Smart Attack-Resistant Internet of Things

DELIVERABLE D4.2
DESIGN OF LOW-POWER TECHNIQUE FOR DTLS-IOT INTEGRATION

Identifier:	Deliverable
Class:	Report
Version:	
Version Date:	
Distribution:	Consortium Confidential
Responsible Partner:	
Contributors:	

DOCUMENT HISTORY

Version	Date	Comments	Status
V01	05/12/2022	Report	Initial version

EXECUTIVE SUMMARY

This deliverable presents the design of low-power technique for DTLS-IoT integration scheme, FDTLS. Consecutively, we introduce the background and challenges, implementation of the scheme and evaluations of FDTLS.

TABLE OF CONTENT

1 INTRODUCTION

This deliverable presents FDTLS, a security framework that combines storage and network/communication-level security on resource limited Internet of Things (IoT) devices using Datagram Transport Layer Security (DTLS). While coalescing the storage and networking security schemes can reduce redundant and unnecessary cryptographic operations, we identify security-and system-level challenges that can occur when applying DTLS towards such concept. FDTLS addresses these challenges by employing an asymmetric key generation scheme, a virtual peer-based handshaking mechanism, and a header size reduction scheme. Our results obtained using Contiki-based implementations on OpenMote devices show that compared to using storage and networking security separately, FDTLS can reduce the network response latency and improve energy savings

2 BACKGROUND AND CHALLENGES OF DESIGNING LOW POWER TECHNIQUE FOR DTLS-IOT INTEGRATION

2.1 Background

Recent advances in low-power embedded hardware and software technologies, combined with increased Internet connections for establishing ubiquitous wireless environments are catalyzing the deployment of various Internet of Things (IoT) services. IoT devices and their applications integrate very close to our everyday living environments and enrich our life styles by offering services that include healthcare, smart homes, and smart cities. While offering convenience, at the same time, these IoT devices open possibilities of security and privacy threats. Specifically, we can think of security threats where sensor data can be used to infer users' activity or health status, are leaked to undesired parties. Such cases can degrade the trustworthiness of an IoT system; thus, the success of IoT technology and their services in the market will rely heavily on how such sensitive data can be protected. While there can be many different forms of security attacks towards IoT devices (e.g., data spill using memory dumping, malware) this work focuses on securing a device's storage and communication link.

With such importance and need in mind, we see that IoT devices that store data in storage must support two important security- and privacy-related aspects. First, IoT devices should prepare themselves for the case when their storage (or the device) is stolen by an attacker. Given their small form-factor, there can be cases when the devices themselves are stolen and the stored data within a device can be physically extracted by an unwanted party. Second, IoT devices should also protect themselves from the case when the data being sent (wirelessly) to another device is intercepted by an attacker. Attackers may steal data at the first hop (from the data initiating IoT device directly) or also through some intermediate node on the route towards the data's final destination. In the widely-used Internet, schemes such as Datagram Transport Layer Security (DTLS) and Internet Protocol Security (IPSec) are utilized to mitigate such network-level security threats.

While addressing these security threats, we must also take in consideration the fact that IoT devices are limited in terms of energy (lifetime), computational power, and network bandwidth. This suggests that implementing separate security stacks for storage and communications (each to address the two threats above) can possess redundancy and unnecessary cryptographic operations. Take the following scenario for example. When an IoT device collects data, the data is encrypted and saved to the device's storage for storage-level security. This encrypted data, when requested to be transmitted, should be decrypted and re-encrypted again with respect to network security mechanisms (e.g., DTLS or IPSEC). As a result, the IoT device performs two encryption operations and one decryption operation before a data can be sent. To reduce such unnecessary cryptographic operations, Bagci et al. introduced Fusion to coalesce storage security and network security. Specifically, Fusion allows locally-stored packets to use the same encryption method as the network security mechanism and stores the network packet (to be transmitted) as a whole to the storage. This allows secure packets on the storage to be fetched and transmitted directly upon packet request.

However, Fusion itself has some limitations. First, Fusion was implemented and evaluated for IPSec, whereas, DTLS is considered as a more suitable security method for IoT security. The work by Bagci et al. mentions the use of DTLS, but no implementation nor evaluation is presented in their work. Second, we noticed that directly applying DTLS to Fusion causes it to be vulnerable towards replay attacks from a malicious node. Finally, Fusion assumes that there is always a pre-established connection; thus, can only operate if (at least one) handshaking process is completed with a peer node.

