D2TLS: Delegation-based DTLS for Cloud-based IoT Services

Eunsang Cho
Seoul National University, Korea
escho@mmlab.snu.ac.kr

Minkyung Park
Seoul National University, Korea
mkpark@mmlab.snu.ac.kr

Hyunwoo Lee
Seoul National University, Korea
hwlee2014@mmlab.snu.ac.kr

Junhyeok Choi
Seoul National University, Korea
jhchoi2015@mmlab.snu.ac.kr

Ted "Taekyoung" Kwon
Seoul National University, Korea
tkkwon@snu.ac.kr

ABSTRACT

The Internet of Things (IoT) becomes proliferated due to the advances in embedded devices, wireless communications, and cloud technologies. However, the security problem in the Internet will be worsened in IoT services considering the constrained resources of IoT devices. We propose a delegation-based DTLS framework (D2TLS) for cloud-based IoT services. D2TLS aims to achieve mutual authentication and to lower the burden of setting up secure connections significantly while keeping the private keys of the IoT devices secret. Leveraging the session resumption in the DTLS standard and introducing a security agent, D2TLS achieves these goals while requiring the modifications only on the client side. That is, the cloud and PKI systems need not change to deploy D2TLS. Numerical results show that D2TLS can achieve better performance in terms of delay and energy consumption than the current DTLS protocol in standalone mode.

CCS CONCEPTS

• **Security and privacy → Security protocols**; • **Networks →** *Network experimentation*;

KEYWORDS

Delegation, DTLS, TLS, Internet-of-Things, Cloud Service

ACM Reference Format:
Eunsang Cho, Minkyung Park, Hyunwoo Lee, Junhyeok Choi, and Ted "Taekyoung" Kwon. 2019. D2TLS: Delegation-based DTLS for Cloud-based IoT Services. In *Proceedings of International Conference on Internet-of-Things Design and Implementation, Montreal, QC, Canada, April 15–18, 2019 (IoTDI '19),* 12 pages.
https://doi.org/10.1145/3302505.3310081

1 INTRODUCTION

The Internet of Things (IoT) is poised to provide connectivity to almost every electronic device so that the devices can be monitored and/or controlled in an automatic fashion. While the IoT is expected to help realize convenient and smart lives by tapping devices that have not been online, the security issues in the current Internet may be worsened due to the limited capabilities of the devices. Designing a trustworthy networking framework will be essential for secure IoT services. The prior frameworks for IoT networking over the last few years have focused on the limited resources, and thus they have considered proxy-based approaches, which is not suitable with the end-to-end principle of the Internet. A vast majority of IoT devices may be constrained in their capabilities; they may periodically go to sleep mode (e.g., to save energy); the transmission rate is too slow (e.g., ZigBee and LoRa); or the computing power is not sufficient to handle many requests from clients. In such cases, perhaps their sensory data will be stored to reverse proxy nodes, which will reply to the requests on behalf of the devices.[1] However, proxy-based IoT services have the following problems: (i) an IoT device may have to share its security information (e.g., its private key) with its reverse proxy, which is the well-known key escrow problem, if it allows the proxy to set up a security association with remote clients [7], and (ii) the sensory data is also shared between the device and the reverse proxy, which indicates another vulnerability by violating the end-to-end principle, not to mention the data tampering issues.

Nowadays, the cloud-based IoT services have become widespread, instead of proxy-based ones. For example, a survey reported that 34 out of 39 IoT platforms take cloud-based or centralized architectures [9]. The reason why the IoT and the cloud get along with each other is two-fold: (i) resource-constrained IoT devices are hardly capable of processing incoming queries as standby servers, (ii) it is more practical for such constrained devices to send data in an on-demand fashion to cloud systems depending on their capability and availability (say, transmit their sensory data while they are awake), and then the cloud will service the queries from clients. We assume that it would be the norm that the cloud systems service IoT requests on behalf of the resource-constrained devices, which corresponds to the device-to-cloud and back-end data sharing models in IoT communications models [21].

Even though cloud-based IoT networking and services are proliferated [6], we believe a security framework for such settings is not well provisioned yet [18]. The insecurity of smart IoT devices has been reported from Symantec with 50 different devices [1] and from the measurement study with 28 devices [19]. While 68 percent of the tested devices rely on cloud services, their control and data messages may be exchanged over untrusted networks [1]. In particular, around 19 percent of the tested devices communicate

[1]We assume a reverse proxy keeps the data of the corresponding devices, and is located within the same administrative domain.

ACM ISBN 978-1-4503-6283-2/19/04...$15.00
https://doi.org/10.1145/3302505.3310081

without encryption, e.g., Transport Layer Security (TLS), and none of the devices provides mutual authentication [1]. [19] investigated 28 devices and found that 39 percent of the devices do not use TLS for communication. The report [1] argues that strong encryption and mutual authentication be required even for resource-limited IoT devices, and recommends efficient cryptographic methods such as elliptic curve cryptography (ECC).

Standards developing organizations (SDOs) seek to develop new networking platforms or frameworks for IoT services. However, as to security, they usually adopt the current Internet security framework like TLS and Datagram Transport Layer Security (DTLS). For instance, the DICE working group in the IETF proposes to use TLS/DTLS profiles [5] for IoT deployments with CoAP [2], which is a lightweight version of HTTP, and the CoRE working group updates CoAP for TCP, TLS, and WebSockets [3]. The ETSI OneM2M consortium also suggests to use the TLS or DTLS [11]. Using DTLS implies that the current authentication mechanisms like the public key infrastructure (PKI) may have to be used in IoT environments. However, due to typical IoT constraints like the energy budget, hardware capability, and/or low speed communications, IoT devices may not be able to handle the PKI and certificates timely. While the DTLS is lighter than TLS, it may still incur the substantial overhead on IoT devices, which will be investigated in this paper.

In the same vein as the above Symantec report [1], we claim that mutual authentication be crucial for secure IoT services. Considering the limited capabilities of IoT devices, we propose a delegation-based DTLS framework (D2TLS) for cloud-based IoT services. The central idea of D2TLS is to leverage the session resumption functionality in the certificate-based DTLS standard [14]. D2TLS also introduces a security agent to relieve a device of the setup burden of DTLS connections. That is, the security agent sets up a secure connection with the cloud system on behalf of the device. In D2TLS, only the device is allowed to keep its private key (unlike [7, 10]) while performing mutual authentication of two endpoints. By keeping the private key within IoT devices and delegating heavy computations to agents, D2TLS solves the private key escrow problem and provides an end-to-end connection model between a device to the cloud.

2 BACKGROUND

The full DTLS handshake may impose a serious workload on the two endpoints. To help mitigate the overhead, DTLS provides a session resumption mechanism by which the same negotiated secret can be resumed if the two endpoints already have set up a DTLS connection. Thus, it is conceivable for a resource-constrained IoT device to delegate the first full handshake to some other entity, while the device performs only the task of taking over the security context for the DTLS session by exploiting the DTLS session resumption.

2.1 Delegation schemes for IoT devices based on DTLS 1.2

Prior delegation schemes [7, 10] usually think of an IoT device as a server. For the resource-constrained IoT devices, the availability as a server could be enhanced by using delegation, which means the IoT device (as a server) is not required to prepare for the incoming connections all the time. The management of the security context is also handled by delegation. However, we target cloud-based IoT services, which means IoT devices operate as clients and cloud systems provide "always-on" availability as servers. More detailed analyses are shown as follows.

