Assessing the performance and security characteristics of QUIC in the context of computationally constrained spacecraft

Candidate no. 1063307

Word count: 18908 (measured by Texmaker)

Submitted in partial completion of the

MSc in Advanced Computer Science

Trinity 2022

Abstract

As satellite hardware configurations progress in terms of compute power, and the number of orbiting satellites rises, there is a need for a modern space communication security standard. While there are space security standards that use symmetric cryptography, we believe that satellite hardware is capable of using asymmetric cryptographic techniques that provide forward secrecy. To deal with potential network issues such as packet loss that are specific to satellite links, and to be able to make use of session-based modern asymmetric cryptographic methods, we propose the use of an application-layer protocol called QUIC. We test the viability of using this protocol in a realistic flight scenario by integrating it in a flight software stack developed by NASA, and measuring the compute time and memory usage overhead it generates when compared to an existing space security protocol that uses symmetric cryptography. We assess the feasibility of using the QUIC protocol in harsh network conditions by artificially inducing significant packet loss and latency. We discuss advantages and disadvantages and conclude that the newly presented protocol holds promising value.

Contents

List of Figures

List of Tables

1 Introduction

In the past decade, one of the most significant and impactful changes on the web was the rapid adoption of HTTPS and TLS (Transport Layer Security). As internet usage grew rapidly, the importance of securing potentially sensitive traffic became apparent to both companies and consumers. In 2015, about 60% [1] of the traffic handled by Chrome on Windows was unencrypted. The adoption of web security was so swift that at the beginning of 2022, more than 90% of the traffic handled by Chrome on Windows was encrypted [1]. Web security thus evolved from an unimportant afterthought to being mandatory and tightly integrated into the latest HTTP/3 standard specifications, published in June 2022 [2].

Although the vast majority of data travels across the globe through terrestrial fibre optic cables, a small yet significant portion of information is transmitted via Earth orbiting satellites. As the overall digital communications volume continues to grow, so does the quantity of data routed through these satellites [3]. Part of the increase in the volume of satellite communications could also be attributed to the recent growth of low-earth orbit satellite constellations such as Starlink [4], which offer lower latency connectivity at consumer prices. Besides facilitating internet access in isolated or war affected areas, satellites are also used to gather large quantities of scientific data that needs to be transmitted back to Earth for processing.

The same need for data encryption and authentication that led to the creation and fast adoption of new HTTP protocols with out-of-the-box security support also applies to satellite communications. Arguably, because of the potentially increased density of sensitive information and relative ease of intercepting radio communications compared to terrestrial communications, satellite security might even be considered to be more important than its terrestrial counterpart. When satellite connections are not properly secured, malicious actors with the right tools and expertise could take over and lock out the rightful owners, causing them to permanently lose access to expensive hardware.

That being said, there are many satellite security methods that are in use right now, including frequency hopping [5], which is mainly used in tactical communications, or the Common Scrambling Algorithm (CSA), mainly used to encrypt video broadcasts. While frequency hopping is a physical layer security method, and is quite expensive as a result of bandwidth licensing requirements, CSA is known to be vulnerable to side-channel attacks [6]. Besides many other potentially untrustworthy proprietary encryption solutions, the Space Data Link Security (SDLS) [7] protocol was proposed by the Consultative Committee for Space Data Systems (CCSDS) in an effort to provide a standardised way for securing space communications across NASA missions, and in the hope of improving international collaboration.

As sending payloads to space becomes financially accessible thanks to the development of reusable rockets, more and more different organisations and individuals will deploy their own cubesats to orbit. As such, the need for a standardised way of securing the communication links of these satellites becomes apparent, and while the SDLS protocol is a good step forward, it only supports symmetric cryptography and is inherently less well tested when compared to the universally adopted TLS protocol, for example.

The proposal and adoption of the SDLS protocol and the fact that modern space missions such as SpaceX's Falcon 9 reportedly use multiple x86 processors and run on modified linux distributions [8] indicate that spacecraft hardware has progressed

to be able to handle software based asymmetric cryptography. As such, in this paper we aim to assess the viability of using modern asymmetric cryptography in spacecraft communications by comparing its performance to a public implementation of the SDLS protocol, which uses symmetric cryptography.

Asymmetric cryptography protocols such as TLS use key exchange algorithms in order to generate session keys for later use in communication. Although we refer to TLS as an asymmetric protocol, it also uses symmetric cryptography for all later communication performed after the session keys are generated. Generally, TLS is used on top of TCP connections, as the intuitive correspondence between TCP streams and TLS sessions makes working with these protocols easier: after a TCP connection is established, a handshake is performed to establish the symmetric keys to be used for the remainder of the connection. TCP connections are established by a three-way handshake, which could be costly in a high latency satellite network. To avoid connection establishment delays and other TCP-specific problems such as head-of-line blocking, UDP could be used instead. UDP also comes with disadvantages, such as the fact that it is not stream-orientated and the fact that packet loss must be handled at the application layer, but there is indication that it is more popular as a protocol choice for satellite communication.

As such, we think that QUIC, an application layer protocol that is the basis for HTTP/3 and that is built on UDP, is a good fit for loss-prone satellite connections. This protocol is tightly integrated with the latest TLS 1.3 standard and provides 1-RTT cryptographic handshakes, essentially eliminating TCP's connection establishment overhead. Moreover, it can be tweaked in multiple ways to optimise data throughput on lossy space links.

We make use of the TLS 1.3 integration and general features of QUIC in order to analyse the performance of asymmetric cryptography in a more realistic flight scenario. This will be done by integrating both SDLS and QUIC in the context of NASA's *core Flight System* (cFS) [9], an open source spacecraft software platform that aims to serve as a ready-to-use starting point for space missions. Furthermore, we discuss the performance-security trade-offs between a number of cryptographic

ciphers supported in the TLS 1.3 standard. To better grasp the computational viability of using TLS in small spacecraft, we measure the memory footprint and compute time of two popular TLS libraries. Finally, we explore how an untuned configuration of the QUIC protocol would handle satellite network conditions, namely high latency and packet loss.

1.1 Contributions

The main contributions of this paper are as follows:

- We created a functional implementation of the QUIC network protocol for the core Flight System.
- We showed that the QUIC protocol behaves adequately under space link conditions that involve significant packet loss and high latency.
- We measured the memory usage overhead of using QUIC in the core Flight System and compared it to the overhead generated by using the SDLS protocol.
- We compared the symmetric cryptography performance of the SDLS protocol to two different TLS 1.3 implementations.
- We assessed the viability of performing a TLS 1.3 handshake in a resource limited space environment by comparing the compute time required to complete the handshake to the compute time required to send packets over a channel secured via symmetric cryptography.

1.2 Structure of the paper

This chapter motivates our work and states the main contributions of this paper. In the next chapter, we further describe the systems and protocols we used as part of our work. In Chapter 3, we present the software setup we used to perform our experiments and detail how we implemented the QUIC protocol in the complex cFS software stack. In Chapter 4, we analyse how QUIC behaves under space

link conditions, compare its cryptographic performance to the SDLS protocol and analyse its memory usage.

2
Background

This chapter will introduce NASA's *core Flight System* (cFS), which provided us with a realistic spacecraft software environment in which to conduct our experiments. Then, we will briefly discuss the Space Data Link protocols proposed by CCSDS, along with the SDLS security protocol proposed by the same committee. We will discuss how the Transport Layer Security (TLS) protocol works, and what changed in the latest version of this protocol. Finally, we will succinctly present some of the features of the QUIC protocol.

2.1 core Flight System

The common practice for software development for new NASA spacecraft was to simply copy the code of an older mission and use it as a starting point for the new one [10]. Although this worked for some time, it also involved a significant amount of reimplementing the same functionality. The reason for this was either that the new flight software team was unaware of previous missions that shared requirements with theirs, or that the requirements of the old mission differed slightly, but the old code was too specific.

As such, in an effort to reduce the costs of developing flight software for new missions, the Software Engineering Division at Goddard Space Flight Center

analysed existing flight software and started work on cFS in 2005 [10]. In February of 2015, cFS was completely open-sourced [11], and its source code is now available on GitHub [12] for anyone to use.

Hardware that is sent into space is often designed to be radiation tolerant, which means that it is constrained in terms of computational power and available memory, and is also more expensive than traditional computers. If radiation tolerance is not a requirement, more powerful but less reliable hardware might also be used. That being said, cFS was designed to be efficient with the resources it uses. Along with FreeRTOS [13] (a real-time operating system for embedded devices), cFS fits on just 800KB of flash storage and 2MB of RAM [9]. To facilitate the constraint-considerate software engineering and design that was needed in order to achieve this small footprint, cFS was written in the C programming language.

2.1.1 Layered design

The core Flight System was designed in a layered manner, so that individual components can be more easily maintained and updated. The layered design also helped to make cFS compatible with multiple different operating systems. Thus, one of the layers handles OS abstraction and essentially hides the OS-specific implementation details to offer higher-level APIs for different functionality, such as sockets or shared memory.

On top of the OS abstraction layer is the core Flight Executive (cFE) collection of reusable flight services. Amongst the modules included in cFE is the Executive Services module, which contains functionality that handles the initial startup and setup of the main program, and the creation and management of other tasks, for example. Another notable module included in the core Flight Executive is the Software Bus service. This is a key part of the design of cFS, and is the only way that modules within cFS can communicate with each other. More specifically, the Software Bus uses the publish/subscribe paradigm and routes messages originating from one cFS module to any other module that requested messages of that type by subscribing to them.

Generic flight and mission-specific applications are the final layer of the cFS architecture. Examples of cFS applications include network uplinks and downlinks, data storage managers, and schedulers. Applications can essentially implement any functionality that is wanted, including acting as I/O interfaces to custom, mission-specific hardware.

2.1.2 Execution flow

The core Flight System was designed to not only be fault tolerant, but to also allow software engineers to rapidly iterate and test new applications. To accomplish this, the execution flow of all apps that are loaded into the cFS system must be precisely controlled. Having a main process that is aware of the state of execution of all running apps allows it to quickly determine when one of the apps is no longer behaving as it should be. Knowing when an app is not functioning according to its specifications further allows the main process to completely reset said app by unloading it and restarting it, thus reverting it back to a known initial state.

For this highly controlled application system to function well, applications must also interact with each other in a very controlled manner. As such, cFS applications can only interact with each other via the Software Bus module. When an application is initialised, it can subscribe to data that is published by some other application. These subscriptions are all handled by the Software Bus module, and are centralised in this manner. Therefore, if Executive Services decides that an application is misbehaving and wants to reload it, it can simply remove the subscriptions of said application from the Software Bus. This indirect manner in which applications interact with each other assures that any problem that might otherwise be caused by the fact that some application is unloaded from memory is avoided.

The requirement that cFS applications be precisely controlled by Executive Services translates into a general code structure that all applications follow. An example of such code structure is given in Figure 2.1.

The cFS software loads applications either as static libraries or dynamic libraries, and runs them on a separate OS thread. Although linking every module together

```
void My_App_Main() {
    My_App_Init(); /* initialise application, perform subscriptions */
    ...
    uint32 RunStatus = CFE_ES_RunStatus_APP_RUN;
    while (CFE_ES_RunLoop(&RunStatus) == true) {
        ...
        /* perform computations, poll I/O,
           process pending Software Bus messages.. */
        ...
    }
    CFE_ES_ExitApp(RunStatus); /* announce exit to Executive Services */;
}
```

Figure 2.1: General cFS application code structure.

statically can lead to various size reductions and performance optimisations by using techniques such as Link Time Optimisation, it means that code changes cannot be performed without a complete system reset. As such, code that is expected to be updated over time or that must be easily reloaded at runtime is loaded as a dynamic library. The system takes the required information to do so from a startup file. This text configuration file contains essential information such as a path to the object file to be loaded, an internal name, the name of the main entry function of the cFS application, and optionally a task priority and a limit of the allowed stack size of said application.

