SDN controller (appraiser) and translator box (target) both exist on the same platform and hence we used local attestation with a trusted enclave that exists on the SDN controller and keypair provisioning. As mentioned above, the TEE hosting the translator box can be instantiated using several alternative approaches, both virtualization-based [33] or process-based [18, 21]. Enterprise deployments should consider remote attestation of translator boxes, or a combination of both as supported by some virtualization-based TEEs [14]. The choice of TEEs depends on constraints on application portability, security, performance, etc.

For TLS interception, we assume that session key materials are distributed to the involving parties including the client, server, and SDN controller prior to the handshake procedure and ME-TLS overhead is explicitly excluded in our evaluation since it only affects the handshake, not the actual communication.

Our translation policy is defined by using features extracted from the traffic flow, namely a combination of specific source and destination IP addresses and port numbers. When an incoming flow matching these features triggers the

[3] https://ryu-sdn.org/

[4] https://github.com/keith-cullen/FreeCoAP

[5] https://github.com/occlum/occlum

translation action and the packets in the matching flow are forwarded to the translator. After translation, the packets are sent back to the switch to be forwarded to their own destination. While distinct translator boxes can be created for inbound and outbound flows (client to server or server to client, see Section 4.3), we use one translator box for both in- and outbound flows.

5.2 Testbed

Our testbed consists of four docker containers representing client, server, a Ryu controller, and a translator box deployed in an SGX enclave (see Figure 3), as it can be seen in Figure 3 the testbed is compatible with different pairs of clients and servers. OvS was installed on the host OS and the four docker containers are connected to the OvS via one bridge (`br0` in Figure 3). Each container is connected to the bridge through its own virtual interface, indicated as `vethp` in Figure 3. Whenever a flow needs to be translated, Ryu creates and attests an SGX enclave inside container 3. The translation is done inside the enclave and the flow to be translated is afterwards forwarded through container 3.

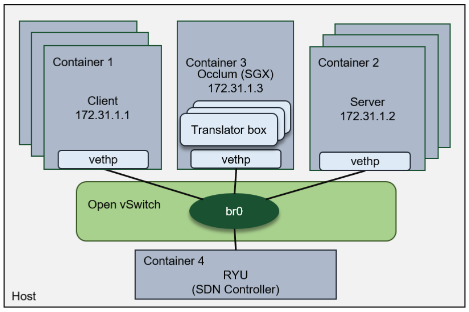


Fig. 3: Testbed Overview

6 Evaluation

6.1 Performance Evaluation

We conducted several tests to evaluate the performance of Chuchotage. In the first test, we send packet batches of different sizes (100, 1000, and 10000 packets) from the client to the server and measured the translation time for the entire batch. We also measure the transmission time, i.e. the time between sending the first and last packets excluding the handshake. In this test, the client sends empty `HTTP GET` messages translated to CoAP confirmable `Reset` messages.

We measured translation and transmission time both with and without SGX, to measure the effect of the TEE on the performance (see Figure 4). Without a

TEE, the translator box is created inside container 3 in Figure 3. As illustrated in Figure 4, both translation and transmission times slightly increase with the use of a TEE (Intel SGX in this prototype); however, this increase is acceptable in most IoT networks considering the added benefit of protecting network traffic confidentiality. Error bars are based on standard deviation.

We also compared our results with the transmission time of a vanilla CoAP to CoAP communication. Confirmable CoAP Reset messages were sent from a CoAP client to the CoAP server. The transmission time for transferring 100, 1000, and 10000 packets respectively are: 0.00719, 0.07428, and 0.70909 seconds. We consider these values as a reference point for the added overhead by the translation procedure compared to a vanilla CoAP to CoAP transfer.

In a second test, we send batches of 100 packets of different sizes (128, 256, and 512 Bytes) from client to server and record their translation and transmission time with and without using a TEE. In this test, we send `HTTP POST` requests from the client to the server and they are translated to CoAP confirmable `POST` requests. The results of this test show that using a TEE (Intel SGX in this prototype) results in increasing both the translation and transmission time (see Figure 5). Packet data length does not affect the translation time.