[7] introduced a delegation server for IoT devices; the delegation server is responsible for the first full handshake on behalf of the devices. Right after the first handshake is done, the delegation server transfers the session ticket to the remote client, which then resumes the DTLS session with the corresponding IoT device (i.e., a server). Their scheme is proposed for highly resource-constrained devices. Thus, the whole handshake process is done by the delegation server, which is called "full delegation." During the delegation process, all the secret data are transferred via pre-configured out-of-band secret exchanges. The weakest point of this scheme comes from the full delegation. Trusting the delegation server by sharing the private keys of the devices means granting full permissions (e.g. security-related operations of the device) to the delegation server.

[10] targets medical sensor networks, where the delegation process is performed at a gateway of the sensor network. In this scheme, the medical sensors are servers, and hence remote clients connect to the sensors directly. The sensors assumed in their work are also highly resource-constrained, and thus they delegate the first DTLS handshake to the gateway like [7]. However, [10] is different from [7] in the following points. First, the gateway is within the same administrative domain as the sensors; thus, it is easier to secure the communications between them compared to [7], where the delegation server can be located outside the domain. Next, only the indispensable data is transferred to the sensor since it may not have the capability of keeping all the session-related information. However, [10] has the same problem as [7] since the private key of a device is shared with the gateway.

2.2 Prevention of Secret Information Sharing for Delegation

On the web, the delegation is used for enhancing end-user experiences by locating the data nearby the end-user, and the aforementioned problem of sharing the private key also happens here. To establish TLS connections between the delegate and clients, usually three solutions are utilized: (i) private key sharing, (ii) certificate-based approaches, and (iii) using a dedicated key server.

Private key sharing is a similar approach to previously-stated IoT delegation schemes [7, 10]. Certificate-based approaches typically exploit the structure of certificates, which means that inserting multiple domain names into the origin certificate, or making a shared certificate between the origin and the delegate [8]. Whereas CloudFlare, a cloud service provider, devises a scheme in which the cloud server does not need to know the private key of the origin server, so-called 'Keyless SSL' [20]. CloudFlare introduces a dedicated key server, which holds the private key of the origin server. Note that the dedicated key server is controlled by the origin server. By introducing the key server, Keyless SSL achieves the authentication of the original server during TLS sessions even if the cloud server sets up TLS connections with the clients. The idea of keeping the private key in the origin server is adopted in our design, to be detailed later.

2.3 Differences in Session Resumption of TLS 1.2 and 1.3

First of all, the full handshake and session resumption processes of DTLS are the same as those of TLS; thus we compare TLS 1.2 [4] and TLS 1.3 [13] in this section. TLS 1.2 provides two types of session resumption mechanisms: a session ID, and a session ticket [16]. The session ID approach puts the liability of keeping security contexts on the server, whereas the session ticket approach relieves the server of such a burden by encoding the whole security context into the session ticket. Thus, some prior studies [7, 10] promote the latter since they view an IoT device as a server. However, as we assume the cloud-based approach, an IoT device is deemed as a client by default. Hence our approach can adopt both the session ID and the session ticket. For TLS 1.3, the only allowed option is the session ticket with pre-shared keys.

The session resumption of TLS 1.3 provides the forward secrecy by utilizing the key share option, which is not possible in TLS 1.2. Using the key share means that the key exchange and agreement are required for each resumption, causing additional delays in TLS 1.3. To sum up, the session resumption is lightweight in TLS 1.2; however, the forward secrecy is possible only in TLS 1.3.

Our proposal will be substantiated under the DTLS 1.2 environments; however, it could be easily applicable to DTLS 1.3 (Draft 28) [15], which is the latest version at the time of writing. For instance, DTLS 1.3 supports Elliptic-curve Diffie-Hellman Ephemeral (ECDHE) by default, which is also adopted by the proposal. The session ID is no longer supported with DTLS 1.3. However, as the session ticket is supported with DTLS 1.3, the proposal can utilize the session ticket instead of the session ID for DTLS 1.3.

3 MEASUREMENT OF IOT PRODUCTS

To design D2TLS, the communication patterns of the cloud-based IoT services are needed to be analyzed. As DTLS is based on the session concept, an IoT service can easily fit if its session consists of a series of successive packets. In order to analyze the communication patterns of cloud-based IoT services, we measured two commercial IoT products: (i) a smart home monitoring system, and (ii) a smart watch.

3.1 Smart Home Monitoring System

The usage of a smart home monitoring system is manifold: watch against unexpected intrusions, check whether the doors and windows are open or closed, detect any moving objects, monitor a baby, control the temperature and humidity, and so on.

We measure the traffic in a Xiaomi's smart home system, which consists of a door (open/closed) detector, a temperature/humidity monitor, a motion detector, a push button, and a smart gateway. The communication interface of the system is IEEE 802.15.4. We use a TI CC2531 packet sniffer to capture the traffic in the IEEE 802.15.4 network. All the sensor devices are connected to the gateway via IEEE 802.15.4, while the gateway has connectivity to the cloud system via IEEE 802.11 as well as IEEE 802.15.4 for the home Internet. A user can access the sensory data in the cloud system using a smartphone app. Note that there is no encrypted traffic; all the packets carry the plaintext payload.

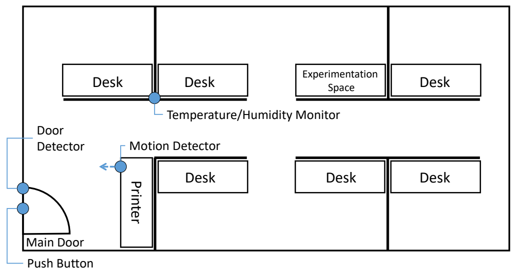


Figure 1: The placement of the sensor devices of the Xiaomi smart home system is depicted.

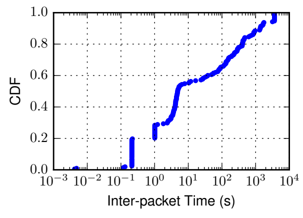
Table 1: Number of packets to be analyzed for each sensor device of the Xiaomi smart home system is shown; the packets are captured over 24 hours.

	Number of Packets
Door Detector	209
Temperature/Humidity Monitor	126
Motion Detector	131
Push Button	45

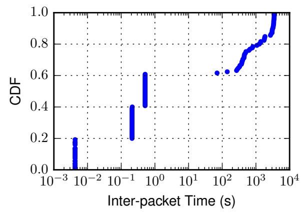
The experiments on the smart home system were performed in our lab office with six people as depicted in Figure 1. The door detector and push button are placed at the main door of the lab, and the motion detector and temperature/humidity monitor are installed at the main corridor inside the lab office. We believe that it mimics usual interactions of a small group of people.

In our measurement analyses, we focus on the distribution of the lengths and the frequency of sessions between the sensor devices and the gateway (towards the cloud) for the following reasons. Compared to consecutively transmitted UDP datagrams (which helps to keep a single session), DTLS sessions require more resources of IoT devices for creating and maintaining sessions. In particular, a session creation consumes more resources than in-session communications due to the heavy computations of public-key operations in the DTLS handshake process. Thus, we believe the overhead of DTLS sessions is dependent on their lengths and frequency.