A cFS application would need to contain a main entry function. The code in Figure 2.1 shows how such an entry function called `My_App_Main()` might look like. Applications would usually also have a function, corresponding to `My_App_Init()` in this example, in which they perform various initialisations. These initialisations might also include subscribing to Software Bus messages, or perhaps setting a function to be used as a resource cleaning callback for when the application is unloaded.

Applications are usually event-driven, and as such the main while loop can be used to poll I/O, check the expiry of a timer, or process Software Bus messages, for

example. The main logic of a cFS app is the response to such an event and might end with the origination of another event for other applications to react to.

The continuation of the execution of some application is dictated by the information returned by the call to `CFE_ES_RunLoop(uint32*)`, which in this case is the condition of the while loop. This function is part of the Executive Services API and is used to instruct an application to stop in the event of an external command or if a fault is detected in the execution of the app.

Finally, `CFE_ES_ExitApp(uint32)` is called as the last instruction at the end of the main application function. If no error occurred, this function marks the cFS app as stopped, and enters a NOOP loop until Executive Services completely kills the thread.

2.2 Space Data Link protocols

The Consultative Committee for Space Data Systems [14] (CCSDS), is an international effort that aims to develop communication standards for use in space. At the time of writing, 11 member agencies are part of the committee, including the European Space Agency and NASA.

The committee has developed two widely-used communication standards: one that is mainly used to control spacecraft, called the Telecommand (TC) standard, and one that is mainly used for telemetry that is sent from the spacecraft back to the ground station, called Telemetry (TM). The Telecommand and Telemetry standards are Data Link protocols and are part of a more general classification referred to as Space Data Link Protocols by CCSDS. By default, the TM and TC protocols do not guarantee the delivery of packets, but there are other upper-layer protocols that CCSDS has developed, such as Licklider Transmission Protocol (LTP) [15], which can be used to achieve reliable data transfers.

2.2.1 Packet structure

Both the Telecommand and the Telemetry protocols have a common primary CCSDS space packet header. This primary header includes an application ID, a sequence

number, and the length of the payload, among other information. In addition to the primary header, there is also an optional extended header that can be used by missions to further classify packets. The size of the primary header is 6 bytes, and the size of the extended header is just 4 bytes. The TC and TM protocols also have their own separate extra headers, that contain different information between them.

Figure 2.2 illustrates how the 6-byte-long Space Packet Primary Header is structured. In this figure, the header is split into three main parts for easier visualisation: identifier, sequence and length.

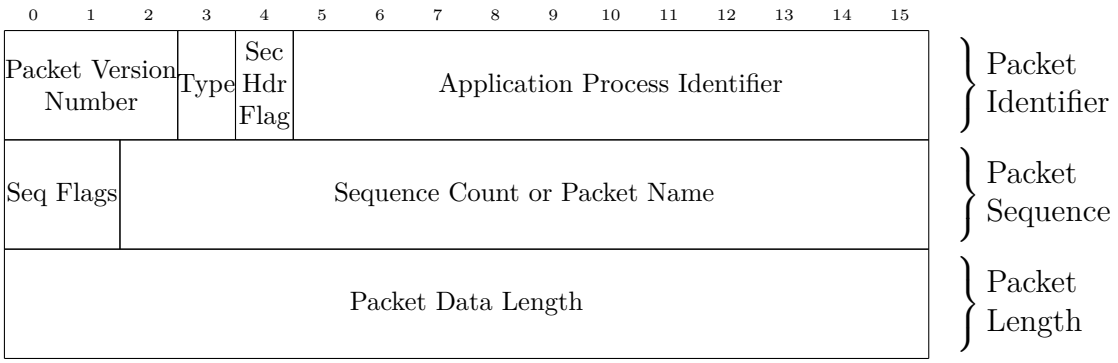

Figure 2.2: Specification of the Space Packet Primary Header, split in three 2 byte sections.

2.2.2 Security protocol

Following the observation that many space missions were developing their own security solutions on top of the Space Protocols briefly introduced in this section, the CCSDS Security Working Group began work on a standardized security solution for space communications [16]. More specifically, a data link security service that is compatible with the Telecommand and Telemetry standards was developed.

The security standard developed by CCSDS was named Space Data Link Security (SDLS). The security promises of SDLS are authentication, data confidentiality and integrity. The standard supports both plain authentication and plain encryption, but also authenticated encryption.

The AES (Advanced Encryption Standard) is a block encryption specification that can be used in multiple modes. One of the modes is called CMAC (Cipher-based Message Authentication Code), and it provides authentication. Another mode in which AES can be used is called CCM (counter with cipher block chaining message authentication code), and it provides authenticated encryption. The GCM (Galois/Counter Mode) mode also provides authenticated encryption. AES, much like other cryptographic algorithms, uses an IV (Initialisation Vector) that provides it with an initial and usually randomised state.

For the Telecommand protocol, CCSDS proposes a base mode of authentication only, which uses the AES-CMAC algorithm with a 128-bit MAC, a 128-bit key, and a 32-bit sequence number used for replay protection. For Telemetry, the baseline mode that is proposed is authenticated encryption using AES-GCM with a 128-bit MAC, 128-bit key and 96-bit IV. As the IV for AES-GCM can be an incrementing counter for example, the sequence number is not needed, and as such is not present in the packet.

Although SDLS only supports symmetric cryptography, it does have a tiered key system. As such, considering the intended usage of said system, a master key would be used to install, enable or disable different traffic keys. This kind of process is called Over-The-Air rekeying, and is now the standard for NATO secure communications.

Figure 2.3 shows a generalised SDLS packet structure, with all optional fields and trailer included. All packets that make use of SLDS security must contain a Security Header. This header contains a 16-bit security parameter index, which is used to indicate which security association to use. This indicates what cryptographic key and mode to use, and as such also dictates what the security header will and will not contain. The Security Header thus contains three more fields, all optional and of differing length (according to the configuration): the IV, a sequence number, and the length of the payload padding. The payload is padded because certain cryptographic algorithms consume messages in fixed size blocks, and as such the total message size must be a multiple of said block size. If authentication is used, the MAC of the message is stored in a Security Trailer located at the end of the packet.

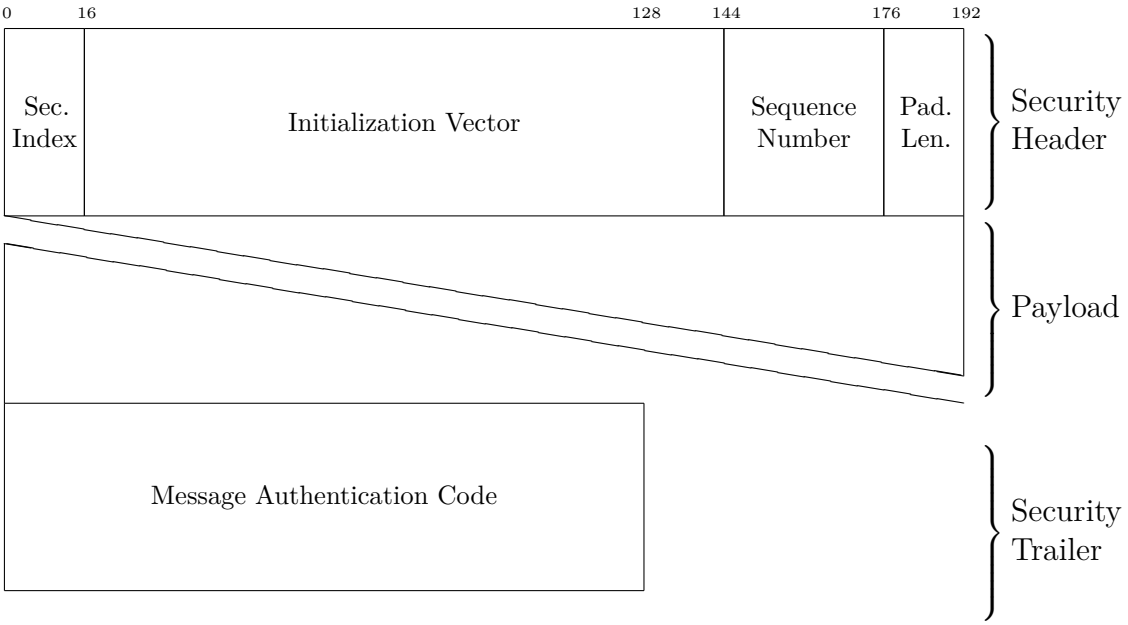

Figure 2.3: Generalised SDLS packet structure. Split into three main parts: Security Header, a variable sized Payload, and the optional Security Trailer.

2.3 Transport Layer Security

TLS, or Transport Layer Security, is arguably one of the most popular cryptographic protocols in use. It is used on almost all websites, but also in many other forms of internet communication, such as in mobile applications. Connections established using the TLS protocol benefit from confidentiality, integrity and authenticity. TLS is a stateful protocol, as it has an initial part dedicated to establishing a communication session. As such, one important part of the protocol is the initial handshake, in which the client and the server negotiate various details of the connection they are trying to establish.

The connection handshake is initiated by a client that would like to connect to a server. The first step is for the client to send a list of ciphers it supports to the server. Then, the server chooses one of the ciphers and sends its public-key certificate to the client. In turn, the client proves the identity of the server by verifying the certificate it received, and then the session-specific symmetric cryptography keys are generated. If forward secrecy is desired, then the Diffie-Hellman key exchange algorithm can be used to derive the session keys that will be used for the rest of the current connection. If forward secrecy is not required, then RSA can also be used.

Forward secrecy means that the session keys are independent of the server's private keys. This in turn means that even if one or multiple session keys are compromised for some reason, no other communication between the two parties will not be affected in any way. Authentication and key exchanges can either be performed via RSA for both, or Diffie-Hellman can be used for the key exchanges and RSA or DSA for authentication.

Both the RSA and Diffie-Hellman algorithms use many orders of magnitude more computational power than symmetric key algorithms. If RSA is used for both authentication and key exchange, there might be a computational advantage in the sense that the algorithm is performed only once. This is because the fact that the server correctly derived the session key indicates that it actually had access to the private key. On the other hand, elliptic curve Diffie-Hellman for the key exchange and an elliptic curve DSA certificate for authentication can be used to provide forward secrecy and might actually be faster than RSA [17].

After the session keys are exchanged, confidentiality in TLS is accomplished by using a symmetric cipher such as AES-GCM to encrypt all payloads. SHA (Secure Hashing Algorithm) is used to assure message integrity. As mentioned, authentication is done using either RSA or, for example, ECDSA (Elliptic Curve Digital Signature Algorithm).

2.3.1 TLS 1.3

In the newest standard of TLS, introduced in August 2018 by the Internet Engineering Task Force (IETF) [18], less secure methods are no longer supported. As such, only key exchange methods that benefit from forward secrecy remain supported in this standard, meaning that RSA is not an option any more. In addition to changes to the key exchange part of the protocol, TLS 1.3 also removed most of the ciphers that were available for use in TLS 1.2. The ones that remain are AES in GCM and CCM mode, and ChaCha20-Poly1305. Moreover, data integrity can now only be established using Authenticated Encryption with Associated Data (AEAD) methods.

As well as radically reducing the number of supported ciphers and no longer allowing RSA based key exchanges, TLS 1.3 also restricts the parameters that can be used with Diffie-Hellman in order to avoid users selecting unsafe parameters. Protocol negotiation is also simplified in TLS 1.3, as a server can only support a limited number of ciphers and only Elliptic-curve Diffie–Hellman (ECDHE) with the X25519 or P-256 curves for key exchange. This leads to a common trick for 1-RTT (round-trip time) handshakes in TLS 1.3 that simply involves sending the required handshake information in the first packet, and assuming that the server will support that ECDHE curve [19]. In the unlikely event that the server does not support said curve, the server can still let the client know which curves it supports with another packet.

