UDC 004.057.4

K. Lesohorskyi, E. Zharikov

A TRANSPORT-INDEPENDENT GENERAL PURPOSE CRYPTOGRAPHIC PROTOCOL

Abstract: The article considers the problem of the transport-independent cryptographic protocol. We analyze requirements for general purpose cryptographic protocol. An analysis of existing protocols is performed to highlight their drawbacks. A protocol schema and implementation architecture are proposed, based on the ideas of perfect forward secrecy, minimal overhead, and backward compatibility.

Keywords: Network protocols, asymmetric cryptography, key exchange algorithms, protocol buffers.

Introduction

In the modern era, most network communications are protected by encryption protocols. Most of these protocols were around since the 80s and are still meeting their functional requirements today.

However, in recent years the threat of quantum computing is constantly rising. Some quantum algorithms can solve math problems that a lot of asymmetric encryption schemes rely on in less than polynomial time. This renders a wide range of modern asymmetric cryptographic schemes obsolete. New encryption schemes are developed that don’t have such vulnerabilities and are slowly integrated into existing protocols.

However, existing protocols still have certain issues coming from backward compatibility necessities, the design philosophy of the time, or underlying security primitives.

Another sharp change in recent years is a shift towards wider adoption of UDP as a transport layer protocol. While still performing its core functions well, TCP is morally obsolete and developments in hardware and network failure rates enable faster communication using UDP. This change is well-manifested by switching to the UDP-based QUICK protocol from TCP in HTTP 3.0 [1] specification.

With the necessity to update underlying cryptographic models and transport protocols, it might be a good time to introduce new transport-independent cryptographic protocols to secure communications in the post-quantum era.

Research problem

The goal of this article is to propose a scheme and architecture for a general purpose transport-independent protocol, that would enable a secure connection between two parties. The protocol should provide perfect forward secrecy, authenticity, and integrity of underlying

communications. The protocol should be able to easily evolve and have minimal possible overhead.

Protocol flow

A protocol flow would consist of two distinct steps: handshake and data transfer. During the handshake, connection details are negotiated, a shared secret key is received, and the server is authenticated. During the data transfer step, a shared secret key is used by symmetric encryption algorithms to encrypt the data on the client side and decrypt the data on the server side and vice versa. General protocol flow can be seen in Figure 1.

