Evaluating CoAP, OSCORE, DTLS and HTTPS for Secure Device Communication

Kristofer Nedergaard, Bhupjit Singh(✉), and Birger Andersen

DTU Engineering Technology, Technical University of Denmark,
2750 Ballerup, Denmark
KNE@ICEpower.dk, {bhsi,birad}@dtu.dk
https://www.dtu.dk

Abstract. The purpose of this paper is to explore the differences and relations between the protocols CoAP, OSCORE, DTLS and HTTPS. Our focus is at performance and the general security benefits of the different protocols. We evaluate the feasibility of using CoAP encrypted with OSCORE or DTLS as opposed to using HTTPS/1.1 or HTTPS/3. We find the CoAP based solutions to be more performant than HTTPS, however we also find them to be harder to implement, less widespread and potentially more insecure than HTTPS. From these findings, in most scenarios we recommend implementing solutions based on HTTPS if the performance and overhead can be tolerated.

Keywords: CoAP · OSCORE · DTLS · HTTPS · Secure communication

1 Introduction

During the past decade, we have been experiencing a significant uplift in small home appliances and sensor stations that are connected to the internet. As our homes become increasingly more connected, we are sharing larger parts of our lives on the internet. This exposes us to potential intruders that can remotely monitor our daily lives.

A range of different methods to secure data transfer from our homes to cloud or from Device to Device (D2D) has arrived, some of which are targeted specifically toward devices. One of the most popular such protocols has become CoAP (Constrained Application Protocol) [1], and more specifically, CoAP secured by DTLS (Datagram Transport Layer Security) [2]. Many cloud providers, however, do not receive data directly in the form of CoAP/DTLS. Instead they receive data primarily through protocols that are traditionally implemented for unconstrained network environments like HTTPS (Hypertext Transport Protocol) [3]. This presents a problem that has been solved by implementing intermediate proxies that translate between CoAP/DTLS and HTTPS/TLS. This proxying however, inherently introduces a security risk at each proxying point, as the DTLS data will need to be decrypted and re-encrypted for further transport as TLS [4].

T. Pereira et al. (Eds.): IoECon 2022, LNICST 458, pp. 117–132, 2023.
https://doi.org/10.1007/978-3-031-25222-8_10

In this paper we explore two different ways to circumvent the security issue with proxying data between different protocols. The first is to encrypt the entirety of the message separately from the transport protocol, something the protocol of OSCORE aims to accomplish [5]. The second is to explore the feasibility of using HTTPS directly on the device, which effectively eliminates the need for proxying between protocols altogether. We also provide an overview of different protocols and their performance and security advantages and disadvantages.

In Sect. 2 we introduce necessary background knowledge about the different protocols. Section 3 and 4 present a test setup and the results of the tests. In Sect. 5 and 6, the results are interpreted and the implementations of protocols are discussed. Following this we present a more in-depth review of the security of the protocols in Sect. 7. Sections 8 and 9 discuss and conclude on our findings.

2 Background

Computers and networking become more and more advanced and we find ways to make our computers smaller, both in terms of physical size and power consumption. This has allowed for devices that we embed into items/everything that we use in our daily lives. Things that are today ranging from light bulbs to kitchen appliances. We often like these devices to be controllable and report back to us remotely, perhaps even when we are not at home, but often, we would also like to keep our data private and secure.

In this section we give an overview of some of the protocols in use today and give some insight about them in a historical aspect that will shed some light on why the networking protocols explored in this paper exist and explain each of their different original purposes.

2.1 HTTP and HTTPS

HTTP, the Hypertext Transport Protocol was introduced in 1996 [6]. HTTP has evolved to become one of the most widespread networked application protocols in existence. It has been, and still is being used for many websites. Because of its widespread use, it is also being used in many IoT device projects, even if it may not be the fastest or most secure option. HTTP is inherently insecure and messages are transferred in plaintext. This allows for anyone with networking skills to pick up and read transferred messages by sniffing tools containing sensitive data like passwords, credit card information, or perhaps just what temperature your home has at a given time. No matter what kind of information, it definitely presents a security risk. HTTPS (Hypertext Transport Protocol Secure) was introduced to mitigate security issues of the HTTP protocol was introduced [7].