2.2 Challenges and threat model

IoT devices can avoid unnecessary cryptographic processing by combining the storage security and communication security mechanisms. Specifically, it is much more cost-effective (on an energy and operational perspective) to reuse a pre-built ciphertext based on the network security when exchanging packets with a peer, rather than performing the decryption and encryption processes on the stored (and secured) data every time a secure packet is newly generated. The authors in Fusion have shown that with IPSec-based network security, their scheme can be cost effective. This work takes a step further and examines how combining storage and network/communication security can be effective when using DTLS.

Unfortunately, when applying the Fusion concept to DTLS, there are some technical challenges to address. Specifically, there can be problems that arise when a single data server and multiple clients use the same key block. As mentioned earlier, unlike IPSec, in which we can compute the authentication data (e.g., ICV) separately for each packet receiving peer, in DTLS, packet authentication operations (to compute the MAC value) is performed before the encryption phase (Figure 1). This means that to maintain unique MACs for different peers, different copies of the same data need to be in the storage. When doing so, the limited storage can become a bottleneck for the system. However, if all nodes share the same keys for the encryption and authentication processes, a potential attacker can easily eavesdrop a packet sent by a client and change its IP address to another client to fool the system. This is a major problem that needs to be resolved before we can adopt the Fusion concept to DTLS.

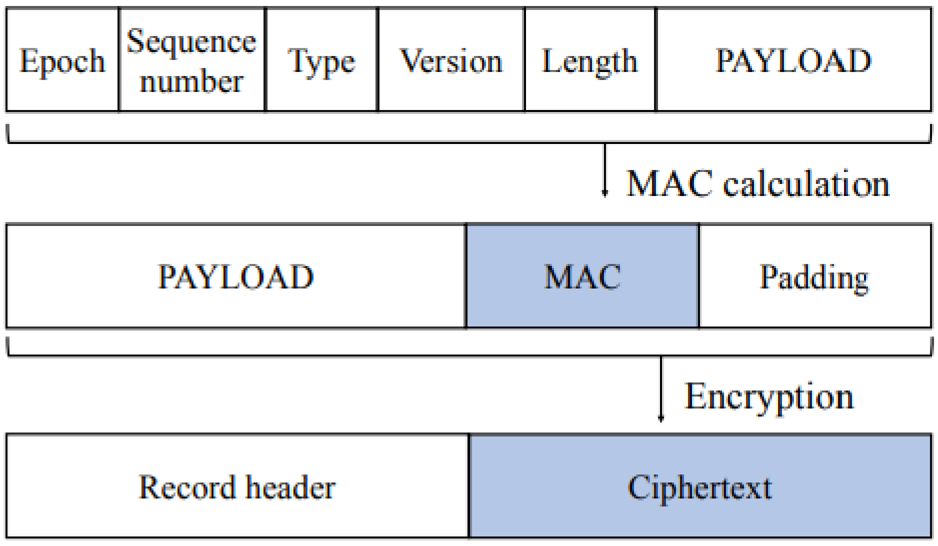


Figure 1 DTLS Encryption Process

Based on the observations above, we present a potential threat model as the following. figure 3: Modified DTLS header used in FDTLS

As mentioned above, for the Fusion operations to properly take place, security keys used for encryption and authentication on the server side need to be kept identical for all communication links. Fusion generates and uses one constant key block corresponding to the PSK for communication between a server device and multiple client devices. To create this constant key block, when key calculation is performed during the handshaking process, a random number is not used in the key calculating process. Fusion creates such “special” key blocks by using its own specified pseudo-random number generator. As a result, the key block is always kept the same when the same PSK is used. Multiple clients use the same key block, which contains the Client MAC key and the Client Encryption key for exchanging and encrypting its packets (e.g., data request packets to the server node). Therefore, when the payload and the parameters used in MAC calculation, such as sequence number and epoch, are kept the same, different clients can create the same MAC value and ciphertext. This causes an attack scenario as shown in Figure 2 to potentially occur.