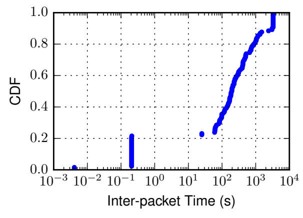
Figure 2 shows the distributions of inter-packet times of the four devices in CDF. Since the DTLS protocol encrypts application-layer packets, we filtered out IEEE 802.15.4 beacon and ACK messages from 6,622 captured packets, and then we used 511 packets for analysis. Table 1 shows the number of packets analyzed for each device. All the subfigures of Figure 2 except 2(c) show that over 50% of data points are distributed close to 0 seconds (less than 10 seconds), which indicates that a majority of packets are generated successively. These successive packets can be considered as traffic belonging to a single session when we apply the concept of DTLS sessions. After the successive packets, the separate sessions can be identified by long inter-packet times. At least, approximately 30% of inter-packet times are longer than 100 s for every graph in Figure 2, which means that there are many separate sessions conceptually within the 24 hours of measurements.



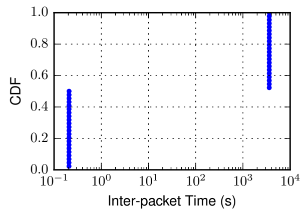
(a) Door Detector



(b) Temperature/Humidity Monitor



(c) Motion Detector



(d) Push Button

Figure 2: The CDFs of inter-packet times of the 4 devices in the Xiaomi home network are plotted.

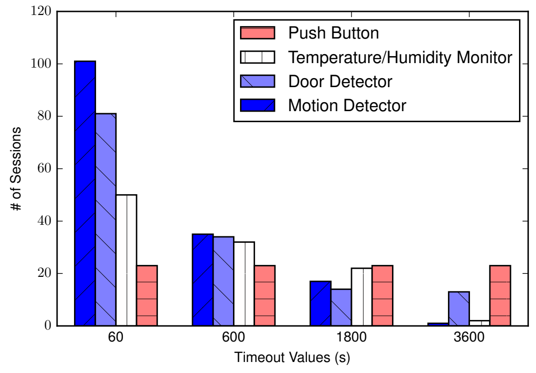


Figure 3: Estimated session frequency of the Xiaomi network for 24 hours is plotted as the timeout value changes.

Let us now estimate the lengths and frequency of separate sessions from the above measurements of the inter-packet times. We seek to find how long a session lasts from the measured plaintext traffic. That is, when we set up a DTLS session, we would like to infer how many consecutive packets will be covered in a single session. Note that there is no concept of sessions when sensors communicate with the gateway in plaintext. Therefore, based on the measured plaintext traffic, the session information/duration is estimated. Figure 3 shows the estimated frequency of sessions as we vary the threshold (of inter-packet time) to separate different sessions. As there is no fixed/standard value for the threshold of session separation, we use the widely-used session timeout values of HTTP and application servers in general Internet services. The timeout thresholds are set to 60s, 600s, 1800s, and 3600s, which are adopted from the minimum and default values of Microsoft IIS servers, the default values of Apache HTTP servers and Apache Tomcat servers, respectively. If we set the threshold to 60s, there are 101 sessions in the motion detector. Obviously, the numbers of estimated sessions decrease sharply as the threshold increases across the four sensors, except for the push button. Considering the constrained resources and high burdens of wireless communications in IoT environments, the session timeout values of IoT devices may as well be shorter than those of HTTP servers in the general Internet environments. Therefore, we believe that the durations of sessions in IoT environments are likely to be close to that of the minimum timeout value (60s) in the graph.

Figure 4 shows the session length distribution when the session timeout value is set to 60s. The means of session lengths are 8.40s, 0.39s, 0.56s, and 0.20s in the four sensors, respectively. However, most of the session lengths are clustered at a few distinct values smaller than 1s. In Figure 4, some of the session lengths are under milliseconds order. It means that there are very short packets whose inter-packet time is larger than the threshold (60s), which can be a default session length in the real D2TLS sessions. The session length distribution reveals that a vast majority of the sessions are short, and hence the overhead of setting up a DTLS connection for each session can incur some substantial workload.

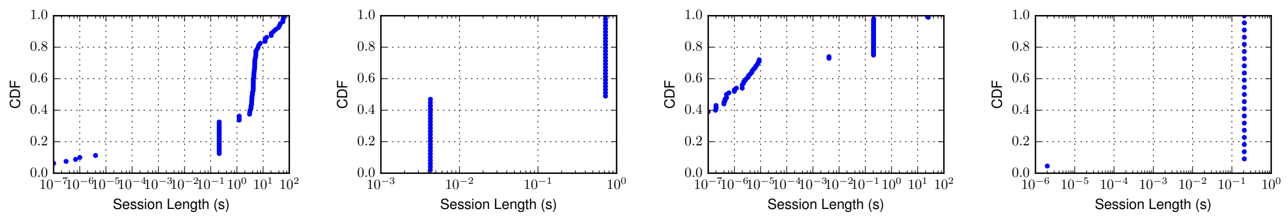
3.2 Smart Watch

A smart watch is a wearable device that is usually connected to a smartphone, which in turn is connected to cloud services via its WiFi or 3G/LTE interface. For example, smart watches using the Android Wear platform require the Internet connection through smartphones to Google services.

We measure the traffic between a Motorola's Moto360 smart watch and a smartphone over Bluetooth. We rely on the Bluetooth HCI snoop feature in the Android OS, so that we can capture all the traffic between the smart watch and the smartphone. The measurement is again focused on the lengths and frequency of communication sessions as described in Section 3.1.

Figure 5 shows the inter-packet times in seconds during the measurement for 24 hours in CDF. The total number of captured packets is 15,202. Since DTLS encrypts packets at the application layer, we filter out messages not containing application data. That is, we exclude Bluetooth signaling messages, which leaves us 2,422 packets for analysis. As shown in Figure 5, most of data points are concentrated close to 0s, and the other data points are scattered between 0s and 1800s. Figure 5 shows a similar tendency to Figure 2, but the concentration level of successive packets is stronger since almost 80% of inter-packet times are shorter than 1s. Note that there are a number of applications running inside the smart watch, which was factory-reset and synchronized before the measurement. For experimental purposes, only a Google account is activated and there is no app installed other than pre-installed ones.

Let us analyze the lengths and frequency of individual sessions. Figure 6 shows the estimated session frequency by varying the threshold of inter-packet time to identify separate sessions. The same timeout values in Section 3.1 are set for session separation. Maximum 88 sessions are identified with the 60s threshold.



(a) Door Detector (b) Temperature/Humidity Monitor (c) Motion Detector (d) Push Button

Figure 4: The CDFs of session lengths of the four sensors in the Xiaomi system for 24 hours are plotted.

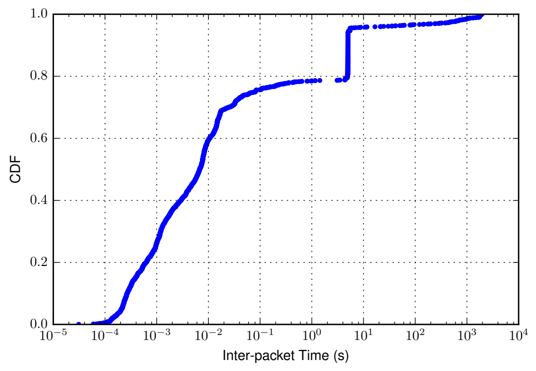


Figure 5: Inter-packet times of a smart watch for 24 hours are shown in CDF.