HTTPS extends HTTP and adds TLS encryption in order to secure the connection between a server and a client. HTTPS and TLS were originally designed for use over the reliable TCP transport protocol which is what the original HTTP protocol was based on, but in recent times, an effort has been made towards switching from TCP to the unreliable lightweight UDP transport protocol in

order to circumvent the overhead in TCP and speed up communication over HTTPS. This effort is what has evolved into QUIC [8] - a protocol which allows HTTPS and TLS connections over UDP, effectively replacing TCP in order to gain a performance increase. Along with QUIC, a new standard for HTTP was defined: HTTP/3.

2.2 CoAP and DTLS

A challenge has always been how to proceed in regard to transferring collected data to and from each device and send them to a cloud server. This, however, has presented itself to be an issue in IoT devices because of adverse network conditions and the fact that many devices are battery operated. This constrained environment has lead to a need for a new networking protocol to break with traditional TCP-based protocols that expect close to perfect networking conditions. This is why CoAP, the Constrained Application Protocol was introduced [9].

CoAP has been designed to mimic the REST capabilities of a normal HTTP server, but due to the aforementioned constrained environments, it has been designed to transfer data over UDP instead of TCP. Furthermore CoAP significantly reduces the header size and as opposed to HTTP, the header's size is also not of variable size which allows for more precise calculations and dependable transfers. CoAP is also significantly better at handling scenarios with high amounts of packet loss. CoAP implements a publisher/subscriber system like the MQTT (Message Queue Telemetry Transport) protocol, where it is possible to subscribe to data through a broker instead of asking the other device. This gives the possibility of receiving data from one another without being able to connect directly, but instead only contacting a common data broker [10].

Securing the CoAP connection is by definition in the RFC (Request for Comments), done via Datagram Transport Layer Security, DTLS [11]. DTLS is based on TLS as in HTTPS. It therefore also provides the same security. DTLS, while designed to be very similar to TLS, provides additional functionality to handle the unreliability of UDP. This also means that DTLS can function well even in tough network conditions with many packet losses. DTLS allows for a secure end-to-end connection between two devices when communicating via CoAP over UDP in a very similar manner to what is seen with TLS and HTTP. Transferring secure data over CoAP has for a while been synonymous with using DTLS.

2.3 OSCORE

OSCORE can be seen as an extension to CoAP. It alters the options processing of CoAP and therefore also modifies the original definition of CoAP. Due to this, changes to CoAP implementation are required to have a fully functional OSCORE implementation. OSCORE is defined in a way that allows it to be transferred over CoAP as well as HTTP. OSCORE also does not encrypt the CoAP messaging and token layer, so when transferring OSCORE encrypted data, these layers will still be in plaintext [12].

This means that OSCORE, while slightly changing the specifications of CoAP, is designed in a way that can be implemented across multiple different networking protocols.

OSCORE was designed specifically to mitigate an issue that presents itself when using CoAP with DTLS. This issue is that, in order to connect to a cloud solution or any other kind of HTTP server, it is often necessary to introduce a intermediate proxy that translates between CoAP and HTTP [13]. The same is the case the other way around, when a HTTP client has to connect to a CoAP server. OSCORE aims to improve this situation by encrypting the payload itself, somewhat independently of the protocol that transports the CoAP encrypted data. By doing this, OSCORE improves upon one of the bigger security concerns of CoAP with DTLS, completely eliminating the potential risks associated with CoAP↔HTTP proxying. Figure 1 further visualizes this.

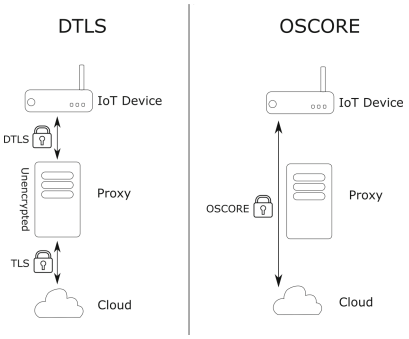

Fig. 1. A visual representation of the differences between proxying DTLS and OSCORE.

OSCORE currently does not define a method of key exchange. This can be seen as an advantage in that the key exchange protocol can be chosen independently of OSCORE. This, however, also has a side effect as different implementations of OSCORE might not be compatible if the key exchange algorithms differ. Because of the design of OSCORE, it can even be possible to encrypt the CoAP messages with DTLS and OSCORE alongside each other, as well as encrypting with DTLS or TLS in only parts of the journey between proxies and endpoints.