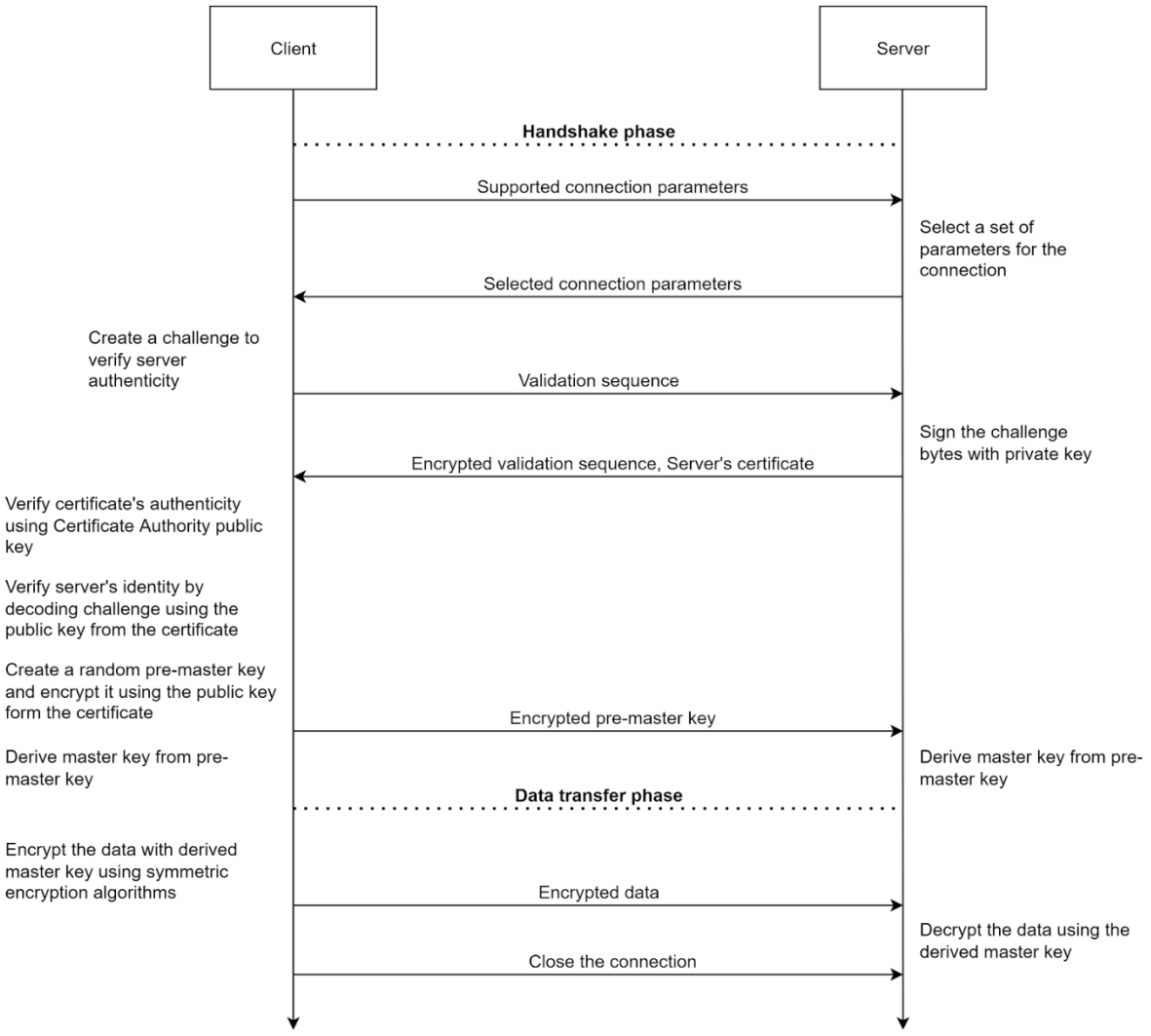

Figure 1. General protocol flow

One of the apparent problems is the number of round trips that are required during the handshake phase. The naive implementation can set up the secure connection in 3 roundtrips, one for each handshake step. However, this is a major overhead and could be reduced. One way to do this is by bundling together the parameter negotiation and authentication phase. This way handshake is implemented in the TLS 1.2 [2], which can be seen in Figure 2.