In Figure 2, we show two receivers (clients) R1 and R2, with one sensor (server) S1 and one attacker A. All nonattacker nodes, R1, R2 and S1 use the same the key block when operating Fusion with DTLS, and the attacker node A does not know of this shared (common) key block in the initial phase. First, R1 sends a data query to S1. When in operation, A can eavesdrop the data query packet sent by R1 and change the IP address of the packet to R2’s address and send it to S1 (replay). S1 recognizes this packet as a packet sent by R2, since the common key value is used, and utilizes its system resources to transmit a proper response query. The attacker may not be able to decrypt the response sent by S1, but a DoS attack, that wastes S1’s resources can easily take place.

Furthermore, if A requests packets with sequence numbers 1-100 (which are already stored in S1’s flash and R1 has already requested for) from S1, victim node S1 may gain fake knowledge that R2 has received these 100 packets. Therefore, when R2 sends a request for packets in the 1-100 range, S1 may think that this is a replay attack from a malicious node and deny the request.

To solve this problem, we can think of two solutions. Note that such a replay attack is only possible because the DTLS ciphertext does not contain information about the potential receivers such as the IP address of the receiver (or client). Therefore, the first (and simple) solution is to change the structure of the DTLS record header to include the Client ID, as we illustate in Figure 3. As described above, the original DTLS header contains a six byte sequence number field, but the modified DTLS header exploits one of these six bytes for the Client ID field, reserving space for 256 unique clients. To allocate a unique Client ID for each client device, the server node (which in our scenario is the sensor node that distributes data) issues a Client ID to each node during the handshaking process. In the attack scenario above, when a receiver node sends the client hello message to S1, S1 responds with a server hello, which includes the Client ID. For example, when S1 allocates Client IDs 1 and 2 to R1 and R2, respectively, this Client ID is included in the MAC calculation when R1 and R2 send queries to S1. By doing so, even if the attacker changes a packet's IP address and Client ID to that of R1's packet (to mimic R2's activities), S1 can realize that the packet is invalid through the MAC integrity check phase. Note that this modified DTLS header is only used when clients (e.g., R1 or R2) send their packets, and S1 does not use this modified header format for its own transmissions so that packets can be successfully decrypted at the receiver.

However, we must keep in mind that the more core cause of the above security treat is the use of the same client key value for all clients. Therefore, a more fundamental solution will be to maintain different client keys for each node. Specifically, the original DTLS key block consists of a client write MAC key, server write MAC key, client write encryption key, and a server write encryption key. If the client write MAC key and the client write encryption key is unique for each node, A's fake packet (with a modified IP address field), will not be properly decrypted at S1; thus, the malicious activity can be implicitly detected.

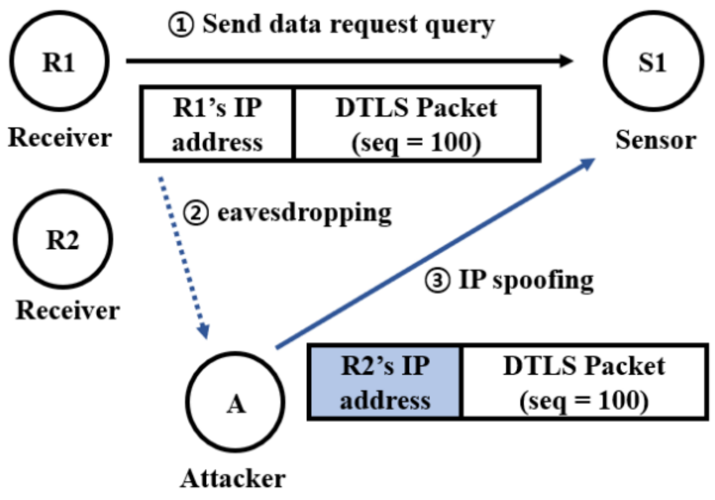
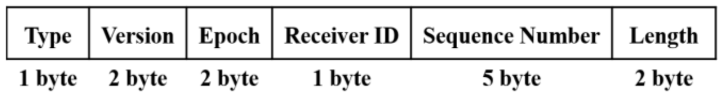


Figure 2 Scenario where attacker node sends an IP spoofing packet. A: attacker, R1, R2: receivers, S1: sensor node.