2.4 Comparison of Protocols

Figure 2 shows the different layers that protocols are working on. This shows the reason why OSCORE encryption can persist through proxies, but TLS and DTLS cannot. QUIC implements TLS functionality as part of the QUIC protocol, so it can be seen as delivering the same kind of functionality as TLS on its own. Each protocol has a number of advantages and disadvantages.

In Table 1 some of these are shown. While HTTPS initially was a TCP-only protocol, more recent editions have been extended to work with UDP. This eliminates some overhead and makes it better suited for device communication.

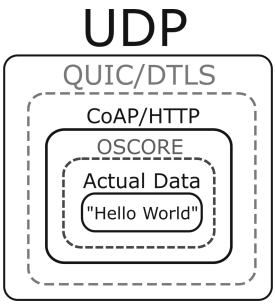

Fig. 2. Slightly simplified visual overview of the layering of different protocols.

Table 1. A comparison between different network protocols.

	Transport	Proxy safe	Forward secrecy	Header size
HTTPS/1	TCP	No	Yes	Variable
HTTPS/3	UDP	No	Yes	Variable
DTLS	UDP	No	Yes	11 bytes
OSCORE	UDP	Yes	No	11 bytes

One issue with this might be that it requires the server to support HTTPS over QUIC, which can be troublesome as HTTPS/3 is not in widespread use yet. CoAP has always been UDP-only, which ensures that the transport overhead stays reasonable low.

3 Test Setup of Protocols

In order to perform a fair test, we have generated a set of random bytes in a few different sizes, ranging from 25 bytes to 2500 bytes as sample data to be used across the protocols and test environments. For performance measurements the application will attempt to transfer the data 10000 times and compute the average time for completion. In all timing measurements these average times are the reported values.

3.1 OSCORE and DTLS

As both OSCORE and DTLS is generally transported via the CoAP protocol, it made sense to find a CoAP library capable of handling both of these. In these tests, the Californium Library by Eclipse Foundation was chosen for this specific purpose [14]. The Californium library has lightweight sample implementation of communication with both CoAP/DTLS and CoAP/OSCORE. Californium also includes examples for handling unencrypted CoAP communication so it is possible to compare the different protocol overheads against using plain text directly.

The tests consist of a simple server architecture that can respond to requests on multiple ports, specifically ports 5683, 5684 and 5685 for respectively OSCORE, DTLS and Plain Text CoAP communication. Regardless of the protocol in use, the server responds to the same request URL with the same data. Californium therefore allows us to have a very effective comparison between the different CoAP security protocols. The client is capable of requesting a resource from the server and recording the time spent from request to response.

3.2 HTTPS/3

Unfortunately, Californium Library only supports CoAP communication. Because of that, the HTTPS/3.0 test setup looks different to the CoAP based protocols. However, we have tried to keep the structure very similar. The exact same naming scheme for retrieving the same bytes has been kept, the client is entirely implemented in Java alongside the CoAP client, and the actual data transferred between the server and client is identical to the CoAP test setup.

The HTTPS/3.0 server is set up with a Caddy2 server [15]. This is capable of answering over TCP and also QUIC which is needed in this test setup. It means that both HTTP/1.1 and HTTP/3 can to be tested and compared on the same server setup. The Caddy2 server specifically has an opt-in functionality 'experimental_http3' that needs to be enabled for allowing Caddy2 to respond via QUIC over UDP. As this is a local test without a proper domain name, the local Caddy2 root certificate also needs to be installed in the test client machine's trust store. Caddy2 serves a QUIC through the QUIC-Go library [16]. This is a common Go library seen in a few recent, but well-known, web-application projects like Caddy, Traefik and SyncThing.

For the client set up, the flupke HTTPS/3 library has been chosen [17]. This library is capable of connecting to secured HTTPS/3 websites and retrieving data in a very similar fashion to the Californium CoAP library. It extends the default HttpClient implementation in Java and because of this it is very easy to implement a HTTP/3 Client with flupke.

3.3 HTTPS/1.1

For good measurement, a test setup with HTTP/1.1 is also prepared. With this we can test whether there is an increase or decrease in performance between HTTPS/3 over QUIC and HTTPS/1.1 over TCP. The HTTPS/1.1 test setup is almost entirely identical to the HTTPS/3. Caddy2 supports answering in both HTTPS/1.1 and HTTPS/3, so no configuration changes are needed.

On the client side, the flupke library for HTTP/3 extends the default HttpClient in Java. This means that the only difference between the test setup for HTTPS/3 and HTTPS/1.1 will be to instantiate the HTTP/1.1 client from the built-in java libraries.