TLS 1.3 even offers 0-RTT sessions, in which data can be sent in the very first packet. This is done using a resumption main secret that was derived during a previous connection and that is used to encrypt the data in the first packet [19]. The downside of 0-RTT in this context is that it allows for replay attacks. As such, it should only be used for GET requests for example, as they do not change the state of the server in any way.

2.4 QUIC

The QUIC protocol, standardized by IETF in May of 2021 [20], was designed to minimise the time it takes for a client to establish a TLS-secured connection to a server. It does this by using UDP instead of TCP, which immediately avoids the three-way handshake that is specific to the latter protocol. An additional motivating factor for the design of QUIC is a different TCP-specific problem called head-of-line blocking. This problem appears when a single connection is used for multiple requests, and the response to one of the first requests is lost in transit, causing the network stack to wait until the packet is retransmitted, instead of processing the responses to the other requests it sent using the same connection.

As QUIC uses UDP, and UDP is not inherently reliable like TCP, loss prevention needs to be handled at the QUIC application layer. As such, some TCP concepts

such as streams are also found in the QUIC protocol. Loss prevention algorithms based on requesting acknowledge frames at set packet intervals are used to assure timely loss recovery. QUIC streams are either bidirectional, meaning that both the initiator and the other party can send data, or unidirectional, meaning that only the party that initiated the stream can send data on it.

QUIC also allows for network path migration in the case of changes in the network topology, allowing clients to essentially change IPs and still maintain the state of their connection, without needing to reinitialise it. This is accomplished with randomly generated connection IDs that are used by the server to keep track of the client. In QUIC, a connection is an abstraction for the shared state between a client and a server.

As TLS 1.3 is a default and mandatory part of the QUIC protocol, connection establishment begins with a TLS handshake. The respective handshake and general crypto packets are framed in a different manner than that specific to TCP, but the result is the same. TLS 1.3 specific 0-RTT is also supported, and as such data can be sent in the first packet if a resumption main secret exists between the two parties.

A diagram showing the connection establishing handshake between a QUIC client and server is shown in Figure 2.4. After the handshake is done, the connection can be used to send and receive application data and other QUIC protocol frames.

Among the supported congestion control algorithms in QUIC are the Reno, CUBIC, BBR and BBRv2 algorithms. QUIC configurations that use one of the two BBR versions seem to perform better than the classic CUBIC algorithm in the context of GEO satellite networks [21].

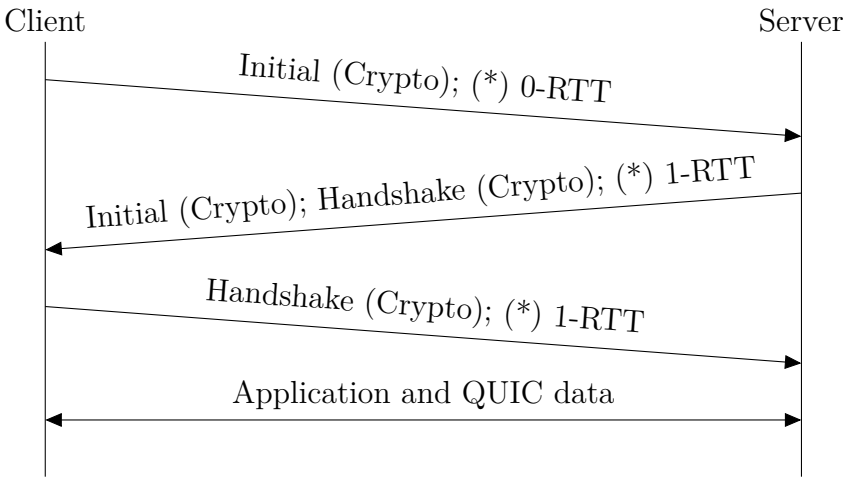

Figure 2.4: QUIC handshake sequence. 0-RTT and 1-RTT data is optional and as such is marked with an asterisk. Client is on the left and server is on the right. The client sends the first packet containing initial cryptographic data and optionally a 0-RTT payload. The server then responds with handshake specific cryptographic data and optionally a 1-RTT payload. Finally, the client sends the final handshake data and optionally a 1-RTT payload. After this, the two peers can communicate over the established session.

3
Software setup and implementation

This chapter will describe the process of integrating a security protocol benchmarking system for SDLS and QUIC into the core Flight System. Our motivation for using cFS as a starting point was the practical requirement that the security methods we present and test be usable in a realistic flight context. Moreover, there exists a public implementation of the popular SDLS protocol that was specifically designed to function in the context of cFS, which further encouraged us to choose this route.

We start by presenting an overview of the setup we used, and describe how we were able to integrate the various components we needed into the complex cFS software stack. Then we explain how we used a publicly available library to implement SDLS security, and the problems we encountered with said library. Finally, we describe the intricacies of using a complex and modular QUIC library in our context.

3.1 General overview

As discussed in the previous chapter, the core Flight System is a very complex software package, but it is also highly modular in nature. This means that we can quite easily add components to it, and that the single modules themselves

are not necessarily that complex, as the complexity of the system stems from the number of components that interact with each other.

3.1.1 Output and Ingest modules

Our goal was to identify how spacecraft running the core Flight System were intended to communicate with ground stations, and modify these components in order to add support for security in the form of SDLS and QUIC. The cFS software does not ship with these network communication components packaged by default. That being said, these two Output and Ingest modules are also created and maintained by NASA, and can be found online on GitHub [22] [23].

The two modules are called Telemetry Output and Command Ingest, and although they are indeed available on GitHub, they have not been updated since 2018. This is a problem only in the fact that the latest cFS development version v7.0.0, which we chose to use, was no longer compatible with the two communication modules. The two modules from 2018 offered the advantage of being modular in the sense that they allowed for plug-in network protocols to be used. By default, they supported UDP and RS-422, but other protocols could be easily added.

To overcome the incompatibility issue we just described, we used the Telemetry Output Lab and Command Ingest Lab applications. These two cFS applications come pre-packaged in the main cFS repository and although they have the word Lab in their name, a quick code inspection revealed that they function in a very similar way as the two separate modules. One significant difference between the two variants is that the Lab versions are not protocol agnostic and instead only support UDP. This is not a problem for our context, as we are interested only in UDP based communications. As such, we will refer to these two Lab cFS applications as Telemetry Output (TO) and Command Ingest (CI), respectively.

We have previously stated that all modules that run in cFS communicate through the Software Bus, and the TO and CI applications are no exceptions to this rule. The Telemetry Output app subscribes to various Software Bus pipes and simply forwards the messages it receives over the network. On the other hand, the

Command Ingest app listens on a predefined network port for any UDP messages, and simply forwards them as Software Bus messages. As such, all communications to and from cFS, which go via CI and TO, have a very specific format that is used by the Software Bus to identity where to forward said messages to.

Although the CI module automatically binds to a predefined port on startup, the TO module must first receive a command message via the Software Bus in order to start forwarding telemetry. This command contains the destination IP where TO should actually send the telemetry messages to. The Telemetry Output application also supports modifying which types of messages it should forward on the fly. It accomplishes this by implementing a number of possible commands that can be used to either add a certain type of message to the set of forwarded messages, or remove an existing one.

These two modules, TO and CI, represent the only way any other module in cFS can communicate over the network in our setup, and we believe a similar mode of operation is also used in practice, albeit not necessarily with UDP. As such, it made sense that we modify these two base modules to integrate SDLS and QUIC in the communication between cFS and a ground station. We will describe in detail how this was accomplished in the next sections of this chapter.

3.1.2 Benchmark Module

We have identified and added the Telemetry Output and Command Ingest components that are used by all the other cFS modules to communicate with external peers via UDP, as described in the previous section. The next step in building a functional benchmarking setup was to implement some logic that handles sending and receiving a fixed number of packets in a set time interval.

It made sense for one part of this logic to be handled by another cFS application, which we named the Benchmark Module. This module was created from scratch but is not very complex in its functionality.

The Benchmark Module follows the same structure of general cFS application which was briefly discussed in the Background chapter on this paper. No I/O

is directly performed by this app, as network related operations are handled by the TO and CI modules discussed in the previous section. As such, the only manner in which external information can reach this module is via the Software Bus, which is repeatedly polled for messages in a while loop, as was the case in the example code we gave last chapter.

The module we implemented can receive a number of different commands through the Software Bus, which it can then process. The first command it can receive is a NOOP. This command is a NOOP in the sense that it does not trigger any action from the module apart from the fact that it is received by it via the pipe.

The second command our module can receive is what we have called a data command, as it contains a fixed sized arbitrary data payload. This command was created to help us measure the compute time of the different security methods we want to test. The command is directed to the Benchmark module we created.

Finally, we called the third available Benchmark module command the start command, as it signals the app to start publishing a predetermined number of packets on the Software Bus, with a fixed time delay between them. The packets the module would publish are Telemetry messages with a specific message type. This message type, in this case specific to the Benchmark Module, makes these packets distinct from all the other messages that could be created by some other components of cFS. It is intended that the Telemetry Output (TO) module we described previously would be listening to these specific packet types, and that it would then forward them over the UDP connection to the ground station.

3.1.3 Benchmark Tool

To complement the cFS module that can receive various commands and then start sending its own data, we needed a tool that could actually send the required commands to it. We call this tool the Benchmark Tool, and we can consider it to act as the ground station in a context where the cFS suite is the spacecraft. Unlike the Benchmark, TO and CI Modules, this tool is completely external of cFS, and is essentially just a normal C executable.

As such, this tool is used to send commands to cFS that will then be forwarded by the Command Ingest module to their final destination, be that the Benchmark Module or some other component. That being said, this is not the only use of the tool, as we also designed it to be able to listen to any incoming telemetry that is sent by the Telemetry Output cFS application.

We previously explained that the Telemetry Output module needs to first receive a command that instructs it where exactly to forward telemetry messages to. As such, the first command that our tool sends to the CI module is one that is directed to the TO app. This first command configures the TO application to send all telemetry to the Benchmark tool. The next command we send to cFS is also directed to the TO app, and instructs it to add telemetry generated by the Benchmark Module to the set of messages it should forward.

After the Telemetry Output module is properly configured by the two commands that we mentioned, we can start properly communicating with the Benchmark Module. This means that we can send what we have called a start command, that will in turn trigger the module to start sending data packets back to us. Alternatively, we can send what we have named data commands to the Benchmark Module. Thus, the Benchmark Tool controls the flow of data packets both going to cFS and coming from cFS.

To be able to more easily integrate SDLS encryption and QUIC into the data flow, we decided to follow the good design choice of the already existing Command Ingest app when it comes to listening for data packets. As such, the part of the tool that listens to incoming telemetry does so in a while loop that continuously checks if there are any pending messages on the bound socket. To be more specific, the CI module does this in a finite for loop that is embedded within a while loop. The outer while loop in this case should not be broken unless an error happens, much like the while loop in the code example from last chapter.