Type	Version	Epoch	Receiver ID	Sequence Number	Length
1 byte	2 byte	2 byte	1 byte	5 byte	2 byte

Figure 3 Modified DTLS header used in FDTLS

To create a key block that contains unique client keys for encryption and authentication over multiple devices, we use random numbers only when calculating the client side key. However, note that the server key must be kept consistent for all communication links given that having different keys would require storing data for all peers separately, inducing storage overhead. Thus, in our solution, the server write keys are created without random numbers. By doing so, we can extract consistent server keys corresponding to the PSK, and also extract unique client keys for each communication link

3 FDTLS: Supporting DTLS-based combined storage and communication security for IoT devices

3.1 Overview

To effectively support DTLS with Fusion, and also address the security threat discussed above, we designed FDTLS. Specifically, FDTLS is a framework that combines DTLS communication security and storage-level security on resource limited IoT devices. FDTLS is implemented on Contiki OS 3.0 and we take the following three points as our core design goals.

1) FDTLS should minimize cryptographic processing, latency and energy consumption when the device responds to data request queries, by combining storage security and DTLS communication security.

2) FDTLS should allow the devices to perform the security functions under any network conditions, and allow DTLS-based network and storage security to take place even before any clients are ready to communicate with the server node.

3) FDTLS should effectively address threat models such as replay attacks that can occur when combining the storage security and DTLS communication security.

To minimize the overhead of repeated cryptographic processing, similar to the concepts introduced in Fusion, in FDTLS, we coalesced the storage security and communication security, but use DTLS as the communication security mechanism. IoT devices encrypt data as a DTLS packet using DTLS communication security and save the packet itself on their storage. If a device receives a data request query, we can reuse the stored data as a DTLS packet just by reading it from the storage. There are no additional cryptographic operations taking place to create a DTLS packet. This mechanism also reduces the data response latency for data request queries given that minimal computational operations take place at the data-sending node when a query is received. However, on the downsize, if the device stores the encrypted data (ciphertext) with DTLS record header in its storage, the overhead on the storage perspective can increase due to the header size. Unfortunately, if we only store the encrypted ciphertext of the DTLS record, we cannot make estimations on the DTLS record header from the ciphertext when trying to send the message. Fields such as the sequence number or epoch of the chipertext are not included in the ciphertext. To resolve this tradeoff, we implemented a FDTLS SET HEADER function to reconstruct the header when creating the DTLS packet without the need to store the full DTLS header for all record headers. Owing to this function, an IoT device only needs to write one DTLS header for all of the data contained in a single file and the storage overhead can significantly be reduced.

Furthermore, if the network faces a connection problem or there are currently no clients ready to communicate with the IoT node that is generating data, the node (acting as the data server) may not be able to perform handshaking with a peer. This is because a proper key block for DTLS cannot be created without an external connection. As a result, the sever node cannot apply the DTLS communication security mechanisms for its storage security operations. To address this issue, FDTLS includes a Virtual Peer generation mechanism to create a key block even if there are no clients ready to communicate. By using this feature, the sever device can encrypt data even though there are no devices currently ready in its surroundings. Note that creating separate keys for all peers can cause storage overhead, but using the same key for all parties can reveal security threats; thus, we propose a *FDTLS PRF*, which utilizes random values only for the client write key generation, and maintains a consistent server write key to revolve these issues.

Note that since the above require minimal modifications to the existing DTLS protocol, IoT devices can also support DTLS with minimal cost. Thus, if the server implements both protocols, it select and use between FDTLS and DTLS depending on the client device configurations.

The following sections will present details on the core schemes of FDTLS as briefly discussed above.

3.2 Virtual Peer Generation

Existing storage security schemes can be used anytime, independent from the existence of other neighboring devices. However, for network security scheme such as DTLS, if there is no device to connect and communicate to a key block cannot be created; thus security scheme cannot be used. However, given that FDTLS is designed to utilize the security schemes for the networking/communication stack as the storage security scheme, lacking a neighboring node to communicate with would mean that packets being stored cannot be secured either. To solve this problem, FDTLS includes a virtual peer generation feature that enables a device to self-create a key block without the need of external connections. Recall that in Fusion the key block is created without a server random and a client random. Therefore, the key block can be potentially be derived from the PSK without information from any other device.