3.4 Test Machine

The tests were performed using two identical VirtualBox Virtual Machines with 1 GB of RAM and a single processor core at 100% capacity allotted. The virtual machines were running Ubuntu 20.04.3 LTS. The host machine was a 6-core, 12-thread Intel I5-11400 with 32 GB of physical memory, running Ubuntu 20.04 LTS. The virtual machines were connected with a VirtualBox Internal Network without outside interference. The Californium library does not build as a whole out of the box from their git repository. Because of this, each virtual machine is installed with a full desktop environment along with a Java IDE. This allows for running the small subset of classes that we need from Californium. In the internal network between the virtual machines, a larger MTU (Maximum Transfer Unit) than what is normally present in networking is allowed. This is to see the performance of each protocol when encrypting significant amounts of data.

Furthermore, in a different set of tests, to ascertain that the Californium Library were communicating correctly, a Windows 10 computer with Wireshark sniffing tool was used to sniff packets sent between the server and client software and validate that the transferred networks packets were indeed encrypted with the expected protocols and contained the expected data. The total bytes transferred were also recorded with Wireshark on the same Windows setup. Using this method, we eliminate insecurities if a library reports wrong values. This makes it possible to verify that the recorded byte lengths are correct.

4 Results of the Practical Tests

A series of practical tests to determine performance and overhead between different protocols were performed.

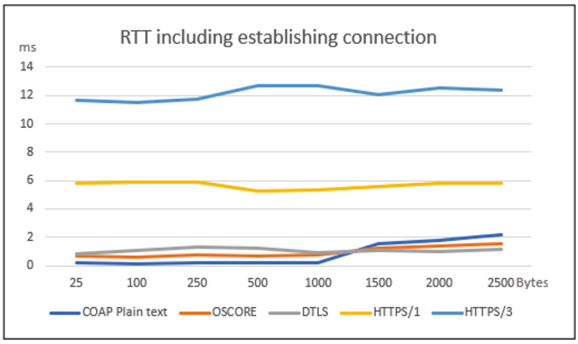

Fig. 3. Total time from establishing connection until complete data is received.

Table 2. Total bytes transferred from establishing connection until data is received.

Data bytes	CoAP plain	OSCORE	DTLS	HTTPS/1.1	HTTPS/3
25	133	158	1708	3703	10622
100	209	233	1783	4209	11123
500	609	633	2183	4610	11435
2500	3071	3196	4183	6530	13026

Figure 3 shows the total time in milliseconds for each protocol, for establishing a connection and receiving the given amount of bytes. Table 2 likewise show the total amount of transferred bytes, including overhead from establishing the connection, for given amounts of data for each protocol.

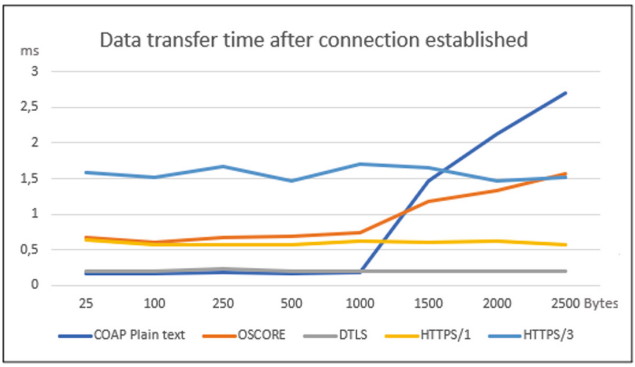

Fig. 4. Time from requesting data until received, after connection is established

Figure 4 shows the time required from the request until the data frame arrived. Important to note is that this is the time subtracted the connection establishment overhead. Table 3 shows the total bytes transferred, from request until data frame is received. Again, as in Fig. 3, subtracted the connection establishment overhead.

Table 3. Total bytes transferred from requesting data, after connection is established.

Data bytes	CoAP plain	OSCORE	DTLS	HTTPS/1.1	HTTPS/3
25	133	158	178	440	797
100	209	233	254	516	953
500	609	633	681	916	1753
2500	3071	3196	2655	2916	2973

5 Interpretation of Data

From Fig. 3 is seen a very large overhead of establishing connections over HTTPS. A surprising factor, however, is how much of a different there is between the HTTPS/3 and HTTPS/1.1 protocols. We would expect that HTTPS/3 should be faster than previous generations, partly by eliminating the TCP overhead, but at least for small amounts of data this is not the case.