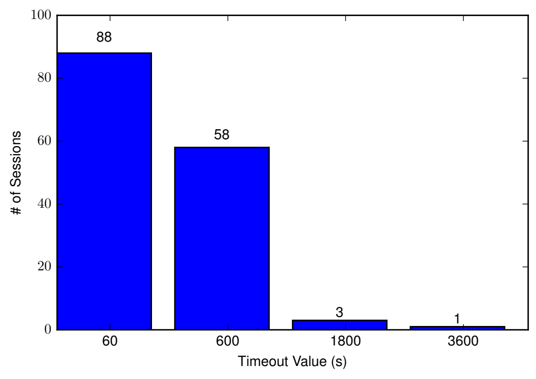


Figure 6: Estimated session frequency of a smart watch for 24 hours is plotted as the timeout value changes.

Figure 7 shows the session length distribution when the session timeout value is set to 60s. The mean of session lengths is 29.25s, but about half of the session lengths are concentrated at a single value: approximately 35s. It shows that the other half of the sessions are shorter, but they are still longer than those of sessions of the sensors in the Xiaomi system.

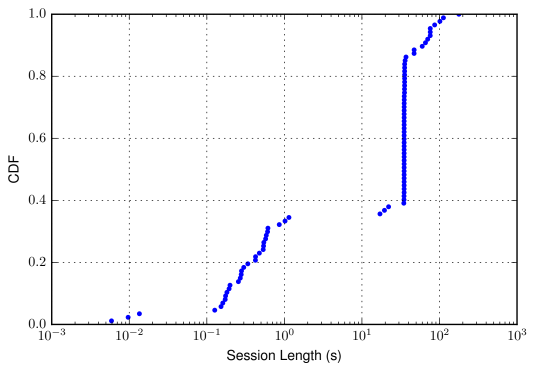


Figure 7: Session lengths of a smart watch for 24 hours are plotted in CDF when the threshold of session separation is 60s.

4 DELEGATION-BASED DTLS (D2TLS)

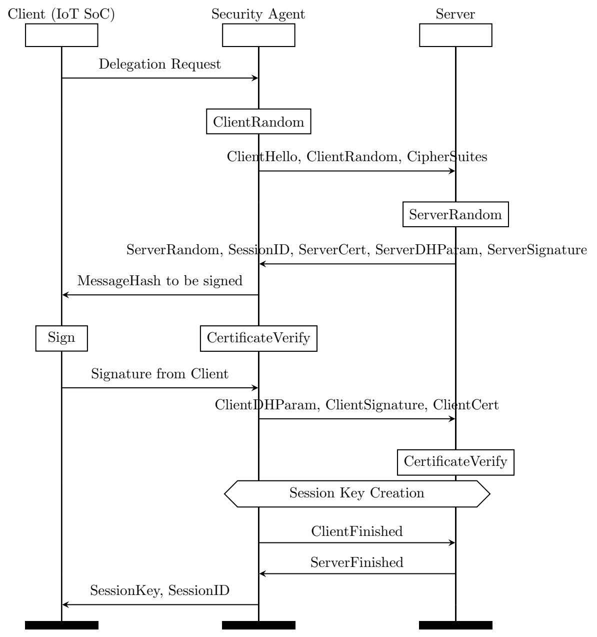
It is revealed that the measured IoT systems based on cloud services typically communicate in short durations as analyzed in Section 3. For instance, in case of condition-triggering sensors, the sensory data would be sent only when an activity/phenomenon of interest is detected. That is, the packet sizes of the sensory data and session durations are usually short.

As mentioned earlier, heavy computations occur when a DTLS[2] connection is created during its handshake process. Our measurement study, however, reveals that the communication patterns of cloud-based IoT services have intermittent and short sessions. Thus, if we have to set up a DTLS connection for each session, the burden of a session creation would be huge. To lower the burden on IoT devices, we leverage the session resumption feature in DTLS and introduce an entity for delegation—a security agent. However, unlike [7, 10], D2TLS does not give the private key of an IoT device to the security agent, and hence there is no key escrow problem.

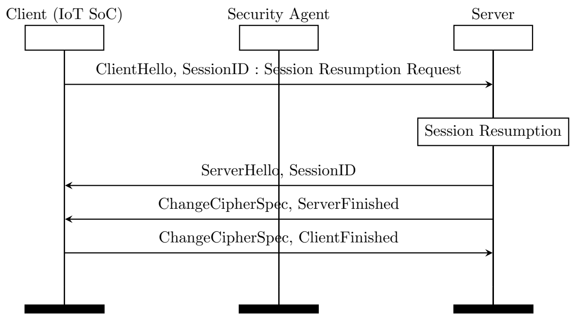
4.1 D2TLS Framework

The goals of the D2TLS framework are (i) to make the private key of an IoT device not be shared with other entities, and (ii) to lower the burden of handling the DTLS handshake on the device significantly.

[2]Even though D2TLS can be applied to TLS as well, we proceed with DTLS for sake of exposition in the remainder of this paper.



(a) Delegation-based full DTLS handshake


Client (IoT SoC)
Security Agent
Server
ClientHello, SessionID : Session Resumption Request
Session Resumption
ServerHello, SessionID
ChangeCipherSpec, ServerFinished
ChangeCipherSpec, ClientFinished

(b) End-to-end session resumption

Figure 8: Message flows for the D2TLS framework and the session resumption for the cloud-based IoT services are depicted.

We assume that the mutual authentication is to be achieved in IoT environments, since automated machine-to-machine communications will be prevalent. That is, human supervisions (i.e., typing ID/password) are almost infeasible in IoT settings. Also, we seek to retain the current DTLS standard as much as possible.

As shown in Figure 8(a), the client part of a DTLS handshake is handled at the security agent (on behalf of the device), which is assumed to be located within the same IoT domain as the device. We assume a secure channel between the device and the security agent before the message flow in Figure 8(a) starts. At first, the IoT device, who wishes to establish a secure session with a remote server in the cloud, sends a delegation request message to the security agent. Then the agent initiates a certificate-based full DTLS handshake with the remote server on behalf of the IoT device. As D2TLS performs mutual authentication, the security agent should create a signature for ECDHE (in Figure 8(a)) on behalf of the device during the handshake. For this, the security agent needs the private key of the device. At this moment, the agent forwards the related information to the IoT device and then the IoT device decrypts/signs the given information. After the device replies back to the agent, the agent resumes the handshake. Other parts in the current DTLS standard remains unchanged. Finally, the agent hands over the session context to the IoT device for the session resumption and wipes out the session context from its memory.

With D2TLS, the IoT device needs to carry out only the decryption and signature generation in terms of computations. All the other tasks are performed by the security agent. As for the remote server, the whole handshake process is exactly the same as the current DTLS standard. The session resumption is not changed from the standard. Modifications are made at the security agent and the IoT device. Therefore the modifications are confined within a local IoT domain, which implies the easy deployability of D2TLS.