Specifically, the virtual peer creation operates as follows. First, the device creates a virtual peer structure and selects what PSK and cipher suite to use. The virtual peer is simply a struct that contains the role of a device (server or client), a PSK, a key block, an epoch, and a sequence number of the next packet to send/store. Next, we calculate a key block using the selected PSK. When calculating the key block using the pseudo-random function, the client random and server random are filled with NULL. The key block is then copied to a virtual peer and the role, epoch and sequence number fields are (virtually) initialized. We assume here that the role of the device that uses the virtual peer is the server node (i.e., data generating sensor that stores its data). Therefore, when encrypting data using the key block, only the server key is used for encryption. The sequence number included in the virtual peer struct increases as the device encrypts a new data and moves on to the next, and the epoch and the sequence number are used for encrypting the next data. Note that the all stakeholders in this process must share the same DTLS version, PSK and a cipher suite. This allows the stored data to be used as a proper DTLS record when communicating with other clients.

One thing to point out is that, while the key generated in this process can be used to “encrypt” data at the server node, when the server receives data from a real peer node, a separate key should be used. The following section discusses how we create such an asymmetric key chain to protect FDTLS-based IoT systems from potential security threats

3.3 FDTLS Pseudo-random Function

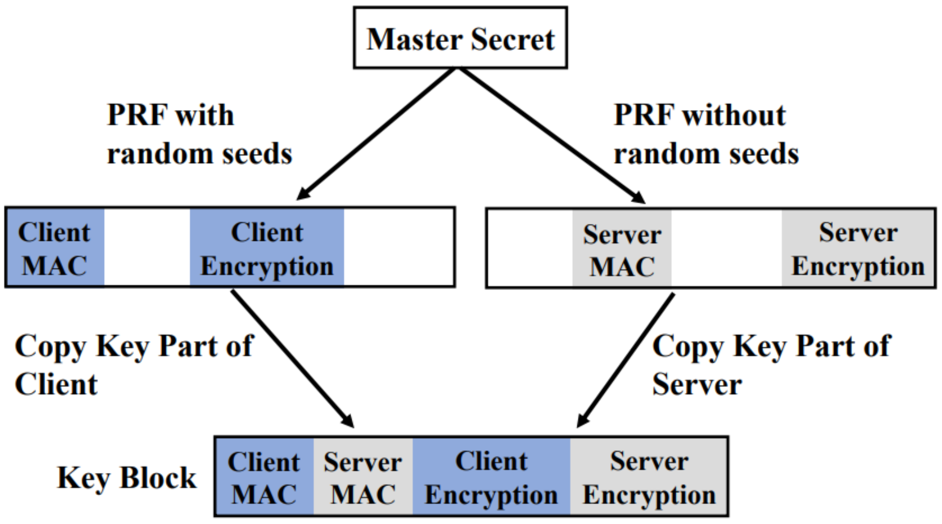


Figure 4 FDTLS Pseudo Random Function

In FDTLS, each data generating sensor device acts as the server node for DTLS operations. While using key blocks created without any random numbers can help unify the DTLS packet for all nodes in the network, this can also lead to the replay attacks as discussed in our threat model. Such an issue can be solved by maintaining asymmetric key generation methods, in which we keep the randomness in the client key generation, while neglecting such randomness in the server key generation phase.

For this purpose, we take a simple approach and propose FDTLS PRF that executes the original PRF operation twice (once with random numbers and the second time without random numbers). Figure 4 illustrates the details of how FDTLS PRF works. Specifically, FDTLS PRF generates two different types of keys: the server key without using random numbers, and the client key that uses random numbers for key generation. This generates a per-client randomized client key, while the server key is kept consistent for all clients, so that the stored DTLS data on the FDTLS device can be consistent for all client nodes to decrypt. Note that the key block used for virtual peers are also kept the same as this consistent server key. Again, the server key that is created without using randomness is shared for all communication links, allowing pre-stored data to be usable for all clients. While this scheme is simple, it is effective enough to resolve the security threat we discussed Given the resource limitations of typical IoT devices, we find it very important to maintain a low-complexity scheme for all on-device local operations.

3.4 Storage Management

As discussed above, a DTLS packet consists of a record header and ciphertext. The ciphertext can be created by using the DTLS security mechanisms and the record header information is used for MAC calculation. If we store all of the generated data in flash as a full DTLS packet (consisting of DTLS header and ciphertext), when receiving a data request from a client, the server node can simply read the packet from its flash and send it directly without any added operations.