In regard to OSCORE, Fig. 3 and Table 2, shows that OSCORE has a significantly lower overhead compared to the TLS based protocols. This is partly due to encrypting the content via a shared symmetrical key as no official methods of key negotiation has been decided yet. This means that there is no key negotiation at all in the test-setup of OSCORE and both sides already know the key to be used before the first communication between them. Figure 4 shows an interesting perspective however, as round-trip time when requesting data is actually significantly higher with OSCORE compared to DTLS and even HTTPS/1.1. This is in contrast to the speed increase that OSCORE has been observed to deliver, according to earlier, different performance measurements like in Fig. 5.

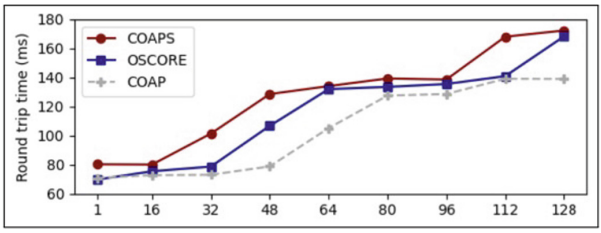

Fig. 5. Alternative measurement of OSCORE, CoAP/DTLS (COAPS) and CoAP Plain text (COAP) Round Trip Time. From: "Evaluating the performance of the OSCORE security protocol in constrained IoT environments" [12].

After a certain data size, the Californium implementations of DTLS and OSCORE fragment their packets automatically, disregarding the fact that the network can accommodate larger packets. In very constrained environments, the HTTPS header sizes might cause a problem. HTTPS headers can be very large and include a lot of, perhaps, unnecessary data [18]. Interesting to note is that the HTTPS/3 library, flupke, also fragments packets. While we see a large increase in data transfer time of OSCORE data transfers, the same issue does not occur with HTTPS/3. This is caused by the same thing that causes the HTTP/3 library to be significantly slower in establishing connections: HTTP/3 is capable of, and by default, opens multiple data channels between the client and server at once.

Having multiple data channels provides a significant boost when transferring large amounts of data between two end points, something that is often required in a desktop browser environment, and this is also why HTTPS/3 does not increase

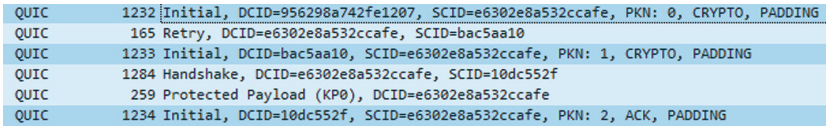

Fig. 6. Wireshark capture of flupke and Caddy establishing HTTPS/3 connection over QUIC in the test setup.

transfer time significantly when fragmenting data packets. Opening multiple data channels, however, also proves to be very disadvantageous when establishing a connection to a small device in a constrained network environment as large amounts of data are transferred in order to establish multiple data channels for no real benefits when transferring small chunks of data. Figure 6 shows multiple connections being established between the Caddy Server and the flupke HTTP/3 java library.

In the QUIC RFC [8], an arbitrary number of streams are allowed, this means that switching to a different implementation of QUIC might result in significantly lower overhead.

6 Ease of Use and Implementation

While implementing the performance measurement setup, one thing, specifically about OSCORE and HTTPS/3 became very clear. The protocols are newly defined and are still not in an entirely mature state. This also means that there is only a very small range of implementations that are actually working - and even between those that are working there seems to be significant caveats stopping the development of a functional application based upon these libraries.

For OSCORE, a few different libraries exist. Californium, CoAP.NET, aiocoap and libOSCORE have existed for a while and should be capable of a basic implementation of OSCORE [4]. Practically testing and working with these libraries, however, reveal that many of them are stagnant in a state where they cannot be implemented and have been for many months. Californium as a library cannot easily be compiled out of the box to a single functional library. Coap.NET no longer has a functional implementation of OSCORE. Aiocoap, while seemingly should have a basic implementation, cannot establish a connection between server and client and libOSCORE has never reached a maturity point where it can be implemented without significant development, even on its target platform of RIOT-OS.

The state of HTTPS/3 seems to be slightly better with more active development behind it. It does however suffer some of the same issues as OSCORE, being a new protocol and thus perhaps not mature enough. For Java, only a single implementation of a HTTPS/3 library seem to exist and only two serious attempts at QUIC implementations (kwik and Quiche4j), and of those two one of them, Quiche4j is also stagnant in a non-functional state.