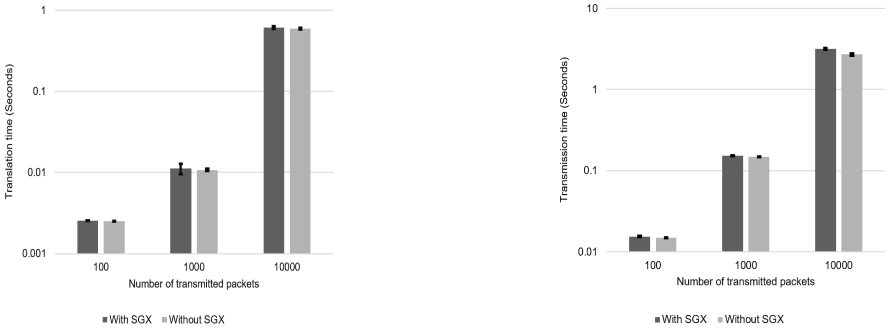
In a third test, we measured the time to complete a successful handshake. The handshake takes place between the client, server, and translator box; however, the translator box is transparent for the client and server. The overall handshake time (an average of 10 handshakes) including local attestation (0.0164 seconds), enclave creation (0.80410 seconds), and additional communication between the Chuchotage components averages 2.83574 seconds. This is roughly equal to transferring and translating 10000 packets; a vanilla CoAP to CoAP handshake averages to 0.000907 seconds. However, the handshake is only performed once before translating all subsequent packets in the flow.

The performance of our proposed protocol translator is not comparable to centralized approaches, such as gateway or proxy-based approaches, since they are not suitable for large heterogeneous distribution deployments and often do not consider security of network traffic. Chuchotage is not also comparable to other existing protocol translation solutions, as earlier highlighted in Section 3.1.

6.2 Security Evaluation

Reflecting the structure of the threat model (Section 2.1) we discuss the security of network communication and of protocol translation.

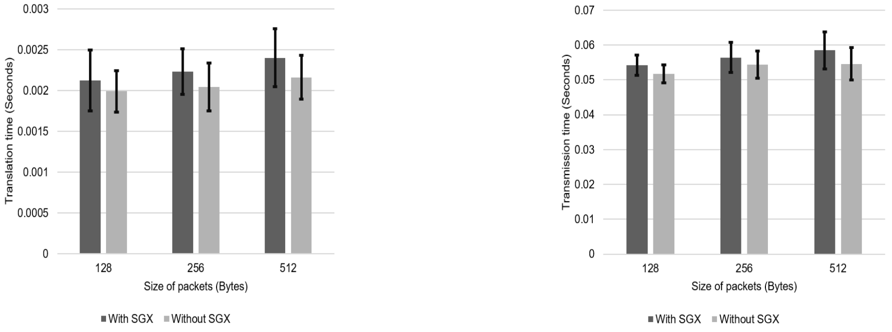
Network Security Chuchotage uses TLS 1.3 [10] to implement transport layer security - including key establishment - and inherits its confidentiality and integrity properties. On the other hand, Chuchotage also inherits any potential vulnerabilities yet to be discovered in TLS 1.3 ; this underscores the importance of following vulnerability management best practices. The security of ME-TLS extensions to TLS 1.3 is reviewed in detail in [20]. There are several types of network based attacks that can target Chuchotage, such as Denial of Service (DoS)



(a) Translation time

(b) Transmission time

Fig. 4: Translation and Transmission Time of Translating Different Number of Packets



(a) Translation time

(b) Transmission time

Fig. 5: Translation and Transmission Time of Translating Different Packet Sizes

or traffic flooding. Similar to other contexts, DoS attacks can be mitigated by DoS prevention techniques including intrusion detection and prevention systems, using load balancers, filtering, etc.

Protocol Translation Availability of a Chuchotage deployment can be ensured through network deployment best practices. High availability is an inherent capability of Chuchotage as translator boxes are instantiated and deployed in TEEs by switches throughout the network topology.