However, compared to storing only the “data” in secured form, such a design would require much more memory space due to the overhead of saving header information for all packets. For example, assuming that the payload size (i.e., data) is 16 bytes and the system is using the TLS PSK WITH AES 128 CCM 8 cipher suite, the 13 byte size DTLS header and the 8 byte CCM MAC will add 131.25% of overhead in terms of additional flash memory space. This not only increases the memory usage, but will naturally increase the latency to write the data to the flash as well leading to more energy consumption. Unfortunately, if we only save the ciphertext (i.e., data) to reduce space, we will not be able to properly reconstruct the full DTLS packet from the flash due to missing fields that cannot be inferred. Nevertheless, there is still a need to reduce the amount of data we save to the flash to achieve system-level efficiency.

For this, we propose a simple scheme that selectively writes header information only for the first data in the file batch. Using this initial header information, we can infer the header information for subsequent data samples. In other words, when creating a “file” (the basic data writing unit of the Contiki File System) and the first ciphertext is written, a 13 byte header is also written together with the data. This header can be used to derive information for constructing the header for subsequent data. This can be done because the

$$\begin{aligned} Seq(TargetCiphertext) &= Seq(FirstCiphertext) \\ &+ \{Loc(TargetCipherext) - Len\,(H)\}\,/Len\,(C) \end{aligned}$$

data (or ciphertext) length is fixed and the initial sequence number/epoch information is included in the first data's header. Specifically, we can extract the sequence number of a specific target ciphertext *Seq(TargetCiphetext)* from a file using the following

Here, *Seq(F irstCiphetext)* returns the sequence number of the first ciphertext written in the file (using the attached header information), and *Loc(T argetCiphertext)* returns the memory location of the target ciphertext with respect to the beginning of the file. *Len(H)* and *Len(C)*, represents the size of the fixed header and the fixed size of the ciphertext record, respectively. We define the FDTLS SET HEADER() command to fetch the sequence number of a specific ciphertext. For example, when a ciphertext length is fixed to 24 bytes (16 bytes of data with 8 bytes of MAC), assuming that the first ciphertext starts at location 1,000 (based on the header) and the target ciphertext is extracted from location 2,413 bytes in the file, then the sequence number of the target ciphertext is 1100 = 1000+(2413-13)/24. Furthermore, the epoch, version, type and length fields can be copied from the first header, As a result of this scheme, we can reduce additional storage overhead from 21 bytes (Header + MAC) to 8 bytes (MAC only) by omitting the header overhead.

4 IN DEPTH EVALUATION OF FDTLS

4.1 Evaluation setup and scenarios

We use the TinyDTLS implementation and ContikiOS 3.0 to implement and test FDTLS on the OpenMote B board. This board uses a CC2538 system on chip equipped with 512KB flash memory and 32KB RAM. We use a pre-shared key for DTLS key exchange, and use TLS PSK WITH AES 128 CCM 8 as the cipher suite. Furthermore, we use the Contiki File System (CFS) to read and write data from and to the device's flash memory. All networks used in this work are IEEE 802.15.4 PHY-based and the IPv6 standards for low-power and lossy networks (6LoWPAN , RPL) are used on the network layer.

We emphasize that a core design goal of FDTLS is to minimize the cryptographic processing costs so that the latency and energy consumption can be minimized. Therefore, as performance metrics, we use the impact of FDTLS on the latency and energy usage performance compared to a typical Internet standard-based system. Furthermore, since these two metrics, when operating security mechanisms, depend heavily on hardware accelerators, we perform evaluations using both software and hardware-based encryption mechanisms.

For thorough evaluations, we breakdown the security mechanisms within a node to six different types of operations and define two different types of test scenarios. We denote a device that uses FDTLS as the FDTLS device, and denote a device that uses the storage security and DTLS security separately as a DTLS device. The six types of operation are as follows:

• CTR encryption We assume that the existing system uses CTR encryption before storing data to the storage.

• CTR decryption We assume that the existing system uses CTR decryption to decrypt data from the storage.

• CFS write This operation writes data to flash memory.

• CFS read This operation reads data from flash memory.

• CCM encryption This operation is used to create a ciphertext for constructing a DTLS packet.

• Transmission DTLS packet is sent to another device.

• Other operations All other operations except the above operations. This includes the operations for decrypting query messages received from the peer nodes.