Because we wanted to listen to telemetry in an infinite loop and still send the commands that we need to send to cFS, we decided to use concurrency. As such, we immediately fork the Benchmark Tool process on startup and make the resulting

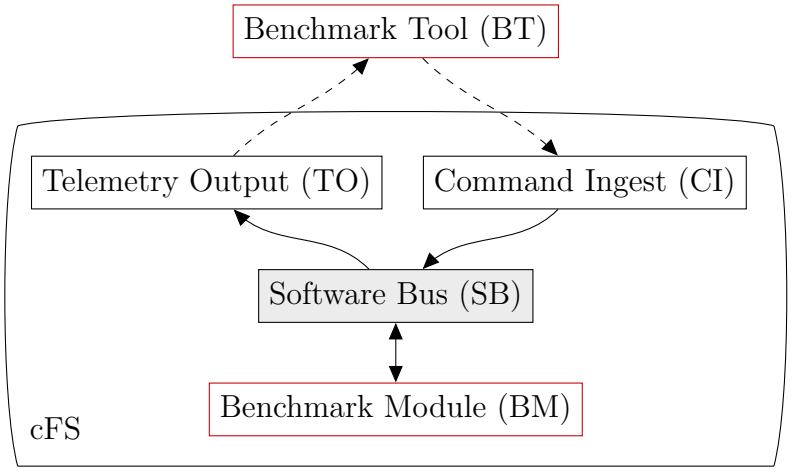

Figure 3.1: Component interaction. Divided in two sections, one for internal cFS components, and the separate Benchmark Tool. Dotted lines indicate communication via UDP. Full lines indicate communication via Software Bus messages.

child process listen to telemetry. We make the parent process send the commands to configure the Telemetry Output module, and then start the benchmark. We use shared memory to facilitate communication between the two resulting processes.

3.1.4 Data flow

Having explained the three cFS modules and the external tool we built for running benchmarks, we can now describe and illustrate the data flow in the system we set up. Later on in this chapter, we will show how we integrated QUIC in this system and compare the two versions.

Figure 3.1 illustrates how data flows between the cFS system and the Benchmark Tool. The Telemetry Output (TO) and Command Ingest (CI) modules can respectively send and receive data to and from the Benchmark Tool, over UDP. The Benchmark Module can both send and receive messages to and from the Software Bus. Although TO can write messages to the Software Bus, we are only interested in its ability to read them. The Command Ingest module is shown to be able to write messages that to the Software Bus in this figure. Many other cFS components communicate via the Software Bus, but they are not shown in the figure.

As such, for a certain command to reach the Telemetry Output module, it has to go through a number of hops. The command is initially sent by the Benchmark Tool

via UDP and reaches the Command Ingest module, where it is placed in the Software Bus with the Telemetry Output module as its destination. Similarly, for a telemetry message to arrive back at the Benchmark Tool, it first travels via the Software Bus to the Telemetry Output module. Here, it is sent via UDP to the Benchmark Tool.

3.2 Space Data Link Security protocol

We will now describe how we implemented the SDLS protocol in the Telemetry Output and Command Ingest modules of cFS, and in our Benchmark Tool. This was accomplished by using CryptoLib, a project started by NASA in October 2020 [24]. The purpose of this project was to build a software solution that implements an extended version of SDLS. Initially, as can be seen by looking at an earlier version of the code on GitHub [25], the library followed the code structure of a cFS module. At some point in the course of its development, which officially ended in September 2021 [24], the library became more general and thus no longer follows said structure.

Although the NASA project that initiated CryptoLib ended in 2021, the open-source community has not stopped contributing to the project on GitHub. As of the writing of this paper, the last commit on the main branch of the repository was pushed on the 23rd of May 2022. Although the project is still undergoing active development by members of the community, a very significant part of the project is still not finished. We will now refer specifically to the version of the library from May 2022 and give some examples of unimplemented functionality. Firstly, only the encryption of Telemetry frames is supported in the version that we mentioned, while the decryption functionality is completely unimplemented. There is another Space Link Protocol called Advanced Orbiting Systems (AOS) [26], and this protocol should have also been supported in CryptoLib, but it is not implemented at all.

CryptoLib does not directly implement the cryptographic algorithms that SDLS recommends. Instead, it uses Libgcrypt [27] as a backend for cryptographic operations such as encryption or authentication. Libgcrypt was initially developed as a module of GnuPG, an open source implementation of the OpenPGP standard, and later exported as stand-alone.

3.2.1 Security associations

As mentioned, CryptoLib implements an extended version of SDLS, that is called the SDLS Extended Procedures [28]. This extension to SDLS includes a key management system that allows for over-the-air rekeying and key activation and deactivation.

To achieve this, CryptoLib, as per the recommended SDLS standard, uses something called Security Associations (SA). This is essentially the security context that is used for communication over a certain virtual channel. A Security Association defines the encryption cipher, the authentication cipher, the IV length and start value, and other parameters. Security Associations could also be modified, disabled or created at runtime in this extended protocol of SDLS.

That being said, we do not make use of this available extended functionality, but instead create two active Security Association contexts that will not be modified at runtime. We intend to use the first SA for Telemetry data that is outgoing from cFS, and the second SA for Telecommand data that is incoming to cFS. Although CryptoLib has many configuration options, we leave the settings mostly on default and use AES-GCM with 256 bit keys. This decision to leave settings on default was also influenced by the fact that the majority of other options are not implemented. That being said, this provides us with authenticated encryption as per the SDLS guidelines.

Part of the code we used to initialise the first Security Association is shown in Figure 3.2. All fields are explained in comments. As we mentioned, we created another Security Association that we linked to security parameter index 1, and that we configured to be used with spacecraft id 1.

The SA shown in Figure 3.2 uses a key with id 130, and the other SA that is not shown was configured to use a key with id 131. These two keys are also manually defined in another part of the code.

3.2.2 Overcoming missing functionality

Although we did not initially plan on making use of the majority of what turned out to be unimplemented parts of CryptoLib, we did want to use the library for both

```
sa[0].spi = 0; /* security parameter index */
sa[0].ekid = 130; /* key id in keyring */
sa[0].sa_state = SA_OPERATIONAL; /* sate of Security Association */
sa[0].est = 1; /* encrypt if 1 */
sa[0].ast = 1; /* authenticate if 1 */
sa[0].ecs_len = 1;
sa[0].ecs = calloc(1, sa[0].ecs_len * sizeof(uint8_t));
*sa[0].ecs = CRYPTO_CIPHER_AES256_GCM; /* cipher to use */
sa[0].shivf_len = 12; /* security header transmitted IV length */
sa[0].iv_len = 12; /* local IV length */
sa[0].stmacf_len = 16; /* security trailer transmitted MAC length */
// allocate IV
sa[0].iv = (uint8_t *) calloc(1, sa[0].shivf_len * sizeof(uint8_t));
*(sa[0].iv + sa[0].shivf_len - 1) = 0; /* initialise IV */
sa[0].abm_len = ABM_SIZE; /* authentication bit mask. length 20 */
// allocate ABM
sa[0].abm = (uint8_t *) calloc(1, sa[0].abm_len * sizeof(uint8_t));
... /* ABM initialisation */
sa[0].arsnw_len = 1; /* anti-replay sequence number window length */
sa[0].arsnw = 5; /* anti-replay sequence number window */
// anti-replay sequence number length. unused with current AEAD mode
sa[0].arsn_len = 0;
sa[0].arsn = NULL; /* unused, as specified above */
sa[0].gvcid_tc_blk.tfvn = 0; /* transfer frame version number */
sa[0].gvcid_tc_blk.mapid = TYPE_TC; /* multiplexing mapping id */
sa[0].gvcid_tc_blk.scid = 0; /* spacecraft id */
sa[0].gvcid_tc_blk.vcid = 0; /* virtual channel id */
```

Figure 3.2: Part of the initialisation of a Security Association, modified from a CryptoLib template. Configuration shown uses key with id 130 to perform Authenticated Encryption using AES-256-GCM. It will be used for messages with spacecraft id 0 and virtual channel id 0, and is identifiable by security parameter index 0.

Telecommand and Telemetry packets. As previously mentioned, the only protocol that is in an usable state is Telecommand. Although we could have completed the implementation of the Telemetry format on our own, this would have been extremely time consuming and the risk of introducing bugs would have been very high.

As such, we found another solution that allows us to make use of the already

implemented functionality but also to send any kind of payload we want, instead of being limited to Telecommand packets. One downside of our solution is an additional 5 byte size overhead, but we believe that the benchmark results will not be affected by this in any significant manner.

The solution was to essentially encapsulate any generic payload by appending a custom Telecommand header to it. This header has a size of 5 bytes, thus explaining the size overhead we just mentioned. The only aspect that had to be considered when doing this was the fact that we needed at least two Security Associations, one that would be used for incoming data and one for outgoing data (observed from the perspective of cFS). As such, the header we appended was either a 5 sized sequence of 0 bytes, or the second byte was set to 1. The second byte of this header is used to determine the spacecraft id. As we mentioned, the two Security Associations we created corresponded to messages with spacecraft id 0 and 1. We decided to use id 1 for actual Telecommand payloads, and id 0 for Telemetry payloads.

The size of the payload of standard Telecommand packets is encoded in the packet header itself. The `Crypto_TC_ApplySecurity` and `Crypto_TC_ProcessSecurity` functions are provided by CryptoLib and intended to encrypt and respectively decrypt TC packets, while also taking care of authentication and other checks. The ApplySecurity function, which is used for encryption, looks inside the header of the input TC packet to determine the size of the payload, although it does also receive a size parameter as an input. To simplify things and keep being able to use an all-zero header, we modified this function to no longer do this, but to instead derive the size of the packet it should encrypt from the input size parameter. We also made a similar modification to the ProcessSecurity function.

3.2.3 Merging into cFS

As per Figure 3.1, in which we illustrated the data flow in our testing system, the Telemetry Output and Command Ingest modules are how other cFS applications interact with the external world. As such, it makes sense that we modify these two

modules to perform the necessary cryptographic operations before the packets are sent via UDP.

That being said, we also need to apply and process security at the other end, namely in our Benchmark Tool. As such, we created a library that is both compatible with the cFS structure, but that can also be used in an external context such as that of our tool. This library, which we have named the Net Library, includes some basic UDP socket functionality, some functions we used for printing debug messages, and other functionality. What is relevant for the purpose of this section are the `NET_encrypt` and `NET_decrypt` functions. These two functions are used to apply and process security, and are essentially wrappers over the CryptoLib `Crypto_TC_ApplySecurity` and `Crypto_TC_ProcessSecurity` functions, but that also handle the artificial Telecommand header we appended.

The Net Library we made has an initialisation function, `NET_LIB_Init`, which also initialises CryptoLib and sets up the Security Associations and keys as we have described in the previous section. As such, we can simply use this library on both ends of the communication, in cFS and in the external tool, in order to apply and process security, if needed. We designed the 2 `NET_encrypt` and `NET_decrypt` functions to support toggling between no security and SDLS-provided security with a simple preprocessor macro.

We modified the Telemetry Output module and the sending part of our Benchmark Tool to first pass the packet through the `NET_encrypt` function, and then send the output of this function over UDP instead. Similarly, we modified the Command Ingest module and the receiving part of the tool to immediately call `NET_decrypt` on all packets they receive, and only then pass the packets further. This enabled us to toggle between raw communication and SDLS without great difficulty.

3.3 QUIC

As the QUIC protocol is quite complex in its nature, it would have been inefficient and also risky for us to implement it from scratch. Our plan was to also test multiple TLS libraries and compare their performance and resource utilisation

when performing the cryptographic tasks needed by the QUIC protocol. As such, we chose to use a project that supports multiple TLS libraries, but that is also well-maintained and worked on.

The library we chose is named *ngtcp2* [29], and is publicly available on GitHub. It is modular in its nature, and it works using callback functions that are called when the library needs to perform certain cryptographic operations. The functionality that needs to be performed by these callbacks could be theoretically performed using any cryptographic library. That being said, the ngtcp2 open-source developers have also created compatibility layers with some cryptographic libraries, so that they can be more easily used together with the QUIC library.

As such, the ngtcp2 project has support for OpenSSL [30] and wolfSSL [31], among other popular cryptographic libraries. OpenSSL is one of the most popular and widely-used cryptographic libraries, but it does not have official support for the custom TLS 1.3 operations needed by QUIC. That being said, there is an independently maintained fork of OpenSSL version 1.1.1, that adds support for QUIC [32]. This is the fork that is supported by ngtcp2, and that we used.