Translator boxes are central to the security of protocol translation and network communication in Chuchotage. Integrity of the protocol translation software deployed in translator boxes is verified through attestation [6]. The chain of trust evaluated through attestation is specific to the platform implementation of the TEE. During protocol translation confidentiality of provisioned cryptographic material and intercepted network traffic is ensured through TEE isolation mechanisms that include memory isolation on hardware or firmware level, run-time memory encryption, and cache flushing upon execution transition [33].

In our current prototype implementation, we use Intel SGX enclaves as a TEE implementation target. SGX is vulnerable to a wide category of attacks reviewed in [24]. Chuchotage can be vulnerable to any attacks applicable to SGX. However, there are a number of mitigation techniques that can be used to mitigate attacks on realistic applications deployed in SGX enclaves [13].

7 Conclusion

In this paper, we proposed Chuchotage, an in-line application layer protocol translator with transport layer security. Chuchotage relies on secure TLS interception, efficient protocol translation, and fault-tolerant distributed architecture. In Chuchotage we translate, and re-encrypt network flows with minimal latency, on the network path. Scalability is guaranteed by growing the number of translator boxes with the number of flows; translator boxes are instantiated by individual software network switches in the deployment. Depending on the capabilities of the underlying platform and their support for TEEs, Chuchotage allows creating translator boxes either in-switch or external to the switch, in kernel space or user space. We implemented a Chuchotage prototype for HTTP to CoAP translation with Intel SGX enclaves and Open vSwitch. Our evaluation indicates a slight increase in the translation and transmission time. This overhead depends primarily on the choice of TEE in the implementation.

Acknowledgments

This work was financially supported in part by the Swedish Foundation for Strategic Research, with the grant RIT17-0035, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

1. Semantic Integration & Interoperability Using RDF and OWL. https://www.w3.org/2001/sw/BestPractices/OEP/SemInt/ (2005), [Online; accessed 15-October-2020]
2. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, Advanced Micro Devices (January 2020)
3. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proc. of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13, p. 7. ACM New York, NY, USA (2013)
4. Baek, J., Kim, J., Susilo, W.: Inspecting TLS Anytime Anywhere: A New Approach to TLS Interception. In: Proc. of the 15th ACM Asia Conference on Computer and Communications Security. pp. 116–126 (2020)

5. de Carné de Carnavalet, X., van Oorschot, P.C.: A survey and analysis of TLS interception mechanisms and motivations. arXiv e-prints pp. arXiv–2010 (2020)
6. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. International Journal of Information Security **10**(2), 63–81 (2011). https://doi.org/10.1007/s10207-011-0124-7
7. Derhamy, H., Eliasson, J., Delsing, J.: IoT interoperability—on-demand and low latency transparent multiprotocol translator. IEEE Internet of Things Journal **4**(5), 1754–1763 (2017)
8. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (Aug 2008). https://doi.org/10.17487/RFC5246, `https://www.rfc-editor.org/rfc/rfc5246.txt`, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919
9. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. on information theory **29**(2), 198–208 (1983)
10. Eric Rescorla: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). https://doi.org/10.17487/RFC8446
11. Garbelini, M.E., Wang, C., Chattopadhyay, S., Sumei, S., Kurniawan, E.: SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 911–925. USENIX Association (Jul 2020), `https://www.usenix.org/conference/atc20/presentation/garbelini`
12. Gregg, B.: Systems Performance, 2nd Edition. Pearson, London (2020)
13. Hosseinzadeh, S., Liljestrand, H., Leppänen, V., Paverd, A.: Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In: Proc. of the 3rd Workshop on System Software for Trusted Execution. pp. 42–47 (2018)
14. Hunt, G.D.H., Pai, R., Le, M.V., Jamjoom, H., Bhattiprolu, S., Boivie, R., Dufour, L., Frey, B., Kapur, M., Goldman, K.A., Grimm, R., Janakirman, J., Ludden, J.M., Mackerras, P., May, C., Palmer, E.R., Rao, B.B., Roy, L., Starke, W.A., Stuecheli, J., Valdez, E., Voigt, W.: Confidential computing for openpower. p. 294–310. EuroSys '21, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3447786.3456243
15. Kate, A., Goldberg, I.: Distributed private-key generators for identity-based cryptography. In: Garay, J.A., De Prisco, R. (eds.) Security and Cryptography for Networks. pp. 436–453. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proc. of the IEEE **103**(1), 14–76 (2014)
17. Lam, S.S.: Protocol conversion. IEEE Trans. on Software Engineering **14**(3), 353–362 (1988)
18. Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: An open framework for architecting trusted execution environments. In: Proc. of the Fifteenth European Conference on Computer Systems. EuroSys '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387532
19. Lee, H., Smith, Z., Lim, J., Choi, G., Chun, S., Chung, T., Kwon, T.T.: maTLS: How to Make TLS middlebox-aware? In: NDSS (2019)
20. Li, J., Chen, R., Su, J., Huang, X., Wang, X.: ME-TLS: Middlebox-Enhanced TLS for Internet-of-Things Devices. IEEE Internet of Things Journal **7**(2), 1216–1229 (2020). https://doi.org/10.1109/JIOT.2019.2953715
21. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. Hasp@ isca **10**(1) (2013)

22. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (Mar 2008). https://doi.org/10.1145/1355734.1355746, `https://doi.org/10.1145/1355734.1355746`
23. Medina, J., Paladi, N., Arlos, P.: Protecting OpenFlow using Intel SGX. In: 2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). pp. 1–6. IEEE (2019)
24. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel SGX. arXiv preprint arXiv:2006.13598 (2020)
25. Noura, M., Atiquzzaman, M., Gaedke, M.: Interoperability in internet of things: Taxonomies and open challenges. Mobile Networks and Applications **24**(3), 796–809 (2019)
26. Safaric, S., Malaric, K.: Zigbee wireless standard. In: Proc. ELMAR 2006. pp. 259–262 (2006). https://doi.org/10.1109/ELMAR.2006.329562
27. Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., Yan, S.: Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In: Proc. of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 955–970. ASPLOS '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378469
28. Svenningsson, J., Paladi, N., Vahidi, A.: Faster Enclave Transitions for IO-Intensive Network Applications. In: Proc. of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure. p. 1–8. SPIN '21, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3472873.3472879
29. Svenningsson, J., Paladi, N., Vahidi, A.: SGX-Bundler: speeding up enclave transitions for IO-intensive applications. In: The 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. IEEE-Institute of Electrical and Electronics Engineers Inc. (2022)
30. Tolk, A.: Composable mission spaces and m&s repositories–applicability of open standards. In: Spring simulation interoperability workshop, Arlington (VA) (2004)
31. Tu, W., Wei, Y.H., Antichi, G., Pfaff, B.: Revisiting the open vswitch dataplane ten years later. In: Proc. of the 2021 ACM SIGCOMM 2021 Conference. p. 245–257. SIGCOMM '21, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3452296.3472914
32. Uddin, M., Mukherjee, S., Chang, H., Lakshman, T.: SDN-based multi-protocol edge switching for IoT service automation. IEEE Journal on Selected Areas in Communications **36**(12), 2775–2786 (2018)
33. Yao, J., Zimmer, V.: Virtual Firmware, pp. 459–491. Apress, Berkeley, CA (2020). https://doi.org/10.1007/978-1-4842-6106-4˙13
34. Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M.: Internet of things for smart cities. IEEE Internet of Things journal **1**(1), 22–32 (2014)

A Appendix

A.1 Common IoT Communication Protocols

In the TCP/IP network model, the physical or data link layer is responsible for physical transmissions; characteristics of applications - such as latency and availability - directly impact traffic characteristics on the link layer. The network layer is responsible for routing and forwarding packets; considering that IoT devices are often resource-constrained, the information necessary for routing should be kept at a minimum. Finally, transport layer protocols (such as TCP and UDP) manage end-to-end communication between network endpoints.