Having no actively and fully developed libraries present, we got problems with these protocols - How can we be certain they are secure? How can we be certain they uphold the RFC itself? With these questions in mind, even if the specification of a protocol is secure, how can we be certain the implementation is secure as well?

7 Security of Protocols

Choosing between the different communication and encryption protocols results in a few different security concerns. In this section we will give a more in-depth overview of pros and cons for the different choices.

7.1 TLS

TLS is the backbone of HTTPS communication. As HTTPS is an extremely widespread communication protocol at this point, TLS is subject to a lot of different attempted (and sometimes successful) attacks. Some of the more recent large scale security breaches we have seen is for example the heartbleed vulnerability in OpenSSL [19]. It would allow an attacker to send a heartbeat with a message that was shorter than the stated length. A heartbeat is answered with the same message and same length. Because of this, the OpenSSL library would try to respond with the same message, but additionally also include any data stored in memory beyond the received message, in order to live up to the message length requirement. Leaked data can include sensitive information like credit cards, passwords and alike as that communication is all usually decoded in OpenSSL.

Many TLS attacks, like the aforementioned heartbleed attack, are directed towards the specific TLS library and not the TLS protocol itself. With TLS 1.3 in general, there are no known major security flaws, however this is not the case with older versions of TLS [20]. As long as an updated and widespread library is used and only the latest TLS is allowed, data transferred over TLS can be considered secure.

7.2 DTLS

DTLS is supposed to be almost entirely identical to TLS and the original RFC of DTLS is indeed only a description of the modifications to TLS. Because of this, in theory there should not be any security issues that are not present in TLS. It has, however, been proven that this is not the case.

DTLS is in many cases implemented entirely from scratch in security libraries, and being an extension to TLS, it can still claim to have the same security benefits and risk factors as TLS. The DTLS is often implemented separately from TLS, however, resulting in versions of DTLS that at times are lacking in patches to known security issues that are being discovered and fixed in TLS [21]. DTLS also follows the TLS versioning with DTLS 1.3 currently in proposal [22].

As DTLS 1.3 is still only proposed and not realized yet, though TLS 1.3 was released in 2018, current DTLS 1.2 implementations only comply to the security standards of the corresponding older versions of TLS, and not the widespread TLS 1.3 [23].

Another shortcoming of DTLS compared to TLS is referred to in the proposed DTLS 1.3 RFC itself: "The DTLS 1.3 protocol is intentionally based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees with the exception of order protection/non-replayability [22]". This means DTLS does not by default enforce protection against replay and reordering attacks, it is entirely optional in a full implementation of DTLS whether countermeasures should be taken against these types of attacks. Replay attacks can be a big security threat that can result in wrong, or perhaps even intentionally harmful information being delivered. An example could be an electric kettle that was told to turn off by the owner. A malicious third party could at an earlier point have sniffed the packet turning the kettle to 100° and 'replay' this message in order to turn on the kettle against the wishes of the owner.

7.3 OSCORE

OSCORE is a very different approach to securing data than the other mentioned protocols. Instead of encrypting the entire data stream, all of the headers of the CoAP or HTTP protocol are still transmitted entirely in plain text. This allows for a small amount of header data from the protocols to be leaked.

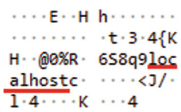

Fig. 7. The ASCII values of a local OSCORE packet. Underlined is the URI included in the "Uri-Host" CoAP option.

For example in Fig. 7 the host URI of the data requested is shown. This can be a security issue, as data encrypted only with OSCORE will leak different CoAP options, some worse than others. The "Uri-Host" for example, is for the proxies to forward the message and might leak the requested domain. Furthermore OSCORE also does not encrypt the CoAP token that identifies a response message from a request. This allows a malicious third party to identify responses to specific messages.

OSCORE is also lacking perfect forward secrecy as seen in Table 1. This effectively means that if the encryption key for the message is leaked, the entire conversation history can be decoded.

The two TLS based protocols, HTTPS and DTLS, support Perfect Forward Secrecy, which allows the communicating parties to exchange new encryption keys on a regular basis. This means that intruders that gain access to a single

key will only be able to deduce a small portion of the data. Meanwhile, OSCORE does not have such an implementation, so a potential key breach is a large problem [2]. In regard to proxy vulnerabilities, all aforementioned protocols except OSCORE will have potential security breaches for each proxy station. HTTPS, however, has an advantage in that the receiving end is often HTTPS-based and therefore this issue affects DTLS the most.