We also used wolfSSL, which is a TLS library that was designed specifically for embedded systems and as such is more lightweight that OpenSSL or other general-purpose libraries. This library has also officially added support for the QUIC protocol very recently. In fact, the pull request [33] that was eventually merged into the main branch of the code was opened on the 21st of July 2022. This pull request was merged into the main branch on the 9th of August 2022. Shortly after, the community of ngtcp2 provided a compatibility layer between wolfSSL and their QUIC library, which made it relatively easy to use wolfSSL as an alternative to OpenSSL for QUIC.

3.3.1 Server specifics

Although the ngtcp2 library provides helper functions for interacting with the multiple TLS backends it supports, a significant part of the logic required to use it still needs to be written. As such, we wrote and adapted around 1000 lines

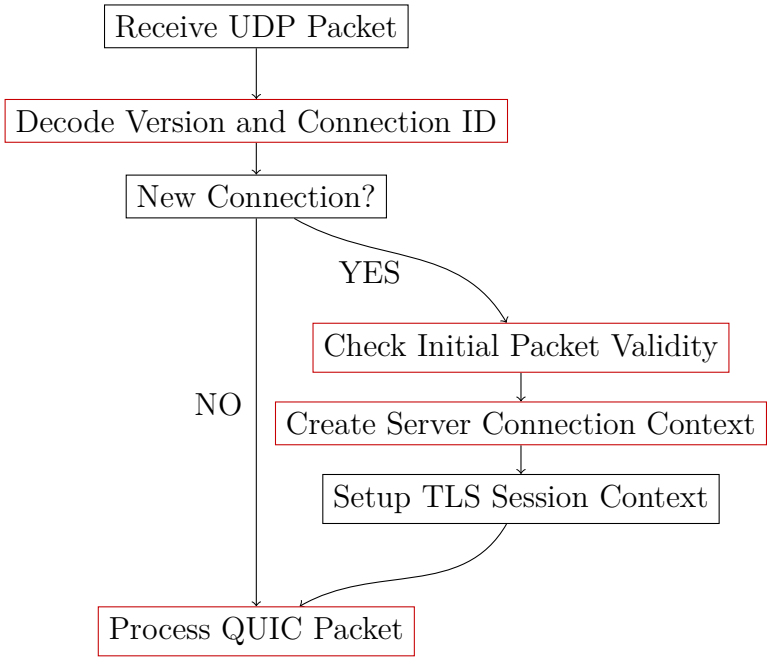

Figure 3.3: Execution flow triggered by a received UDP packet on the server side. Black boxes represent operations that happen at the level above the ngtcp2 library. Red boxes represent operations that are carried out by ngtcp2 library functions.

of code to further encapsulate the functionality of the ngtcp2 library for a single connection QUIC server and client. Our goal was to make two similar and easy to use interfaces that we could then easily integrate in cFS and into our Benchmark Tool. Following the C++ examples provided by the developers of the library, we created our own high level abstractions in C.

The ngtcp2 library was designed to be used in combination with event-based non-blocking socket logic. In essence, this means that the user application continuously checks for a signal that some socket is ready to be read from or written to. When the kernel decides that said socket is indeed ready to be acted on, it marks it accordingly. The user application observes that the socket is ready to for that action and starts executing its logic: either reads packets or writes them. The advantage of this approach is that the user application can still do any other computation it wants to when the socket is not ready to be read from, or when there is nothing to write to it.

Although the library also uses timers that need to be managed by the higher level application, the logic for them is quite simple. As such, we begin by presenting a diagram of what the server application does when a packet is received. Figure 3.3

shows the execution flow of the server application, presenting both the case in which the packet corresponds to an already known connection, and the case in which the server must begin the process of initialising a new connection. As can be seen by the differently coloured boxes, library functionality is interleaved with higher level functionality. For example, whenever the library creates a new connection context object, the application also creates an abstraction of this connection, and they are associated together.

As shown in Figure 3.3, a specific part of a QUIC server is accepting a new client connection. In fact, this is the only part that majorly differs between the client and the server. There are also some error states which are not shown in the diagram, such as if the initial packet does not contain the required information for example. Various error checks are spread throughout the whole code, but the diagram only shows what happens if no errors are encountered.

3.3.2 Client specifics

In general, QUIC servers can handle multiple concurrent connections. For simplicity of use and integration into the whole system, we only implemented functionality that allows us to handle a single connection at a time, from a peer client. While the server has a part that handles new connection requests that are identified by the fact that the connection ID was not seen before, the client is the one that actually initiates the connection establishment process.

The QUIC client library was designed to be able to connect to the server endpoint and open a data stream that it can then write any user payload to. In QUIC, there are both unidirectional and bidirectional streams. Our client implementation opens a unidirectional stream as soon as it is possible to do so. Only the peer that opened the unidirectional stream is actually able to send any data on it, as per the QUIC protocol specifications.

We mentioned that we open a unidirectional stream as soon as possible, and we do this by providing a callback to the ngtcp2 library. More specifically, we provide it a custom callback parameter, which is then called by the library when

the number of unidirectional streams can be increased. Inside of the function we supplied as a callback, we open a single unidirectional stream and associate it with an internal data context.

3.3.3 Common elements

The ngtcp2 library function that was at the end of the Figure 3.3 diagram and that is used to process an incoming QUIC packet in the server application is also used in a very similar way in the client. The difference comes from the fact that the client does not require the previous step that decodes the Connection ID from the packet, as there is only one connection. As is also the case when used in the server, this function handles the QUIC handshake, making sure that received cryptographic information is passed on to the TLS library, for example. That being said, the function in question, called `ngtcp2_conn_read_pkt`, does not perform any writing. Although the same function is used in both the client and the server, what it does internally depends on the connection object it is given. This object of type `ngtcp2_conn` stores the state of a connection, and is used both in the server and in the client context.

When the socket is ready to be written to, both if the peer has application data that it would want to send and otherwise, the `ngtcp2_conn_writev_stream` should be called. In the case that the user wants to actually write application data to the connection, a stream id must be specified as an input parameter to this function. This stream id would previously have been set by the user when the stream was opened. In our case, we open a unidirectional stream in the client, and as such only the client can use the previously mentioned function to actually write user data to this stream. On the other hand, the server does still use the same function to prompt the ngtcp2 library to write to the socket, but these writes simply do not include user data. What they do include is different QUIC-specific frames, among which are handshake frames, acknowledge frames and so on.

We previously mentioned the fact that the ngtcp2 library also makes use of timers. These can be quite easily integrated in the same event-based system that triggers events when sockets are ready to be read from or written to. Library

```
typedef struct {
    SSL_CTX *ssl_ctx; /* TLS library context data */
    Connection *connection; /* could be a list of connections */

    int epoll_fd; /* file descriptor for epoll handle */
    int socket_fd; /* socket file descriptor */
    struct sockaddr_storage local_addr; /* socket local address */
    size_t local_addrlen; /* socket local address size */

    ngtcp2_settings settings; /* ngtcp2 library settings */
} QuicServer;
```

Figure 3.4: The main data structure created in the QUIC server implementation. Fields are briefly explained in the comments next to them.

documentation specifies what to do when a timer expires, but the library itself does not actually create or trigger the timers itself. Instead, this is left to the user, and the library only specifies when these timers should be called by the user, as a result from the `ngtcp2_conn_get_expiry` function. When the timers do expire, we call `ngtcp2_conn_handle_expiry` and then the same function we use for writing to the connection, `ngtcp2_conn_writev_stream`.

Because this library was designed to be modular and compatible with multiple TLS backends, it ended up also being harder to use. We say that it is harder to use because the API it offers is quite low level, and complicated to use. That being said, the advantage of offering a lower level API is that it makes the library more robust and a good fit for more scenarios.

3.3.4 Implementation details

As we have previously mentioned, we wanted a very simple interface which we could then integrate in the complex cFS system. We now present the Client and Server interfaces we created, and describe how they work.

3.3.4.1 Server interface

```
void quic_server_deinit(QuicServer *server);
void quic_server_init(QuicServer *server, ngtcp2_recv_stream_data read_cb,
                      bool log_print);
void quic_server_bind(QuicServer *server, const char *host,
                      const char *port);
void quic_server_setup_epoll(QuicServer *server);
int quic_server_step(QuicServer *server, int steps);
```

Figure 3.5: The server interface functions. These will be used by a higher level application such as the Benchmark Tool.

Figure 3.4 shows the Server data structure. Comments were introduced to briefly explain what each field represents. As explained in a previous section, an event-based design was adopted, and as such we use `epoll` to know when our socket is ready for operation or when our timer has expired. Using epoll requires that we keep a file descriptor handle to manage it, which is what the `epoll_fd` field is for. There are three fields used to store socket-related information such as the file descriptor and the local address. We also created a `Connection` abstraction, which is used for connection-specific operations that are mostly shared between the Client and the Server. Finally, the `settings` field holds configuration parameters for the library, and the `ssl_ctx` pointer holds TLS library specific general data.

Figure 3.5 shows the declarations of the functions that should be used with a `QuicServer` object. The second function, `quic_server_init`, is used to initialise a `QuicServer` object it receives via the first parameter. This function configures the ngtcp2 library appropriately and creates a TLS library context which involves loading the server's private key and certificate from a hardcoded file path. The second parameter of this function is of function type, and if it is not null, the server will use it as a callback for when user data is received. That is, whenever a QUIC packet that contains user data is received, this function will be called. This is the intended way for data to be passed to the higher lever from this library level. Finally, the last parameter indicates whether or not debug printing should be active.

After the initialisation function described above is called, the intended use is for `quic_server_bind` to be called next. This resolves the host and port to an address and binds the UDP socket, setting the `socket_fd` field to the newly obtained file descriptor. After this is done, the user should call `quic_server_setup_epoll`, which opens the epoll file descriptor and subscribes to read and write status changes for the previously bound socket.

Finally, after the three initialisation functions that we presented above are all called and successfully return, the user can call the `quic_server_step` function in a loop. The second argument of this function, named `steps`, indicates how many times the function should also internally loop while checking for any events. Effectively, this function should be used to perform any pending QUIC operations such as reading from a socket, handling an expired timer, or writing to the socket. We show a simplified version of this function in Figure 3.6. There are three lines, corresponding to handling the three events, that are now shown entirely. `epoll_wait` is used with a last argument of 0 that indicates it should not block at all, to essentially ask the kernel if there are any pending events. The `nfds` integer it returns is the number of file descriptors that have one or more pending events. We use two if statements to determine if the file descriptor in question is the socket or a timer, and then process the event accordingly. Two more if statements are necessary to determine if the event triggered on the socket file descriptor is a read or a write, or both.

Finally, the `quic_server_deinit` function is used to free the `Connection` and `SSL` objects that were created in the lifetime of the Server.

3.3.4.2 Client interface

We present the main client data structure in Figure 3.7. Many of the fields are shared with the previously described server structure, but the client version is simpler.