4.2 End-to-End Secure Connection by Session Resumption

The end-to-end principle is one of the tenets of the Internet. It is also applied to security-related protocols and mechanisms; the notable example is the authentication of two endpoints. Based on the authentication, DTLS provides confidentiality and data integrity.

There are two ways of resuming a DTLS session. One is the abbreviated handshake [4] and the other is the session resumption without server-side state [16]. The former requires both of the endpoints maintain the session state, and hence the session is resumable only with the session ID. The latter issues a session ticket including its session state, and the session is resumable with the session ticket, which removes the overhead of maintaining the session information (on the server side). A device in D2TLS takes the former approach and works as a client in cloud-based IoT environments, which means the number of connections would be limited by the number of the counterpart servers in the clouds. Moreover, the size of the session ticket for a connection is up to 64 KB, whereas a session ID takes up to 32 bytes. For TI CC2538 device in our experiments in Section 6, which has 32 KB of RAM, the session ticket approach does not fit to the RAM in the worst case, however, the session ID approach does fit. For TI CC3200, which has 256 KB of RAM, both approaches can be applied. Thus, we believe it is not a significant burden that maintains a small number of connections on an IoT device with the session ID approach. Also, the session ID is much smaller than the session ticket, which is suitable for low bitrate wireless networking.

As shown in Figure 8(b), the device sends a ClientHello using the session ID to the server. After matching its session cache with the session ID, the server replies back a ServerHello with the same session ID if the server is willing to re-establish the connection. If the session ID is not matched, the server replies back a ServerHello with a new session ID, and the IoT device gives up this resumption attempt and restarts a new session as described in Section 4.1. The other parts of the current DTLS standard remain unchanged.

Overall, the D2TLS framework makes the device keep its private key, and lowers the burden of setting up a secure connection significantly. By leveraging the session resumption, D2TLS holds the end-to-end communication/security model. The processing overhead for keeping the private key will be investigated in Section 6.

5 SECURITY CONSIDERATIONS

We assume an adversary who controls one of the entities; a security agent, a cloud server, or an IoT device. The adversary's goal is to perform an impersonation attack or to eavesdrop the traffic by using the compromised entity. As the security mechanisms of either the cloud server or the IoT device are based on DTLS, we focus on the threat analyses of the security agent which is a unique entity of D2TLS.

Compromised security agent: If an adversary compromises a security agent (of an IoT device), she may seek to act as the device and to establish a connection with the server in the cloud. As the mutual authentication is required for D2TLS, the private key of the IoT device is essential for the connection establishment. However, as the adversary does not have the private key of the device, she cannot set up a D2TLS connection by impersonating the device.

Another attack can take place as follows. The adversary can obtain the security context (of a device) in the agent if the agent has been compromised by the adversary. From the context (e.g. master secret), the adversary may be able to exchange DTLS messages with the cloud server. However, this impersonation attack of the adversary can be detected by the IoT device and/or the server since both of the endpoints will check the DTLS epoch and the sequence number at the record layer. The epoch is a counter value that increments whenever the cipher changes. The sequence number is reset to 0 when the new epoch starts, and increments for each record. If the adversary does not know the next sequence number exactly (i.e., not being a gateway node on the path), the server could detect the impersonation. If the adversary knows the next sequence number (i.e., being a gateway node), the IoT device will fail to resume the previous session and hence the masquerading attack can be detected afterwards. Thus it is important to keep track of the DTLS epoch and sequence number explicitly, which is standardized in [14]. Also, the security context should be discarded at the security agent after a handshake considering that the agent might be compromised later (in Section 4.1).

The operator of the IoT domain should watch the security agent in order to check whether the agent is compromised. If the security agent exchanges traffic with the cloud server even if the DTLS connection setup (on behalf of the device) is over, it is suspected that the (compromised) agent may try to impersonate the device. It is desirable to monitor and analyze the communications of the security agent with the entities outside the domain.

For passive eavesdropping attacks, the compromised security agent can eavesdrop the traffic if it is on the path (say, a gateway) between the device and the server. To thwart such attacks, we can place the agent out of the data path. Even if the agent is on the path, TLS 1.3 provides the forward secrecy by updating the encryption key at every resumption as described in Section 2.3.

Compromised cloud server: If an adversary compromises a server in the cloud, the adversary could gain access to security contexts which were established previously. Then the adversary could resume any connection with the device since the session resumption process does not require re-authentication by default. Also the private key of the server could be taken by the adversary in this case. However, this attack can occur regardless of the proposed scheme. To mitigate the problem, the lifetime of the session should be made shorter. Thus there is a tradeoff between the security against the cloud compromise and the D2TLS connection setup overhead, which is also the same for DTLS without delegation.

Table 2: Comparison of IoT Devices under Evaluation

SoC	TI CC3200	TI CC2538
Microcontroller	ARM Cortex-M4 @ 80MHz	ARM Cortex-M3 @ 32MHz
RAM	256KB	32KB
Flash Memory Storage	1MB	512KB
Radio Transceiver	IEEE 802.11b/g/n 2.4GHz	IEEE 802.15.4 2.4GHz

Compromised IoT device: If an adversary compromises the device, the adversary could access its private key. Therefore the compromised device should be sanitized and the new public and private key pair should be re-generated. Again, this attack could exist regardless of the delegation.

6 EVALUATION

We compare D2TLS with the current DTLS handshake performed by a standalone device (i.e. without delegation).

6.1 Evaluation Environments

D2TLS is implemented for numerical evaluations whose test setting consists of IoT clients, a security agent, a cloud server, and an OCSP server[3].

To investigate the feasibility and performance of D2TLS, we use two products for IoT clients: TI CC3200 SoC and TI CC2538 SoC as shown in Table 2. The security agent is co-located with a gateway to the Internet. The agent is a machine with Intel Core i5-4690 CPU at 3.5GHz clock speed and 8GB of RAM. The agent is connected to a TI CC3200 device through a WiFi AP. Or the agent uses a USB interface to connect to a TI CC2538 node, which in turn is connected to another TI CC2538 node via IEEE 802.15.4.

We consider LAN and WAN environments between the IoT domain (i.e., the device and security agent) and the cloud/OCSP server. In LAN settings, the cloud server and OCSP server are co-located in a machine equipped with Intel Xeon E3-1245v3 CPU operating at 3.4GHz and 16GB of RAM. The IoT clients and the security agent are located in the same room; the cloud server and OCSP server are located in the other building in the same campus network. The average RTT between the security agent to the cloud/OCSP server is 1.366ms.

In WAN environments, the cloud server and OCSP server are virtual machines (VMs), each of which has 8 virtual cores of Intel

[3]An online certificate status protocol (OCSP) server keeps track of the validity of certificates in charge.

Table 3: Delay of a full DTLS handshake at an IoT device with 256-bit ECC key is shown in LAN environments (in ms).

Setting		Full Handshake	Session Resumption
TI CC 3200	DTLS	2,920	57
	D2TLS	851 (Signature: 414)	
TI CC 2538	DTLS	63,161	3,902
	D2TLS	16,748 (Signature: 9,903)	

Xeon E5-2666v3 and 15 GB of main memory. The locations of the IoT clients and security agent are in the campus network in east Asia. However, the location of the VMs is in a data center in Oregon, USA. The average RTT between the security agent to the cloud/OCSP server is 167.888ms.