As experimental scenarios, we specify the following.

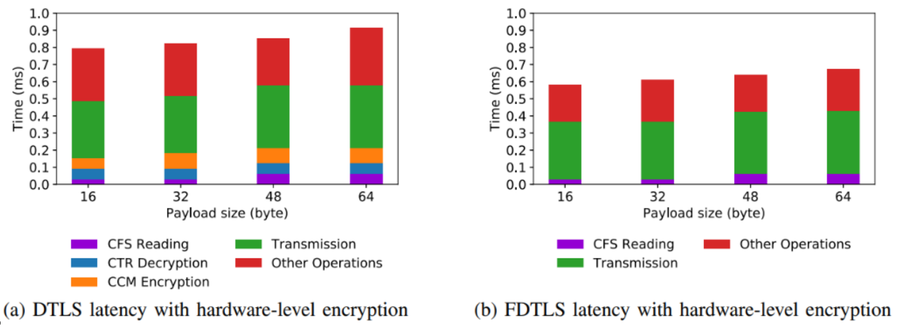
• Scenario 1 (naive DTLS-based systems): In naive DTLS-based IoT systems, the storage and communication security are independent. Thus, (1) the sensor node encrypts data using CTR encryption for storage, (2) the sensor node stores the ciphertext and waits for a data request query, (3) one or more clients send a data request, which leads to reading and decrypting the stored data using CTR decryption, (4) DTLS packet construction using CCM encryption and finally, (5) the sensor transmits a packet to the data requesting client(s).

• Scenario 2 (FDTLS): In FDTLS, a sensor node performs the following. (1) The sensor node creates a DTLS packet using the data via CCM encryption, (2) the node stores the ciphertext and waits for a data request, (3) a client sends a data request, and the sensor node reads the ciphertext from storage and restores the header information, and lastly, (4) the sensor transmits the packet to client(s).

Based on these two experimental scenarios, in our evaluations, we measure the latency for packet transmissions (from the query to transmission) and also compute the amount of energy consumed for packet operations and transmission. In addition, to examine the performance for potentially a wide range of resource-limited platforms, we compare for the cases where the security operations take place in a designated hardware or through software implementations.

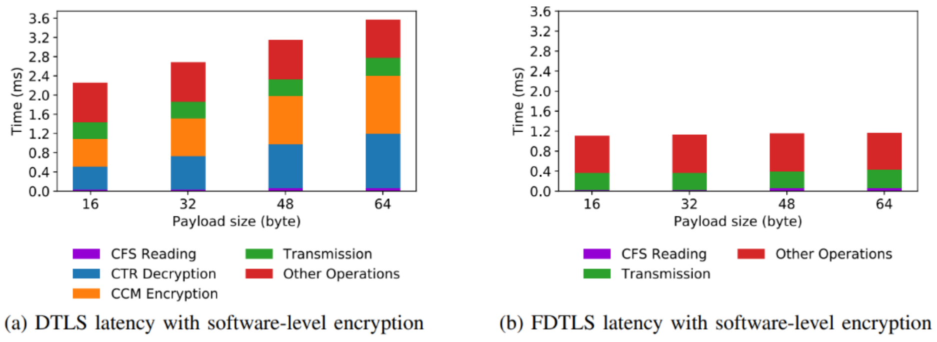
4.2 Operation latency

We start by measuring the latency for a sensor node to receive a query packet from a peer node until the target data is transmitted using the two scenarios discussed above



(a) DTLS latency with hardware-level encryption (b) FDTLS latency with hardware-level encryption

Figure 5 Latency of the packet response process for naive DTLS (left) and FDTLS (right) with hardware encryption.



(a) DTLS latency with software-level encryption (b) FDTLS latency with software-level encryption

Figure 6 Latency of the packet response process for naive DTLS (left) and FDTLS (right) with software encryption.

Figure 5 plots the latency results for each operation when using software-based encryption for different payload sizes. Notice here that FDTLS (in Figure 5b shows significantly lower latency when compared to that of traditional schemes that use storage and network security operations separately. This gain in latency is due to the fact that FDTLS does not need to perform any operations related to CTR decryption and the additional step of CCM encryption, which accounts for more than 50% of the total latency, are not needed when sending a pre-stored packet. Another interesting point to note is that FDTLS shows a relatively constant latency over different payload sizes. This is except for the small added duration due to reading longer packets frame the flash and for transmission, all other operations are kept nearly the same.