Unlike DTLS, OSCORE does provide protection against replay attacks and for OSCORE this protection is not opt-in. With the aforementioned CoAP tokens it can also bind responses to specific requests which provides protection against maliciously injected responses as those would be discarded for a wrong or missing token.

As OSCORE is a relatively newly proposed protocol which currently is in production at very few locations, not a lot of security research has been done in regard to investigate vulnerabilities of OSCORE. While DTLS suffers from a lack of active development, so does OSCORE to a much bigger extent. Currently a very limited amount of libraries are available and not a lot of active development is being done on OSCORE. This results in a platform where, even if the protocol itself is secure, it is likely that vulnerabilities in the libraries will be discovered, but not immediately patched, leaving a gap in security.

8 Discussion

From our results we see that HTTPS has a higher overhead and processing time than any of the other protocols. Especially in cases where small amounts of data is to be transferred, HTTPS seems to be a counter intuitive choice considering the alternative CoAP solutions available. That being said, while the overhead of HTTPS might burden the device more than CoAP, there is a significant cost with introducing CoAP to the device as the amount of available libraries are severely limited compared to HTTPS. The DTLS based CoAP seemingly performs better than the OSCORE based solution in our tests, currently seems to be only a single entirely functional implementation of OSCORE, which is in Java, designed for desktop computers. No further tests have been made to verify this. While both the HTTPS and CoAP/DTLS based solutions will have to mitigate the same issue of proxies having to decrypt and reencrypt data, the proxies are often introduced in specifically in order to translate between CoAP devices and HTTPS-based devices. Considering this, while the same security issue exists in both of these platforms, it is likely a bigger issue in regard to CoAP than HTTPS.

Many IP-based devices have spare resources. In those cases it can be argued that the benefits of running a full HTTPS client on the device may outweigh much the potential gains of implementing something like CoAP. HTTPS will eliminate some of the reasons to use a proxy between the IoT device and the cloud service. Furthermore, the HTTPS protocol and libraries have a tendency to be more well-documented and widespread, which can ensure the security of the device and add stability. Devices in today's market are becoming increasingly

more powerful and the networking solutions are becoming increasingly more performant with no significant battery hits, and if this trend continues, reasons for selecting CoAP might be worth considering.

CoAP is designed for IoT applications in very constrained environments. An argument can also be made that perhaps even using IP is, to a certain extent, overkill. Many different wireless protocols like Z-Wave, SigFox and LoRaWAN does not support IP at all, but instead implement their own protocol stack. In many cases, connecting device to an IP-based network might not make sense, which is also something to consider before investing in CoAP technology.

In the cases where IP is desired on a device that does not have resources to spare for HTTPS, it can be worthwhile to consider CoAP with DTLS or OSCORE. In these cases the potential use case must be considered. In some cases, where for example proxies are required to transfer data, the OSCORE model might make more sense, but in other cases the added security benefits of a more well-tested and perhaps secure protocol like DTLS might make more sense.

For scenarios where security has high priority but network resources are not abundant, it can also be worthwhile to consider that OSCORE and DTLS are not mutually exclusive. It might be a viable solution to encrypt packets with both OSCORE and DTLS. This will add load to the CPU, but the overhead in terms of total bytes transferred and time on the air will not be increased dramatically. This would allow for the replay and proxy protection of OSCORE while also pertaining the benefits of the DTLS protocol encrypting the entire message and adding perfect forward secrecy.

The tests show a general trend towards HTTPS/3 over QUIC being even heavier than HTTPS over TCP, so for limited devices, the HTTPS/3 protocol does not seem to make sense, though this might also be slightly due to the implementation of HTTPS/3. The QUIC protocol is designed with multiplexing of data streams in mind - and not a general optimization of overhead, which is clear from the results.

9 Conclusion

In this paper we have tested multiple different networking protocols, some designed with limited devices in mind, some not so much. Keeping in mind the test setup was not as constrained, We found that CoAP based solutions are faster and more performant than HTTP based solutions. Furthermore we found no significant performance advantages of OSCORE compared to DTLS, especially if establishing the connection is not factored in. We would also argue that HTTP/3 while implemented on top of UDP does not provide any significant advantages compared to HTTP/1.1 over TCP, but rather increases overhead and limits performance dramatically.