The settings to implement the certificate-based DTLS handshake are as follows. We use OpenSSL 1.0.1p for creating the certificates and the OCSP server application. For security operations, all the entities rely on WolfSSL 3.7.0 except for CC2538 node. Since the size of the binary WolfSSL is bigger than the available RAM size of CC2538, we use tinyDTLS 0.8.2 and the relic toolkit only for the CC2538 device. The ciphersuite used for evaluation is ECDHE-ECDSA-AES128-CCM-8. Each plot is the average of five measurements, unless stated otherwise.

6.2 Delay

As shown in Table 3, the delay of a full DTLS handshake in LAN environments is measured from two viewpoints: (i) with and without delegation, and (ii) two different IoT devices.

For the TI CC3200 device, the full handshake time is compared between (i) DTLS (TI CC3200 sets up a DTLS connection by itself) and (ii) D2TLS (the security agent sets up a DTLS connection, and then TI CC3200 takes over). The delay of the D2TLS handshake consists of a delegation request from the device to the agent, a delegated handshake at the agent, a signature operation at the IoT device, a transmission of the session context to the device, and a session resumption from the IoT device to the remote server as shown in Figures 8(a) and 8(b). The full handshake in D2TLS takes 851ms, while the one in DTLS takes 2,920ms. Among the delay of the handshake in D2TLS, the signature-related operations take 414 ms, which means the burden of signing operations are about a half of the total delay. The delay for the session resumption of TI CC3200 in D2TLS takes only 57ms. The delay of the full handshake in D2TLS is substantially reduced by 70.9% due to the delegation handshake, and the session resumption takes a short time at TI CC3200.

For TI CC2538, the full handshake in DTLS takes 63,161ms. The full handshake in D2TLS takes 16,748ms and the session resumption takes 3,902ms on average. Recall that TI CC2538 has very limited resources. However, compared to the TI CC3200 results, the initial handshake time with TI CC2538 is not so long, considering the session resumption time. Notice that the reduction ratio (73.5%) of the full handshake time of TI CC2538 is higher than that of TI CC3200 (70.9%). It means that the delegation helps to reduce the delay for resource-limited devices. TI CC2538 has a crypto module for ECC. The signature generation time could be reduced with the crypto module, which will be discussed in Section 7.3.

Table 4: Delay of a full DTLS handshake at an IoT device with 256-bit ECC key is shown in WAN environments (in ms).

Setting		Full Handshake	Session Resumption
TI CC 3200	DTLS	3,581	537
	D2TLS	1,840 (Signature: 415)	

Table 5: Energy consumption of a DTLS/D2TLS handshake in LAN settings at IoT devices with 256-bit ECC key is shown.

Setting		Delay (ms)	Power (mW)	Energy Consumption (mJ)
TI CC 3200	DTLS	2,920	234	683
	D2TLS	851	276	235
	Session Resump.	57	364	21
TI CC 2538	DTLS	63,161	573	36,161
	D2TLS	16,748	591	9,898
	Session Resump.	9,903	592	5,863

The delay of a full DTLS handshake in WAN environments with the TI CC3200 device is shown in Table 4. The full handshake delays are compared between D2TLS and DTLS. The handshake delay is increased by 661 ms for DTLS, and 989 ms for D2TLS, compared to the LAN settings. The larger increase in D2TLS comes from the session resumption, which is not needed in DTLS. The delay for the session resumption is 537 ms, which is increased by 480 ms compared to the LAN settings. Even though the increment of the handshake delay is larger, D2TLS outperforms DTLS about two times faster with TI CC3200.

6.3 Energy Consumption

We measure the average power consumption of the IoT device by using a Monsoon power monitor. As shown in Table 5 in LAN settings, D2TLS consumes slightly more power than DTLS in TI CC3200. However, DTLS consumes approximately 2.9 times of energy than D2TLS, since the delay of DTLS is around 3.4 times longer than that of D2TLS. The third row in Table 5 shows the delay/power/energy incurred during the session resumption part in the full handshake in D2TLS. The session resumption consumes only 21 mJ of energy, and it is about 3% and 9% of energy for D2TLS and DTLS, respectively. The power consumption of the session resumption is higher than the average power consumed during the full handshake process. However, it takes a small portion in contrast to the time in the full handshake, and hence the session resumption is energy-efficient.

The average power and energy consumptions of TI CC2538 during DTLS and D2TLS handshakes show similar tendency with ones

Table 6: Energy consumption of a DTLS/D2TLS handshake in WAN settings at IoT devices with 256-bit ECC key is shown.

Setting		Delay (ms)	Power (mW)	Energy Consumption (mJ)
TI CC 3200	DTLS	3,581	234	838
	D2TLS	1,840	249	458
	Session Resump.	537	285	153

Table 7: Flash Storage and RAM footprints for a full handshake in DTLS and in D2TLS with 256-bit ECC key at IoT Devices are shown in bytes.

Setting	Flash Usage (text+data)	RAM Usage (data+bss)	text +data +bss	Max. Dynamic Memory Usage
TI CC3200 (DTLS)	116,832	47,372	161,892	21,493
TI CC3200 (D2TLS)	99,144	47,388	144,204	15,237
TI CC2538 (DTLS)	77,330	14,253	91,033	1,960
TI CC2538 (D2TLS)	73,550	14,849	87,849	1,148

of TI CC3200 as shown in Table 5. Note that the session resumption takes about 59% of D2TLS energy consumption due to its large delay.

In WAN settings with the TI CC3200 device, the measured power consumptions are slightly reduced compared to the ones in LAN settings, except for DTLS as shown in Table 6. The IoT device may go into the idle/sleep state more frequently when it does not have loads for computations or communications, and that is why D2TLS and the session resumption consume less power than DTLS since they are lightweight in terms of computations and communications.

6.4 Code Size and Memory Requirements

Considering the resource constraints of the devices, the sizes of compiled binary and memory usage should be investigated. As shown in Table 2, TI CC3200 has 256KB RAM, and TI CC2538 has only 32KB RAM. Their flash memory sizes are larger than RAM sizes, respectively. Therefore the major limitation comes from the RAM usage for both static/predefined and dynamic memories.

As shown in Table 7, D2TLS at TI CC3200 takes less flash and dynamic memories, compared to DTLS. The compiled code of D2TLS is reduced since it requires less public-key related operations. TI CC3200 adopts the Energia OS, which uses its dynamic memory mainly for heap space. To find out the maximum usage of dynamic memory, we change the size of the minimum heap space that can make the program run. The total memory requirement of D2TLS is 62,625 bytes.

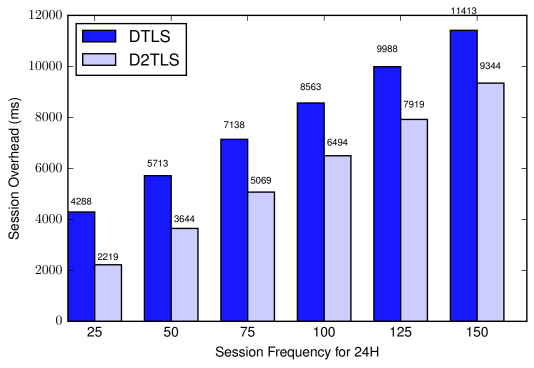


Figure 9: Session overhead for 24 hours as we change the session frequency is plotted in ms.