Next, in Figure 6 we show the latency performance of the two schemes when using hardware-based encryption. Results here, when compared with Figure 5, suggest that using hardware encryption reduces the latency significantly. Due to the same reasons, using FDTLS shows a faster performance. Notice that for FDTLS when comparing with the software encryption-based evaluations, the “other operations” portion shows a noticeable decrease. This is due to the fact that this portion of latency includes the time it takes to decrypt a secured query packet from the peer. Therefore, with hardware support, the latency decreases for such operations.

4.3 Energy Consumption

To measure the power consumption introduced from FDTLS, we use the Monsoon power monitor connected to the OpenMote B board running the two schemes discussed earlier. Since a single operation takes only a small amount of time (and energy), we measure the energy usage (in μAh) while repeating our two experimental scenarios for 100 times.

Figure 7 shows the energy consumption for 100 iterations when using software-based security mechanisms. We can see that the energy used for exchanging data when using FDTLS is only half of that of typical schemes when DTLS is used together with a separate storage-level security scheme. The results here can be directly explained from our observations made for the latency experiments. Since we spend less time in taking care of the security operations, we can put the device to a low-power state more frequently.

Finally, Figure 8 plots the energy consumption for processing 100 data request queries when using hardware-level encryption. Here, we can see that the energy usage differences are much smaller compared to when using software-supported security. The reason for the reduction in energy consumption gap is due to the cryptographic operations overhead decrease due to the hardware encryption, again, similar to our observations for the latency plots. Another point to note is that with FDTLS, the node stores more data to the CFS. For example, when writing a 16 byte data, in FDTLS, 24 bytes are written to the flash due to the MAC overhead. This adds overhead to the writing process. Given that the cryptographic operations induce less overhead, this increase in writing becomes a more noticeable factor. Nevertheless, using FDTLS still shows a lower energy footprint for all cases, which is important given the battery limitations of most IoT systems.

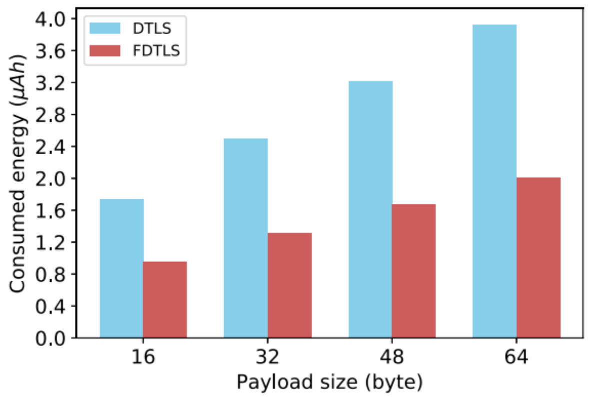


Figure 7 Energy consumption comparison for native DTLS and FDTLS with software encryption (100 iterations)

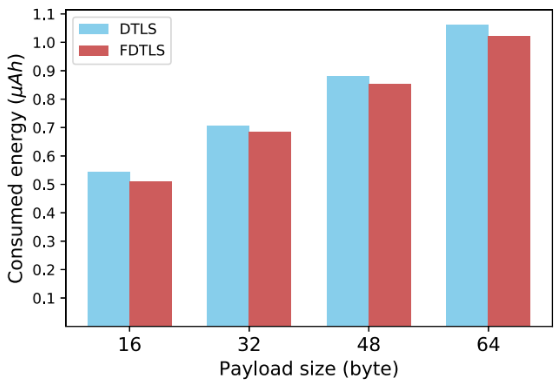


Figure 8 Energy consumption comparison for native DTLS and FDTLS with hardware encryption (100 iterations)

5 CONCLUSIONS

In this document, we present FDTLS, to combine the storage and networking security mechanisms for resource limited IoT devices using DTLS. To address security threats that can potentially occur when configuring such combined security, FDTLS includes a new pseudo-random function for asymmetric key generation and optimizes the storage patterns to minimize additional storage overhead. Our observations and results suggest that FDTLS successfully avoids replay attacks and reduces the computational latency and energy usage cased from supporting multiple security functions on an IoT device.