While the test results do indeed show advantageous conditions with regard to network overhead for the CoAP based protocols, the performance and networking overhead associated with HTTPS does not show enough of a disadvantage to justify using less widespread and potentially more insecure libraries that

are perhaps unfinished or stagnant at best. Furthermore, it shows no significant advantages of using a CoAP/OSCORE combo over a HTTPS solution in a product, unless a very constrained network or MCU is being used, and in many of those cases, using the IP stack instead of other protocols specifically designed for limited devices might be worth reconsidering altogether.

In cases where CoAP is the chosen solution, we argue that it is possible to protect data with a combination of both DTLS and OSCORE as they can be used in combination rather than as alternatives.

References

1. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (CoAP). RFC 7252. IETF (2014)
2. Gündoğan, C., Amsüss, C., Schmidt, T.C., Wählisch, M.: IoT content object security with OSCORE and NDN: a first experimental comparison. In: Proceedings of 19th IFIP Networking. IEEE Press (2020)
3. Amazon Web Services Inc: Device communication protocols. https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html. Accessed 23 Jan 2022
4. Jensen, J.C., Stormholt, A.R.: Survey and evaluation of OSCORE security and implementation. Unpublished Paper, Technical University of Denmark (2021)
5. Selander, G., Mattsson, J., Palombini, F., Seitz, L.: Object security for constrained RESTful environments (OSCORE). RFC 8613, IETF (2019)
6. Fielding, R., et al.: Hypertext transfer protocol - HTTP/1.1. RFC 2616, IETF (1999)
7. Rescorla, E.: HTTP Over TLS. RFC 2818, IETF (2000)
8. Iyengar, J., Thomson, M.: QUIC: a UDP-based multiplexed and secure transport. RFC 9000, IETF (2021)
9. Gündoğan, C., Kietzmann, P., Lenders, M., Petersen, H., Schmidt, T.C., Wählisch, M.: NDN, CoAP, and MQTT: a comparative measurement study in the IoT. In: Proceedings of ACM ICN. ACM (2018)
10. MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. OASIS standard, 07 March 2019
11. Rescorla, E., Modadugu, N.: Datagram transport layer security version 1.2. RFC 6347, IETF (2012)
12. Gunnarsson, M., Brorsson, J., Palombini, F., Seitz, L., Tiloca, M.: Evaluating the performance of the OSCORE security protocol in constrained IoT environments. ScienceDirect (2020). https://www.sciencedirect.com/science/article/pii/S2542660520301645. Accessed 23 Jan 2022
13. Hristozov, S., Huber, M., Xu, L., Fietz, J., Liess, M., Sigl, G.: The cost of OSCORE and EDHOC for constrained devices. arXiv:2103.13832 (2021)
14. Eclipse Foundation: Eclipse Californium CoAP framework. https://www.eclipse.org/californium/. Accessed 23 Jan 2022
15. Caddy Server Project. https://caddyserver.com/. Accessed 23 Jan 2022
16. QUIC-Go project. https://github.com/lucas-clemente/quic-go. Accessed 23 Jan 2022
17. Flupke Project. https://bitbucket.org/pjtr/flupke. Accessed 23 Jan 2022
18. Gadient, P., Nierstrasz, O., Ghafari, M.: Security header fields in HTTP clients. arXiv:2111.03601 (2021)

19. CVE Mitre database. https://www.cve.org/CVERecord?id=CVE-2014-0160. Accessed 12 Apr 2022
20. Stebila, D.: Attacks on TLS. https://www.douglas.stebila.ca/files/research/presentations/tls-attacks/tls_attacks_table.pdf. Accessed 23 Jan 2022
21. AlFardan, N.J., Paterson, K.G.: Plaintext-recovery attacks against datagram TLS. https://www.isg.rhul.ac.uk/kp/dtls.pdf. Accessed 23 Jan 2022
22. Rescorla, E., Tschofenig, H., Modadugu, N.: The datagram transport layer security (DTLS) protocol version 1.3. draft-ietf-tls-dtls13-43, IETF (2021)
23. Qualys SSL Labs: SSL Pulse. https://www.ssllabs.com/ssl-pulse/. Accessed 23 Jan 2022
24. Ahmed, M., Akhtar, M.M.: Smart home: application using HTTP and MQTT as communication protocols. arXiv:2112.10339 (2021)