Figure 3.8 shows the interface provided by our QUIC client implementation. The functions are very similar to those on the server side, with a few notable differences. The main initialisation function, `quic_client_init`, does not receive a read callback as a parameter, as our version of the client does not support receiving

```
int quic_server_step(QuicServer *server, int steps) {
    struct epoll_event events[4]; /* maximum of 4 events at once */
    for (int i = 0; i < steps; ++i) {
        const int nfds = epoll_wait(server->epoll_fd, events, 2, 0);
        for (int n = 0; n < nfds; n++) {
            if (events[n].data.fd == server->socket_fd) {
                if (events[n].events & EPOLLIN)
                    ... /* handle incoming packet */
                if ((events[n].events & EPOLLOUT) && server->connection)
                    ... /* handle writing to socket */
            } else if (server->connection &&
                   events[n].data.fd == server->connection->timer_fd) {
                ... /* handle timer expiry */
            }
        }
    }
    return 0;
}
```

Figure 3.6: A modified part of the `quic_server_step` function. Although this is not exactly how the function looks in reality, it can be used to understand how it works. The constant 4 used as the maximum number of events that can be received at once is arbitrary and can be changed.

```
typedef struct {
    SSL_CTX *ssl_ctx; /* TLS library context data */
    Connection *connection; /* one single connection */

    int epoll_fd; /* file descriptor for epoll handle */

    ngtcp2_settings settings; /* ngtcp2 library settings */
    ngtcp2_transport_params transport_params; /* extra ngtcp2 settings */
} QuicClient;
```

Figure 3.7: The main data structure created in the QUIC client implementation. Fields are briefly explained in the comments next to them.

```
void quic_client_deinit(QuicClient *client);
void quic_client_init(QuicClient *client, bool log_print);
void quic_client_setup_epoll(QuicClient *client);
int quic_client_step(QuicClient *client, int steps);
void quic_client_connect(QuicClient *client, const char *host,
                         const char *port);
int quic_client_write(QuicClient *client, const uint8_t *data, size_t size);
```

Figure 3.8: The client interface functions. These will be used by a higher level application such as the Benchmark Tool.

user data. That being said, it would be simple to implement this functionality in a similar manner as that in which the server handles it.

Furthermore, instead of the bind function offered in the server context, the client has the `quic_client_connect` function, which should be used to connect to a QUIC server that is listening at the given destination. This function should be called after the first initialisation function, but before `quic_client_setup_epoll`. This order is again similar to that of the server part.

As we have mentioned before, the client implementation automatically opens a unidirectional stream as soon as possible. That being said, the `quic_client_write` can be used to push arbitrary user data to said stream. Before the call to locally open the stream was made, this function simply returns -1 if it is called. Otherwise, if the stream was initiated locally, this function copies the data it receives as input into a library buffer, and sends it as soon as possible.

Finally, the `quic_client_step` function is almost identical to that in the server, and Figure 3.6 illustrates how it works as well. The client part also requires this function to be continuously called in a loop for it to be able to process expired timers and socket-pending operations. As such, the server and client parts work in a uniform manner, which makes them even easier to integrate in a wider system.

3.3.4.3 Connection and stream

Both the server and the client data structures we described above contain a pointer to an object of type `Connection`. This object is used internally to

```
typedef struct {
    SSL *ssl; /* TLS library session data */
    ngtcp2_conn *conn; /* ngtcp2 connection handle */
    ngtcp2_crypto_conn_ref conn_ref; /* ngtcp2 callback storage */
    ...
    Stream *stream; /* stores stream data that is not yet acknowledged */
} Connection;
```

Figure 3.9: Part of the structure used to abstract a QUIC connection. Fields are briefly explained in the comments next to them.

encapsulate the state of a connection. Figure 3.9 shows the important part of the actual data structure as it is implemented. The `ssl` member points to session-specific data that is used by the TLS library. The `conn` member is the ngtcp2 specific connection handle, and `conn_ref` is essentially a technicality that is used by ngtcp2 to facilitate TLS library integration. Finally, the last member of the structure as it is shown here is of `Stream` type and is used by a peer that sends data over QUIC streams to store it until it is acknowledged by the other peer.

Among the functions that deal with objects of `Connection` type is a function named `connection_start`. This function is used to initialise the TLS library-specific session context and connect said session object to ngtcp2. The initialisation of said context depends on whether or not it will be used for a server or a client, and as such this function has a parameter that indicates that. Two other important functions that deal with `Connection` are `connection_read` and `connection_write`. The first one is used only by the client to process incoming packets, but its functionality is also embedded in the server specific packet processing logic. The latter function is used by both the client and the server, and handles all socket and stream writing.

All objects of type `Connection` have a member of type `Stream`. The structure named `Stream`, used to represent a QUIC stream, and an additional helper structure, are shown in Figure 3.10. This is used to abstract the functionality needed to store data specific to a QUIC stream. As per the ngtcp2 requirements, data must be

```
typedef struct Node_ {
    struct Node_ *next; /* next Node in the list */
    size_t size; /* data size of this node */
} Node;
typedef struct {
    int64_t id; /* stream id */
    Node *head, *tail; /* head and tail list nodes */
    size_t sent_offset; /* start offset of sent data */
    size_t acked_offset; /* start offset of acknowledged data */
} Stream;
```

Figure 3.10: The structures used to abstract a QUIC stream. Fields are briefly explained in the comments next to them.

stored until it is acknowledged by the peer. As the ngtcp2 library reasons about data using offsets, information about how much of the data was sent and how much of the data was already acknowledged is also stored as offsets.

To add data that is to be sent to a `Stream` object, a push function is called with a pointer and a size. In this function, a new memory blob of size `sizeof(Node) + data_size` is allocated. The data pointed to by the pointer given as an argument to the function, which is of size `data_size`, is then copied to offset `sizeof(Node)` in this blob. Thus, a `Node` contains the two fields that are visible in the figure, and the actual data is stored contiguously in memory, right after this header.

A stream contains a list of data nodes, for which it stores the head and tail. Whenever data is pushed to the stream, a node that contains a copy of the given data is appended at the end of the list. Whenever data is pulled and marked as sent from a stream, `sent_offset` is incremented. When a callback we installed in the ngtcp2 library is called, indicating that a certain section of the data has been acknowledged by the peer, the corresponding node is found in the list and deallocated, and `acked_offset` is also incremented.

In practice, the client pushes data to a stream using the previously mentioned `quic_client_write` function. Data is then sent as soon as possible in a call to `connection_write`, and marked as sent. Whenever said data is acknowledged,

which can only happen as a result of an incoming packet that was processed by `connection_read`, data is marked as acknowledged and can finally be deallocated. The reason for which the data must remain in scope until it is acknowledged is that it could need to be resent if the initial packet that contained it was lost.

3.3.5 cFS integration

After creating two easy-to-use interfaces for the server and client parts of QUIC, all that is left is to integrate them both in cFS and in our Benchmark Tool. We modified our Benchmark Tool and added both a QUIC client and a server, as this tool needs to be able to send and receive data. We added the QUIC client to the Telemetry Output module, which is used only to send data, and we added the QUIC server module to the Command Ingest module, which only receives data. We maintained the option to use raw connections or SDLS security, and used preprocessor macros to toggle between these two modes and a special QUIC mode.

In this section we explain how we integrated the two parts in the respective cFS modules. This should be enough to fully understand how they are used and also motivate the design choices that were made when creating the server and client interfaces. That being said, usage in the context of the Benchmark Tool is very similar but less complicated.

3.3.5.1 Command Ingest modifications

We start by describing how we modified the Command Ingest module to add support for the QUIC server. As we have mentioned before, cFS applications usually have a function in which they perform initialisation. In the case of the Command Ingest module, among other initialisation it does, it also opens and binds a UDP socket.

As such, we identified the portion of logic that does this in the initialisation function and surrounded it with preprocessor if guards. We added the code snippet shown in Figure 3.11 that is executed when QUIC is desired instead of the two other modes. The `CI_LAB_Global.quic_server` variable is of type `QuicServer` and is stored in the global data portion of the CI app. We use the default address

```
quic_server_init(&CI_LAB_Global.quic_server, recv_stream_data_cb, false);
quic_server_bind(&CI_LAB_Global.quic_server, "127.0.0.1", "1234");
quic_server_setup_epoll(&CI_LAB_Global.quic_server);
```

Figure 3.11: Usage of the QUIC server initialisation functions in the context of the Command Ingest module.

```
quic_server_step(&CI_LAB_Global.quic_server, 10);
```

Figure 3.12: Usage of the QUIC server step function in the context of the Command Ingest module.

and port, which are also used in the other modes. The `recv_stream_data_cb` function is given as a parameter, indicating that that function will be called whenever application data is received.

The Command Ingest module, as other cFS apps in general, executes most of its logic in a while loop. In this while loop, it calls the `CI_LAB_ReadUpLink` function, where it processes at most 10 messages that are pending on its socket. In this 10 iteration loop, we previously added support for SDLS security. As such, messages are read from the socket, optionally decrypted, and pushed to the Software Bus to be consumed by other cFS modules.

We simply surrounded the whole logic in this function with preprocessor guards and replaced with a singular line that is executed in the QUIC case. This line is shown in Figure 3.12. As the initial for loop always did 10 iterations, we considered it appropriate to use the `quic_server_step` we previously described and also do 10 iterations.

This effectively deals with any incoming connection requests as we have previously explained, and also calls the `recv_stream_data_cb` function that we set up in the initialisation part whenever user data is received. As we wanted the flow of data to remain the same no matter what type of security method we used, we moved and adapted the logic from `CI_LAB_ReadUpLink` to the callback function. As such, the same logic that pushed the received messages to the Software Bus is also

executed in the QUIC case, but the code itself is in a different function. The `recv_stream_data_cb` function has multiple arguments that are specific to ngtcp2, but only two of them are relevant: `const uint8_t *data` and `size_t datalen`. The processed user data is thus received via a pointer and a size variable, which we then use to `memcpy` the data to a Software Bus compatible buffer.

3.3.5.2 Telemetry Output modifications

The Telemetry Output module does not immediately start forwarding telemetry to the ground station. Instead, it requires a command that configures it to start doing so. The function that is used to open a UDP socket when this command is received is called `TO_LAB_openTLM`. To enable QUIC client functionality for this module, we modified this function and added three lines that initialise the client and make it start a connection to the destination address that was just received in the command packet.

The main logic flow from the TO module also starts in a while loop, much like in the Command Ingest module. In the TO module, the `TO_LAB_forward_telemetry` is called in the main while loop, and it only exits after all available messages were read from the Software Bus pipe. In the event that telemetry forwarding was not yet enabled, meaning that the enable command was not yet received, messages are simply discarded and not forwarded.

As such, we again surrounded a portion of the while loop that processes said Software Bus messages in guards. This section optionally encrypted a message and then simply sent it via the previously opened UDP socket. We replaced this functionality with a single line, as shown in Figure 3.13. Here, the `SBBufPtr` variable points to the beginning of the current Softare Bus message, and `size` represents the size of this message. In the case that the QUIC handshake was not yet finished and as such the message cannot be queued for sending, this function returns -1. We treat this as if the enable command had not yet been received, and simply discard the current telemetry message.

```
quic_client_write(&TO_LAB_Global.quic_client, (const uint8_t *) SBBufPtr,
                  size);
```

Figure 3.13: Usage of the QUIC client write function in the context of the Telemetry Output module.

```
#ifdef QUIC
    if (TO_LAB_Global.downlink_on == true) {
        quic_client_step(&TO_LAB_Global.quic_client, 10);
    }
#endif
```

Figure 3.14: Usage of the QUIC client step function in the context of the Telemetry Output module.

Finally, at the end of this function, after all messages that were in the Software Bus pipe have been pushed to the `Stream`, we call the step function. There is no equivalent to this functionality for the case in which QUIC is not used, as messages are immediately sent to the socket in that case, and not just copied in another location. Figure 3.14 shows how this is done in the QUIC case, where the step function is called only if the downlink has been activated by the enable command. We use only 10 steps as in the Command Ingest case, so as to not block the execution flow in a loop for too long a time.

4 Analysis

In this chapter we contrast the performance of asymmetric cryptography to that of symmetric cryptography by comparing SDLS to TLS. We use a relative metric that will help us better understand how much costlier a TLS handshake is compared to symmetric cryptography. We also compare the performance of the symmetric algorithms implemented by CryptoLib for SDLS, to those used in TLS. We briefly discuss performance-security trade-offs between different TLS 1.3 ciphers and groups, and also compare OpenSSL to wolfSSL. Finally, we briefly look at how an untuned QUIC implementation handles packet loss and high latencies.