Physical network gateways are commonly used for interoperability in the physical and network layers or transport layer [25]. Gateways have limited scalability [25]: as the number of IoT devices increases, special connectors are required for their interaction, thus adding both cost and complexity to the network.

Application communication between network endpoints is implemented on the application layer. Middleware can perform translation in the application layer; however, connecting middleware components risks further reducing interoperability by locking applications to a specific technology. Interception proxies are an alternative for application layer translation; however, proxies cause delays since all traffic transits through proxies even when translation is unnecessary [7].

Proxies and middleware currently available for application layer protocol translation are increasingly unsuitable for secure, distributed, and transparent application layer protocol translation.

Several application layer protocols - namely HTTP, CoAP, MQTT, and AMQP - have been widely reviewed in academic publications and adopted in large scale deployments. We compare these protocols in Table 1.

Table 1: IoT protocols comparisons

IoT protocols	HTTP	CoAP	MQTT	AMQP
Transport layer	TCP	UDP	TCP	TCP
Security	TLS/SSL	DTLS	TLS/SSL	TLS/SSL
Architecture	Req/Res	Req/Res	Pub/Sub	Pub/Sub
QoS	No	Yes	Yes	Yes
Low Power/Lossy Networks	Fair	Excellent	Fair	Fair
Dynamic discovery	No	Yes	No	No

A.2 Open vSwitch Overview

OpenvSwitch (OvS) is an open source programmable switch [31] that implements packet forwarding on the datapath; it is a flow-based switch, where clients install flows determining forwarding decisions. Flows are installed in a cache level structure that assists the datapath to execute actions on received packets, e.g. allow, drop, etc. For each ingress packet, the datapath consults its cache and forwards the packet to its destination if matching entries exist. For each cache miss, the datapath issues an upcall and forwards the packet to ovs-vswitchd. A datapath can be deployed as a kernel module or in user space with additional firmware support. Packet classification in OvS is computationally expensive, mostly due to the many types of matching fields. Matching is implemented in a hash table of flow rules, with matching fields hashed as keys. OvS uses a modified Tuple Space

Search (TSS) algorithm for packet classification. The algorithm searches through the hash map tables based on the maximum entry's priority and terminates after finding the highest priority matching flow rule. Early OvS releases implemented OpenFlow processing exclusively as a kernel module. However, the difficulty of developing and updating kernel modules motivated moving packet classification to user space. A multi-level cache structure kernel implementation compensates the resulting performance impact. The cache structure consists of two levels with increasing lookup costs: a microflow cache (or Exact Match Cache) and a larger megaflow cache. The megaflow cache matches multiple flows with wildcards [23].

Open vSwitch Forwarding Figure 6 illustrates the OvS internals. An incoming packet reaches the datapath from either a physical or virtual NIC (1). In the datapath, the switch runs a first search based on an exact match (2). If there is a matching entry in the microflow cache, the packet is sent to the specific table in the megaflow cache to retrieve the required actions. Otherwise, the forwarding process performs a second search in the next cache line (3). Failing to find a match, the datapath uses upcalls (4) to inform the ovs-vswitchd that it cannot handle the packet. The ovs-vswitchd uses the classification process (5) to obtain a matching rule via its flow tables. Next, ovs-vswitchd returns to the datapath, inserts the entry in the cache (6), and returns the packet to the kernel (7). Finally, the datapath forwards the packet to the intended destination (8). Failing to find matching information in the flow tables, ovs-vswitchd sends a packet-in request to the network controller to get a matching rule for the unknown packet.

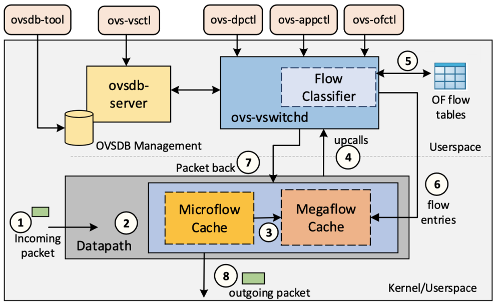


Fig. 6: An overview of Open vSwitch internals