The memory usage of TI CC2538 cannot be compared with those of TI CC3200 since different DTLS modules and microcontrollers are used. The compiled code of D2TLS is reduced due to the same reason as the TI CC3200 case. TI CC2538 adopts the Contiki OS, which has no memory allocator and heap space but it has a stack space only. The maximum dynamic memory is measured by memory dump at the end of the handshake, by comparing zeroed memory dump before the handshake. The total memory requirement of D2TLS is 15,997 bytes only.

6.5 Session Overhead Depending on Session Frequency and DTLS Context Lifetime

So far, we evaluate the overhead of a single handshake of a session of D2TLS and DTLS. For sake of clarity, we introduce a notion, DTLS context lifetime, which means how long the DTLS connectivity and security context will be maintained in a device or a server. Meanwhile, a session refers to a burst of packets whose inter-packet arrival time is less than a threshold. Usually, a session duration is smaller than the DTLS context lifetime even though both values are configurable. The session overhead for 24 hours with varying the session frequency and DTLS context lifetimes can be calculated based on our measurement results in Section 3.

The session overhead accumulated for 24 hours as we vary the session frequency is plotted in Figure 9. Here, the session frequency means how often sessions appear between the IoT device and the cloud server. The session overhead is calculated as the sum of delays of DTLS session creations (i.e., establishments) and of resumptions. In this experiment, the lifetime of a DTLS context is set to 24 hours and the delays are measured from TI CC3200 with the LAN configuration. Notice that D2TLS has the lower overhead than DTLS due to the shorter handshake delays. However, the relative reduction in overhead is getting smaller as the session frequency grows since the difference in the overhead comes from the handshake delay which is fixed and is related to the lifetime of a DTLS context.

Figure 10 shows the accumulated session overhead for 24 hours as we vary the lifetime of a DTLS context. The session frequency is set to 100 (for the 24 hours) and the delays are measured from TI CC3200 with the LAN configuration. The reduction of the session overhead of D2TLS over DTLS grows as the lifetime of a DTLS

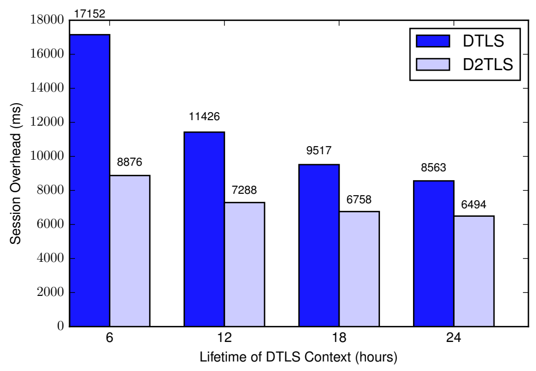


Figure 10: Session overhead varying lifetimes of a DTLS context for 24 hours is plotted in ms.

context decreases since the shorter lifetime brings the more session creation that is substantial to the session overhead. With 6 hours of session ID, the overhead of D2TLS is approximately 48% lower than one of DTLS.

Overall, the accumulated overhead is notably reduced with D2TLS over DTLS. Especially the reduction of the overhead grows with as the lifetime of a DTLS context gets shorter. It is advised to shorten the context lifetime considering the attacks to the session master key. However, the short lifetime incurs the more handshake overhead, which increases inefficiency. D2TLS reduces the first full handshake overhead of a DTLS session, and thus it does not incur much overhead even in the short context lifetime. Therefore D2TLS is suitable for the short lifetime of a DTLS context with the repeated session resumptions in terms of efficiency and security.

7 DISCUSSIONS

7.1 IoT device as a Server

In cloud-based IoT services, an IoT device typically operates as a client. While D2TLS assumes a device to be a client, it can also serve as a server (i.e., the cloud is not needed). If the device operates as a server, a remote client needs to know the server address information. The IETF CoRE group suggests resource discovery mechanisms [2, 17] for this purpose, which can be used in D2TLS.

Using any resource discovery mechanisms, the remote client can find out the IP address of the device/server. The problem is that the device may not be able to perform a full DTLS handshake by itself. Our idea is to use Mobile IPv6. We assume there is a home agent in the same subnet as a device. When the home agent receives a ClientHello message destined to the device, it informs the remote client of the IP address of a security agent by sending a Binding Update message in Mobile IPv6. In this way, the client is ready to receive messages from the security agent. The home agent also delivers the ClientHello message to the security agent as well as to the device. Then the device sends a delegation request message in Figure 8 if it wishes to proceed. Once the security agent receives the delegation request from the device, it will process the ClientHello and continue the full handshake with the remote client. The rest of the handshake and session resumption will be the same.

Table 8: Delay of a full TLS 1.3 handshake at TI CC3200 with 256-bit ECC key is shown in LAN environments (in ms). 'PSK', '0-RTT', and 'ECDHE' mean the preshared key, 0-RTT, and ECDHE option enabled, respectively.

Mode	Handshake	Session Resumption
PSK-ECDHE	4,082	842
PSK	3,880	649
0-RTT-ECDHE	3,904	860
0-RTT	3,879	694

While the session resumption in D2TLS is made by the session ID only (in Section 4), the session ticket extension [16] of the session resumption in DTLS can be considered for server mode operations of the device. This extension allows the client to hold its session context, while the server need not maintain its previous sessions. Therefore it helps an IoT device to act as a server.

7.2 Considering DTLS 1.3 Session Resumption

As the DTLS 1.2 results in LAN environments in Section 6.2, the delay of a full DTLS handshake (2,920ms) is approx. 51 times larger than the delay of a session resumption (57ms) on the TI CC3200 device. However, it is expected that the session resumption of DTLS 1.3 would be much slower than 1.2, because of adopting the preshared key and key share options. As stated in Section 2.3, the terms DTLS and TLS are interchangeably used when we focus on handshaking.

WolfSSL posted that TLS 1.3 has a performance degradation in a resumption handshake, compared to TLS 1.2 [22]. The performance trade-off results from decrypting a session ticket, performing more encryption/decryption, and processing hashing operations in a TLS 1.3 handshake. It is stated that a TLS 1.3 resumption handshake is more than 20% slower when running a client and a server on the same computer. Also, TLS 1.3 performs the key share option, which takes 13 times as long as TLS 1.2 resumption in case of ECDHE and at least twice as slow in case of Elliptic-curve Diffie-Hellman (ECDH) (2048-bit DH parameters in both cases).

To evaluate the performance of TLS 1.3 handshake and session resumption in our evaluation environment, we use WolfSSL 3.12.0 that supports TLS 1.3 (Draft 18) [12], which has minor differences in implementation from the current RFC [13] in terms of session resumption. The other settings are not changed from Section 6. Table 8 shows the results. The delays of both a full TLS handshake and a session resumption are increased compared to those of DTLS 1.2. Especially the session resumption takes much more execution time, which is approximately 11 to 15 times longer than before. Enabling the key share option causes approximately 200ms more time, while improving the confidentiality by keeping the forward secrecy.