4.1 Experimental setup

The experiments were conducted on a Dell laptop with a Intel i7-8750H CPU and 16GB of DDR4 memory. The processor has 6 cores and 12 threads, and was capped to a stable 3.5GHz under load. The memory speed is 2667 megatransfers/s. As using actual satellite hardware was not a possibility, we opted to use a personal computer and focus on the relative performance of the security methods we wanted to test.

All code was compiled with GCC-11 and was run on Ubuntu 20.04. For all software components, including the TLS libraries, the `-O3` compiler flag was

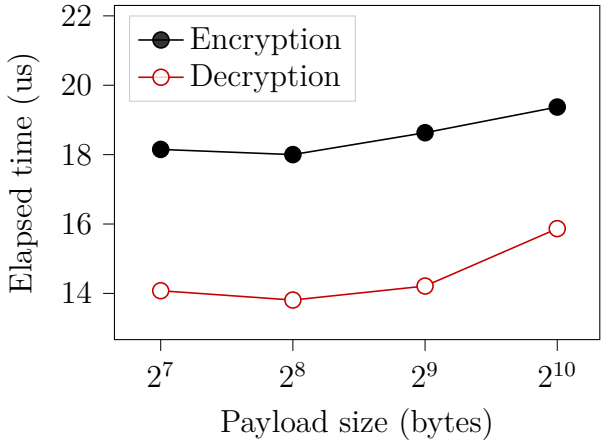

Figure 4.1: SDLS Encryption and Decryption times for varying payload sizes. Time is in microseconds.

specified to enable the most amount of performance optimisations without sacrificing arithmetic precision.

Although the cFS system creates multiple tasks, they can only use a single kernel thread. We made sure that the kernel does not switch the process to another thread by invoking it with `taskset`, thus making sure that it is pinned to a specific core.

All measurements are taken directly from the respective cFS modules. As such, the Benchmark Tool is only used to interact with these modules and acts as a ground station.

4.2 Symmetric performance comparison

Figure 4.1 illustrates the encryption and decryption (including authentication) time of payloads of different sizes using SDLS. Tests were repeated 20 times and the mean time was taken. We tested 4 different payload sizes: 128, 256, 512 and 1024 bytes. As a result of multiple limitations, including in CryptoLib and in the cFS system, we decided to not test payload sizes greater than 1024.

We can see a slight increase in compute times for the authenticated encryption operation as the payload size increases. That being said, this increase is quite subtle, with the mean time difference between encrypting a payload of size 128 and one of size 1024 being just 1.2 microseconds. Decryption times are significantly

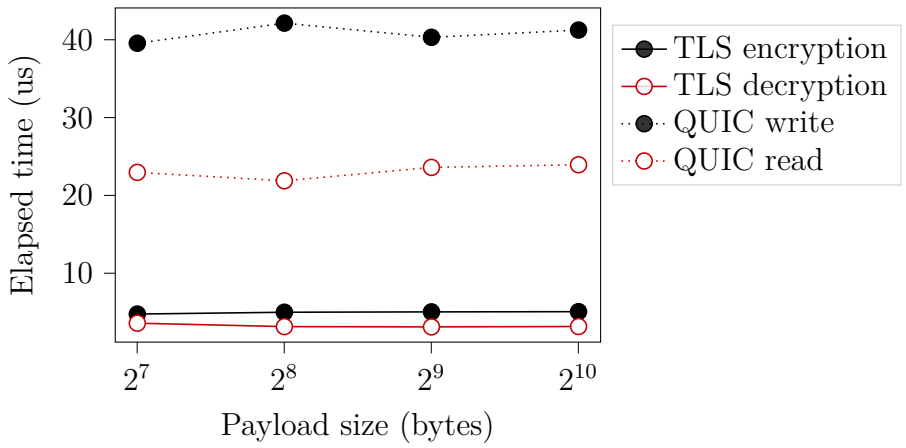

Figure 4.2: QUIC packet processing and writing times, and AEAD payload encryption and decryption times, for varying payload sizes. TLS backend is OpenSSL and cipher is AES256-GCM-SHA384. Time is in microseconds.

shorter at all sizes, although they also increase with payload size. The relatively small difference of compute time necessary to process payloads of size 128 compared to payloads of size 1024 indicates that the majority of time is not spent performing the main AES operations on data.

In Figure 4.2, we present QUIC timings for the same payload sizes. Once again, experiments were repeated 20 times and the mean time was taken. Both the times to actually encrypt and decrypt the payloads using the chosen AEAD algorithm, and the time spent writing and processing QUIC packets are shown. To record the payload encryption and decryption times, we replaced the respective ngtcp2 cryptography callbacks with versions that encapsulate the real logic in timing code. As such, the QUIC times represent the entirety of compute time spent on preparing an outgoing QUIC packet or ingesting an incoming QUIC packet, meaning that they include the TLS encryption and decryption times. Although other information besides user data can be sent in QUIC packets (such as acknowledge frames), these times are representative of packets that contain user payloads of the specified size.

It is immediately evident that the time spent performing AEAD operations represents a small percentage of the total compute time. Although the time spent in QUIC operations does appear to slightly increase with payload size, the variations are too small to to accurately be discerned from measurement noise. Another

Table 4.1: QUIC read times for different TLS ciphers and backends. Times are in microseconds and there are four payload sizes shown.

Cipher	Backend	128B	256B	512B	1024B
AES-128-GCM-SHA256	OpenSSL	25	23.7	24.5	24.4
AES-128-GCM-SHA256	wolfSSL	24.5	23.4	25.5	25.8
AES-256-GCM-SHA368	OpenSSL	23	21.8	23.6	24
AES-256-GCM-SHA368	wolfSSL	24.4	25.8	24.1	26
CHACHA20-POLY1305-SHA256	OpenSSL	23	22.8	22.7	22.4

Table 4.2: QUIC write times for different TLS ciphers and backends. Times are in microseconds and there are four payload sizes shown.

Cipher	Backend	128B	256B	512B	1024B
AES-128-GCM-SHA256	OpenSSL	42.5	41	42	41.4
AES-128-GCM-SHA256	wolfSSL	41.1	44.3	43.5	43.5
AES-256-GCM-SHA368	OpenSSL	40	42.1	40.3	41.3
AES-256-GCM-SHA368	wolfSSL	41.2	40.6	42.4	43.5
CHACHA20-POLY1305-SHA256	OpenSSL	41	37	40	41

obvious conclusion is that the encryption operation takes longer than decryption, and that writing a QUIC packet takes significantly more time than ingesting one. These observations match what we have seen in the SDLS case.

Considering the 1024 byte payload, the total time spent on applying SDLS security was around 19 microseconds. For the same size, our QUIC implementation with OpenSSL and the same AES-256-GCM cipher spent about 41 microseconds applying TLS security and performing protocol specific tasks. In the case of packet ingest, just under 16 microseconds were spent processing SDLS security, and under 24 microseconds ingesting the QUIC packet. This means that once a session is established, compared to SDLS, only 1.5x more time is spent on processing incoming QUIC packets, and about 2.15x more time is spent creating outgoing packets.

4.2.1 Testing different ciphers and TLS backends

To quantify the performance difference between the less secure and more secure ciphers available in TLS 1.3, we decided to test three of them. Moreover, we tested both the OpenSSL and the wolfSSL implementation of these ciphers to discover if there is any significant speed difference.

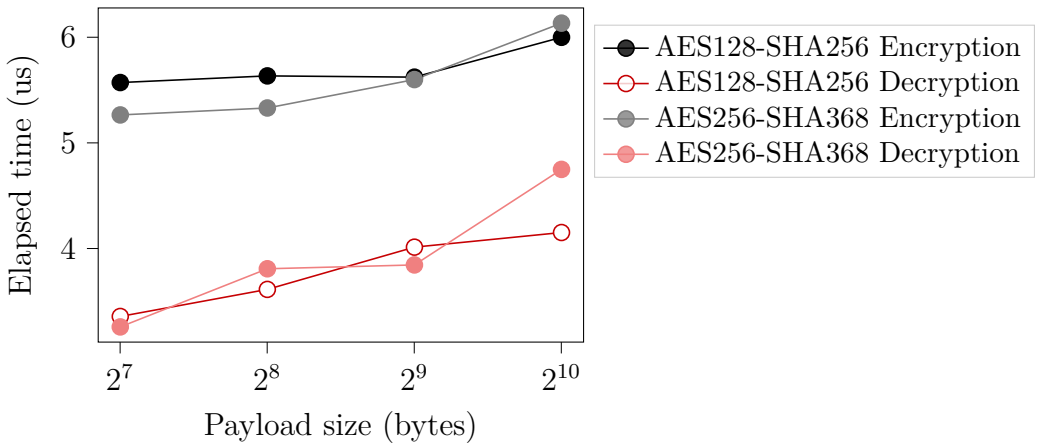

Figure 4.3: The payload encryption and decryption times in microseconds of two different ciphers with wolfSSL as the TLS backend.

Table 4.1 and Table 4.2 show the times we obtained for the complete QUIC reading and writing operations, which include the cryptographic parts. We tested the four sizes about which we previously discussed in this chapter, and both TLS backends where possible. In the case of the CHACHA20-POLY1305-SHA256 cipher, we only tested OpenSSL, as the wolfSSL development version we used did not behave correctly with this cipher.

Although we collected 20 different times and took the mean, the findings are still quite noisy. As such, it is not clear if wolfSSL is faster than OpenSSL or the other way around. Moreover, it does not seem that the cipher used has any significant impact on the overall QUIC ingest or output performance with these relatively small packet sizes.

When looking only at encryption and decryption timings, which exclude the QUIC logic, it is once again unclear which one of the two TLS libraries is faster. Figure 4.3 shows the times obtained by the two AES variants with wolfSSL as a backend. Although it is easy to see that times increase with payload size, it remains uncertain whether or not one of the two ciphers we tested is faster than the other.

4.3 Handshake performance

As previously mentioned, the handshake part of the TLS protocol is the most computationally costly operation performed in the lifetime of a session. We measured

Table 4.3: QUIC handshake times from a client and server perspective. Times are in microseconds.

Curve	Client	Server	Total
Curve25519	463	732	1195
P-256	512	779	1219
P-384	2204	2526	4730
P-512	4550	4794	9344

the total compute time spent on the handshake part of the QUIC protocol from both a client and a server perspective. These measurements include the key exchange operations performed as part of the TLS 1.3 protocol.

Table 4.3 shows the client, server and total compute time spent performing the QUIC handshake. We tested the four curves supported in the TLS 1.3 standard. All the curves are used to perform the same Elliptic-curve Diffie–Hellman protocol. We used OpenSSL as a TLS backend, as we expect wolfSSL performance to be similar, as it was in the symmetric case. We performed each measurement 10 times and took the mean.

Using Curve25519, which offers 128 bits of security with 256 bit keys, resulted in the fastest handshakes, with the P-256 configurations being the second fastest. The P-384 and P-512 configurations were much slower but they offer more bits of security.

Looking again at Curve25519, which is also the default choice in the OpenSSL implementation we used, the compute time spent from a client perspective is equivalent to sending about 12 messages with payloads of size 1024. In the same case, the compute time required by a QUIC server to perform a full handshake is similar to that required to send 18 messages over a previously established TLS session.

4.4 Memory usage

We measured the memory usage of the entire cFS process in four different configurations in order to be able to assess how much memory overhead SDLS and QUIC produce over the baseline version without any security. We used Heaptrack [34] to record the peak heap and the resident set size (RSS) memory usage of

Table 4.4: Memory usage of the whole cFS system in different security configurations, as recorded by Heaptrack.

Configuration	Peak heap usage	Peak RSS usage
No security	84.5 KB	6.7 MB
SDLS security	89.5 KB	8.3 MB
QUIC with OpenSSL	583.3 KB	13.3 MB
QUIC with wolfSSL	344.8 KB	9.5 MB

the process. RSS represents the amount of memory a process that is in RAM is currently using. Although the heap memory usage metric is exact, the RSS usage reported by Heaptrack includes an overhead resulting from the measurement tool.