The session resumption is beneficial in terms of delay since the delay of a full DTLS handshake is much longer than that of a session resumption. For TLS 1.3, the delay ratio of a full handshake to a session resumption is in range of 4.5 to 6.0 from our results. In comparison with the delay ratio of DTLS 1.2 (around 51), the gain of a session resumption over a handshake is reduced in TLS 1.3. In

Table 9: Execution time of sign/verify operations at TI CC2538 is measured with 256-bit ECC key (in s).

Setting	Signature	Verification
Software-only	9.126	20.534
Hardware-accelerated	0.34	0.70

general, a session resumption is more preferable as the delay ratio of a full handshake to a session resumption becomes higher.

In the D2TLS framework, the handshake delay is significantly reduced due to the delegation mechanism. Obviously, the full handshake is more secure than the session resumption. Therefore, considering the trade-off between the security and delay, the IoT operator may prefer the full handshake to the session resumption by setting the DTLS context lifetime shorter.

7.3 Hardware-assisted IoT Security

We measure the execution times of cryptographic operations at TI CC2538, which is the most expensive part in D2TLS in terms of computational cost. Section 6.2 indicates that signing operations at TI CC2538 take significant time. The signature generation time costs approximately 59% of the total delay, which is about 10s. However, the device has a dedicated hardware crypto module, which can expedite public key operations including ECC and RSA.

Table 9 shows that the execution times for sign/verify operations on a message of 256 bytes for software-only and hardware-accelerated implementations, respectively. For the signature generation time, the hardware-accelerated case costs only 0.34s, which is approximately 3.7% of that of the software-only one. The signature verification time of the hardware-accelerated implementation is also reduced to approximately 3.4% of that of the software-only one. It demonstrates that an IoT device that has a hardware crypto module can process signature-related operations much faster in D2TLS.

Nevertheless the crypto hardware does not reduce the benefits of D2TLS. A majority of IoT SoCs with a crypto hardware module currently supports the AES acceleration only, not ECC/RSA. Even if an ECC-capable crypto hardware is adopted with IoT SoCs, D2TLS can still leverage the higher performance of a high end machine in terms of communication and computation overheads. In other words, the hardware acceleration enhances D2TLS as well as DTLS.

8 CONCLUSIONS

We propose a delegation-based DTLS framework (D2TLS) for cloud-based IoT services. D2TLS allows for a resource-limited IoT device to set up a DTLS connection with small computational overhead; most of the DTLS handshake processing is performed by a delegated agent in D2TLS. Unlike prior schemes that make the private key of a device shared with the agent, D2TLS solves the key escrow problem. We investigate the communication patterns of two cloud-based IoT products, which demonstrates that most of the IoT service sessions are short and intermittent. Thus, creating a secure connection for each session will incur the substantial overhead. To lower the burden of setting up DTLS connections, D2TLS leverages the session resumption and introduces a security agent, which establishes DTLS connections on behalf of the device. Numerical evaluations are conducted with two kinds of IoT products (of different hardware capabilities), which reveals that D2TLS can achieve better performance in terms of delay and energy consumption in comparison to running DTLS standalone. In particular, the execution times of cryptographic operations in IoT devices can vary significantly depending on their capabilities and networking environments.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1A6A3A11032626). Also, this work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2016-0-00160, Versatile Network System Architecture for Multi-dimensional Diversity). The ICT at Seoul National University provides research facilities for this study.

REFERENCES

[1] Mario Ballano Barcerna and Candid Wueest. 2015. Insecurity in the Internet of Things. White Paper. *Symantec* (2015). http://www.symantec.com/content/en/us/enterprise/media/security$_r$esponse/whitepapers/insecurity-in-the-internet-of-things.pdf
[2] Carsten Bormann, Klaus Hartke, and Zach Shelby. 2014. The Constrained Application Protocol (CoAP). RFC 7252. (June 2014). https://doi.org/10.17487/rfc7252
[3] Carsten Bormann, Simon Lemay, Hannes Tschofenig, Klaus Hartke, Bill Silverajan, and Brian Raymor. 2018. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. RFC 8323. (Feb. 2018). https://doi.org/10.17487/RFC8323
[4] Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. (Aug. 2008). https://doi.org/10.17487/rfc5246
[5] Thomas Fossati and Hannes Tschofenig. 2016. Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925. (19 July 2016). https://doi.org/10.17487/rfc7925
[6] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645 – 1660. https://doi.org/10.1016/j.future.2013.01.010
[7] R. Hummen, H. Shafagh, S. Raza, T. Voig, and K. Wehrle. 2014. Delegation-based authentication and authorization for the IP-based Internet of Things. In *2014 Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. 284–292. https://doi.org/10.1109/SAHCN.2014.6990364
[8] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. 2014. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *IEEE S&P 2014*. 67–82. https://doi.org/10.1109/SP.2014.10
[9] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. 2016. A gap analysis of Internet-of-Things platforms. *Computer Communications* 89-90 (2016), 5 – 16. https://doi.org/10.1016/j.comcom.2016.03.015 Internet of Things Research challenges and Solutions.
[10] S. R. Moosavi, T. N. Gia, E. Nigussie, A. M. Rahmani, S. Virtanen, H. Tenhunen, and J. Isoaho. 2015. Session Resumption-Based End-to-End Security for Healthcare Internet-of-Things. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*. 581–588. https://doi.org/10.1109/CIT/IUCC/DASC/PICOM.2015.83
[11] oneM2M Partners. 2014. oneM2M Security Solutions. Web page. *oneM2M Partners* (1 August 2014). http://onem2m.org/images/files/deliverables/TS-0003-Security$_S$olutions-V-2014-08.pdf
[12] Eric Rescorla. 2016. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-18. Internet Engineering Task Force. https://tools.ietf.org/html/draft-ietf-tls-tls13-18 Work in Progress.
[13] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. (Aug. 2018). https://doi.org/10.17487/RFC8446
[14] Eric Rescorla and Nagendra Modadugu. 2012. Datagram Transport Layer Security Version 1.2. RFC 6347. (Jan. 2012). https://doi.org/10.17487/rfc6347
[15] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. 2018. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-dtls13-28. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-28 Work in Progress.

[16] Joseph A. Salowey. 2008. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077. (Jan. 2008). https://doi.org/10.17487/rfc5077

[17] Zach Shelby, Michael Koster, Carsten Bormann, Peter Van der Stok, and Christian AmsÃijss. 2018. *CoRE Resource Directory*. Internet-Draft draft-ietf-core-resource-directory-15. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-core-resource-directory-15 Work in Progress.

[18] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers. 2016. Twenty Security Considerations for Cloud-Supported Internet of Things. *IEEE Internet of Things Journal* 3, 3 (June 2016), 269–284. https://doi.org/10.1109/JIOT.2015.2460333

[19] A. Sivanathan, H. Habibi Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman. 2018. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing* (2018), 1–1. https://doi.org/10.1109/TMC.2018.2866249

[20] Douglas Stebila and Nick Sullivan. 2015. An analysis of tls handshake proxying. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Vol. 1. IEEE, 279–286.

[21] Hannes Tschofenig, Jari Arkko, Dave Thaler, and Danny R. McPherson. 2015. Architectural Considerations in Smart Object Networking. RFC 7452. (March 2015). https://doi.org/10.17487/RFC7452

[22] wolfSSL. 2018. TLS 1.3 Performance Part 1 — Resumption. Web page. *wolfSSL* (23 May 2018). https://www.wolfssl.com/tls-1-3-performance-resumption/