Table 4.4 presents the data we collected. The absolute value in the RSS column might not be relevant, but we can look at the relative increase over the no security configuration to get a better idea of the memory cost of SDLS and QUIC. As such, the libraries used by SDLS only use 5KB more memory on the heap, but consume 1.6 more megabytes when looking at RSS.

The QUIC configuration that uses OpenSSL as a TLS backend uses the most memory by far, with about 500KB more heap memory usage and 6.6MB more total usage when compared to the baseline. Although not tuned for low memory usage, the wolfSSL library still uses significantly less memory than OpenSSL, with a total memory usage only 1.2MB more than in the SDLS configuration. Heap memory usage is also much lower with wolfSSL than with OpenSSL, and can be avoided altogether by configuring the library to only use static buffers.

4.5 Behaviour under packet loss and high latency

We used the NetEm [35] Linux tool in order to introduce artificial randomised latency and packet loss on a local network interface. We configured the packet loss at 15% and latency at 200ms ± 25ms. Although packet loss this high is unusual even in satellite networks, it resulted in a good stress test for our system.

We recorded all UDP traffic using WireShark [36]. Although WireShark is not able to inspect the contents of QUIC packets by default, we also configured our QUIC

Table 4.5: Part of a QUIC conversation in high latency and packet loss conditions. Client is cFS and server is the Benchmark Tool. Time is seconds elapsed since first entry.

Time	Sender	Pkt. Num.	Packet Content
0	client	47	Stream data offset 5328
0.21	server	37	ACK
0.48	client	49	Stream data offset 5184
0.49	client	48	Stream data offset 5472
0.68	server	38	ACK
0.71	server	39	ACK
1.52	client	50	Stream data offset 5616
1.71	server	40	ACK + PING
2.00	client	51	ACK
2.50	client	52	Stream data offset 5760

server to dump TLS session secrets to a file, which we loaded in WireShark. As such, we were able to inspect the packets in detail and see exactly what data they contained.

After configuring our test system to continuously send small payloads to the external Benchmark Tool, we inspected the packet conversation and found an instance which shows how QUIC handles packet loss and packet inversion. We show a part of this conversation in Table 4.5. The data in the Packet Number column is taken directly from the QUIC frame as decoded by WireShark. The packets we show contained stream data, acknowledge frames and a ping frame. As mentioned in a previous section, the QUIC protocol reasons about the data sent in a stream using offsets. As such, the first packet from the table has offset 5328. The packets used in this test have a total size of 144 bytes, making this the $5328/144 + 1 = 38$th payload from this particular session.

There are two interesting behaviours of the protocol that are present in this data. The first one is a packet inversion that can be seen by looking at the 3rd and 4th entry of the table. Both these entries represent packets going from the client to the server, and both of them contain stream data. By looking at the Packet Number data, it becomes clear that these two payloads were delivered out of order: the 3rd packet has a QUIC sequence number of 49, while the 4th packet has a sequence

number of 48. Although these packets were delivered out of order by the network stack, the QUIC protocol was designed to handle this situation without errors.

The more relevant but also more subtle behaviour that can be observed in this data is packet loss. More specifically, the first packet in the table, with sequence number 47, contains 144 bytes of stream data starting from offset 5328. The server receives this packet and sends an ACK which is received after 210ms. This ACK packet indicates to the client that the server does not yet have the payload identified by offset 5184. As such, the client resends this payload in the 3rd packet of this table, with sequence number 49. By reordering the packets sent by the client (by the Packet Number), we get the sequence of offsets 5328, 5472, 5184 and 5616. Clearly, the payload identified by offset 5184 was thus resent because it was lost.

4.6 Overview

The experiments we carried out help us compare the computational impact of the TLS protocol relative to the SDLS protocol. As such, for symmetric encryption, SDLS takes about 20 microseconds to process a 1024 byte payload, and performing the required QUIC logic takes around 40 microseconds. For symmetric decryption, SDLS needs around 15 microseconds while QUIC can process an incoming packet in around 22 microseconds.

Although the ciphers we tested all performed similarly, using Curve25519 as part of the key exchange algorithm was the fastest. More specifically, we found that the QUIC client was able to perform the handshake using this curve in less than 500 microseconds, while a server needed around 750 microseconds. Expressing these times in relative terms based on the SDLS timings we gathered, a client handshake is equivalent to sending about 25 SDLS secured 1024-byte payloads. Using the same metric, a server handshake is equivalent to sending around 37 messages using SDLS.

While we showed that QUIC performs satisfactorily under space link conditions, the SDLS protocol does not have the same inherent property when it comes to packet loss, and its symmetric cryptographic performance is not significantly faster than that of QUIC.

5
Conclusion

As more and more independent satellites are launched into orbit, the need for a well adopted space security standard that uses the latest available cryptographic techniques becomes evident. Starting from the premise that hardware that is sent into space has progressed enough to be able to handle the notoriously costly TLS handshake, but also taking into account the fact that space links might be subject to high latency and packet loss, we proposed the use of the QUIC protocol to secure space links.

To demonstrate the viability of using this protocol in a realistic flight scenario, we integrated it into the cFS software stack that is currently being used in space missions deployed by NASA. We compared the compute time required by various aspects of the QUIC protocol to the widely used SDLS protocol in order to further asses the viability of using this protocol on typically compute-constrained space hardware. We tested an embedded TLS library named wolfSSL and showed that is consumed significantly less memory while preserving compute performance when compared to the widely used OpenSSL library. Finally, we tested an untuned version of the QUIC protocol under very harsh conditions that included significant packet loss and high latency, and found that it handled these conditions well.

We believe that the results we showed in the previous chapter are indicative of the fact that TLS is viable and can be used successfully in a space context.

Although indeed much costlier than simply encrypting payloads using AES, the TLS handshake compute time was shown to be equivalent to sending 12 messages containing 1024 byte payloads from a client perspective. As such, at least under the assumption that once established, TLS sessions are used for reasonable amounts of time, the cost of establishing the session itself becomes less significant.

Moreover, using the QUIC protocol with an untuned version of the wolfSSL embedded TLS library only results in 1.2MB more RSS usage than using SDLS provided security. Not only does QUIC bring the advantage of forward secrecy through session specific keys, but it also offers a way to handle unreliable space connections at the application layer.

Both the QUIC protocol and the embedded TLS library we used have not been tuned in any way to behave more appropriately for this context. We believe there is significant potential in adjusting the QUIC protocol parameters in order to minimise packet size and optimise the behaviour of the protocol for the specific space link it is ran on. As mentioned, QUIC supports multiple congestion control algorithms, some more suitable for space links than others [21]. The fact that QUIC handles packet loss at the application layer means that it is also easy to adjust it after deployment, perhaps even to generate multiple configurations depending on the satellite position in orbit or other conditions. As for the wolfSSL library, we believe that memory usage can be decreased by restricting the available ciphers and groups using compilation flags. Moreover, the library can also be configured to use different versions of the necessary cryptographic algorithms that use less memory at a small performance penalty.

Although more optimisations can be performed, we believe that the QUIC protocol would be a good fit for securing space communications, as it provides asymmetric security using the latest TLS 1.3 standard while not consuming exaggerated amounts of resources and handling harsh network conditions well.

Bibliography

[1] *HTTPS encryption on the web*. https://transparencyreport.google.com/https/overview. Accessed: 01-08-2022.

[2] M. Bishop. *HTTP/3*. RFC 9114. RFC Editor, 2022.

[3] *Satellite traffic study*. https://www.nsr.com/?research=satellite-traffic-study-how-much-data-over-satellite. Accessed: 01-08-2022.

[4] *Startlink home page*. https://startlink.com. Accessed: 01-08-2022.

[5] Lun Li et al. "Secure spectrum-efficient frequency hopping for return link of protected tactical satellite communications". In: *2016 IEEE Military Communications Conference, MILCOM 2016, Baltimore, MD, USA, November 1-3, 2016*. Ed. by Jerry Brand et al. IEEE, 2016, pp. 254–258. URL: http://dx.doi.org/10.1109/MILCOM.2016.7795335.

[6] Wei Li. "Security Analysis of DVB Common Scrambling Algorithm". In: *The First International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007)* (2007), pp. 271–273.

[7] *CCDS. Space Data Link Security Protocol - Recommended Standard*. Green Book. 2015.

[8] *Falcon 9 Hardware discussion*. https://space.stackexchange.com/questions/9243/what-computer-and-software-is-used-by-the-falcon-9. Accessed: 03-08-2022.

[9] *core Flight System - Presentation*. https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf. Accessed: 01-08-2022.

[10] *core Flight System - Introduction*. https://cfs.gsfc.nasa.gov/Introduction.html. Accessed: 03-08-2022.

[11] David McComas, Jonathan Wilmot, and Alan Cudmore. *The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft*. 2016.

[12] *cFS GitHub repository*. https://github.com/nasa/cFS. Accessed: 03-08-2022.

[13] *freeRTOS - Website*. https://www.freertos.org/. Accessed: 25-08-2022.

[14] *CCSDS - Website*. https://public.ccsds.org/. Accessed: 03-08-2022.

[15] *Space Data Link Protocols - Summary of Concept and Rationale*. Green Book. 2015.

[16] *Space Data Link Security Protocol—Summary of Concept and Rationale*. Green Book. 2018.

[17] *Keyless TLS*. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details. Accessed: 06-08-2022.

[18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, 2018.

[19] *A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)*. https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/. Accessed: 06-08-2022.

[20] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, 2021.

[21] Aitor Martin and Naeem Khademi. "On the Suitability of BBR Congestion Control for QUIC over GEO SATCOM Networks". In: ANRW '22. Philadelphia, Pennsylvania: Association for Computing Machinery, 2022. URL: https://doi.org/10.1145/3547115.3547194.

[22] *cFS Telemetry Output Module - GitHub*. https://github.com/nasa/CFS_TO. Accessed: 12-08-2022.

[23] *cFS Command Ingest Module - GitHub*. https://github.com/nasa/CFS_CI. Accessed: 12-08-2022.

[24] *cFS Cryptography Library (CryptoLib)*. https://techport.nasa.gov/view/96783. Accessed: 06-08-2022.

[25] *CryptoLib - GitHub*. https://github.com/nasa/CryptoLib. Accessed: 06-08-2022.

[26] *AOS Space Data Link Protocol*. Blue Book. 2021.

[27] *libgcrypt - GnuPG*. https://www.gnupg.org/software/libgcrypt/index.html. Accessed: 06-08-2022.

[28] *Space Data Link Security Protocol—Extended Procedures*. Blue Book. 2020.

[29] *ngtcp2 - GitHub*. https://github.com/ngtcp2/ngtcp2. Accessed: 07-08-2022.

[30] *OpenSSL - Main page*. https://www.openssl.org/. Accessed: 07-08-2022.

[31] *wolfSSL - Main page*. https://www.wolfssl.com/. Accessed: 10-08-2022.

[32] *OpenSSL 1.1.1 Quic - GitHub*. https://github.com/quictls/openssl/tree/OpenSSL_1_1_1q+quic. Accessed: 10-08-2022.

[33] *Wolfssl QUIC pull request - GitHub*. https://github.com/wolfSSL/wolfssl/pull/5384. Accessed: 22-08-2022.

[34] *Heaptrack - GitHub*. https://github.com/KDE/heaptrack. Accessed: 27-08-2022.

[35] *NetEm - MAN page*. https://man7.org/linux/man-pages/man8/tc-netem.8.html. Accessed: 27-08-2022.

[36] *WireShark - Website*. https://www.wireshark.org/. Accessed: 27-08-2022.