 ISSN 1560-8956

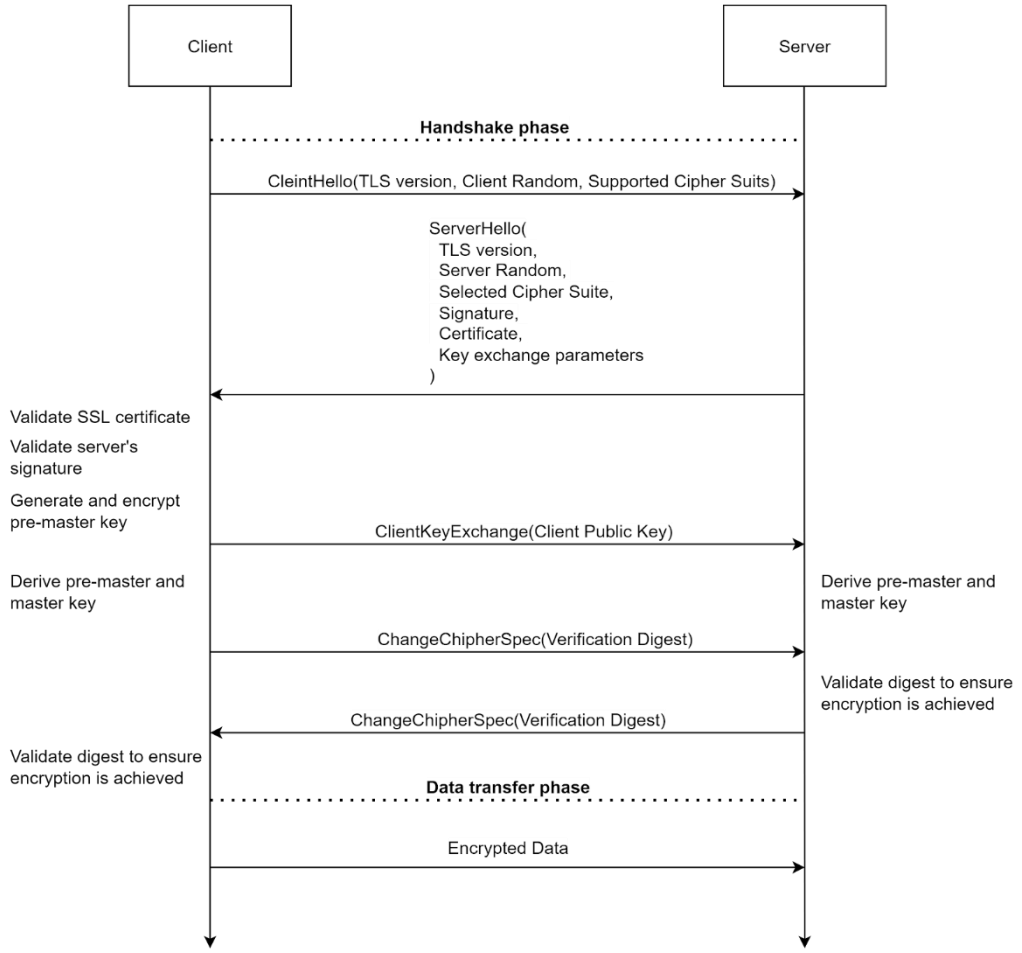

Figure 2. TLS 1.2 protocol schema [2]

However, a handshake can be even faster. To achieve this, ephemeral key-sharing algorithms should be used. With TLS 1.2, the pre-master key is encrypted with the server's public key which cannot be retrieved in one round trip. However, this can be avoided if the client and server share public keys and derive shared secret keys independently. It is only possible using dedicated key exchange algorithms, such as the Diffie-Hellman algorithm.

However, in this case, the same algorithm cannot be used for both verifying authenticity and sharing a key. Thus, a separate asymmetric algorithm is used to authenticate the server. Such a schema is used by the TLS 1.3 [3] which greatly increased performance [4].

Another benefit of using an ephemeral key exchange schema is that it provides perfect forward secrecy to the underlying communications. Classical asymmetric schemas rely on the security of the private key. If a private key is compromised, all past and future messages that are encrypted with it can be decrypted by an adversary. Ephemeral key exchange schemas create session keys instead. Even if a private key for one session is compromised, all past and future sessions will stay secure.

This also requires negotiating the parameters for the key exchange algorithm ahead of time, which limits the negotiation phase and cipher suite selection.

The proposed protocol handshake schema

Cipher suite is a set of ciphers that will be used to set up encrypted communication.

Since our protocol goal is to reduce the number of round trips, we are forced to use an ephemeral key exchange, thus cipher suite will consist of 3 algorithms:

- key exchange algorithm - derive a pre-master secret key from pre-shared parameters and other party’s public key;
- authentication algorithm - authenticate the identity of the server;
- symmetric encryption algorithm - encrypt the data for transfer.

A key exchange algorithm, its shared parameter set, and authentication algorithm must be negotiated ahead of time since the client must send its public key and a a special call (a sequence of bytes to be encrypted in the server’s response) in the initial request. This limitation does not apply to the symmetric encryption algorithm, however, as it can be selected by the server.

To tackle this, we will split the cipher set into two components: asymmetric and symmetric parameters. Asymmetric parameters are defined by the client and the server can only accept or reject them. Asymmetric parameters are parameters for the key exchange and authentication algorithm. We will refer to them as “prime parameters” for brevity. Symmetric parameters are proposed by the client and selected by the server.

If the server does not support specified prime parameters, it responds with the error packet and specifies a supported prime parameter list. Clients can then either try to connect with the new parameters or downgrade to an insecure connection if the security level is considered too low.

This approach would enable fast flow for the majority of cases when the client and server are up to date and support the most up-to-date cipher suites and would allow a gradual decrease in security in case one of the parties is out-of-date. The flow diagram of such a handshake can be seen in Figure 3.

As we can see, the fast flow handshake is completed in one round trip. If the fast flow was not possible, fallback flow could be completed in one additional round trip. One thing to note is that fallback flow can enable the usage of asymmetric encryption key exchange schemas since the server returns the certificate to the client even if the initial handshake request was not fulfilled. This would enable a quick fallback to other algorithms in case a vulnerability is found in ephemeral key exchange algorithms.

Unlike TLS which proposes a wide range of supported algorithms out-of-the-box, we would propose using a limited range of algorithms. While it limits the fallback options in case one of the algorithms is compromised, it increases the interoperability between different client implementations and decreases the server’s certificate size, which is especially critical for some post-quantum asymmetric algorithms as their key size can reach megabytes [5].

 ISSN 1560-8956

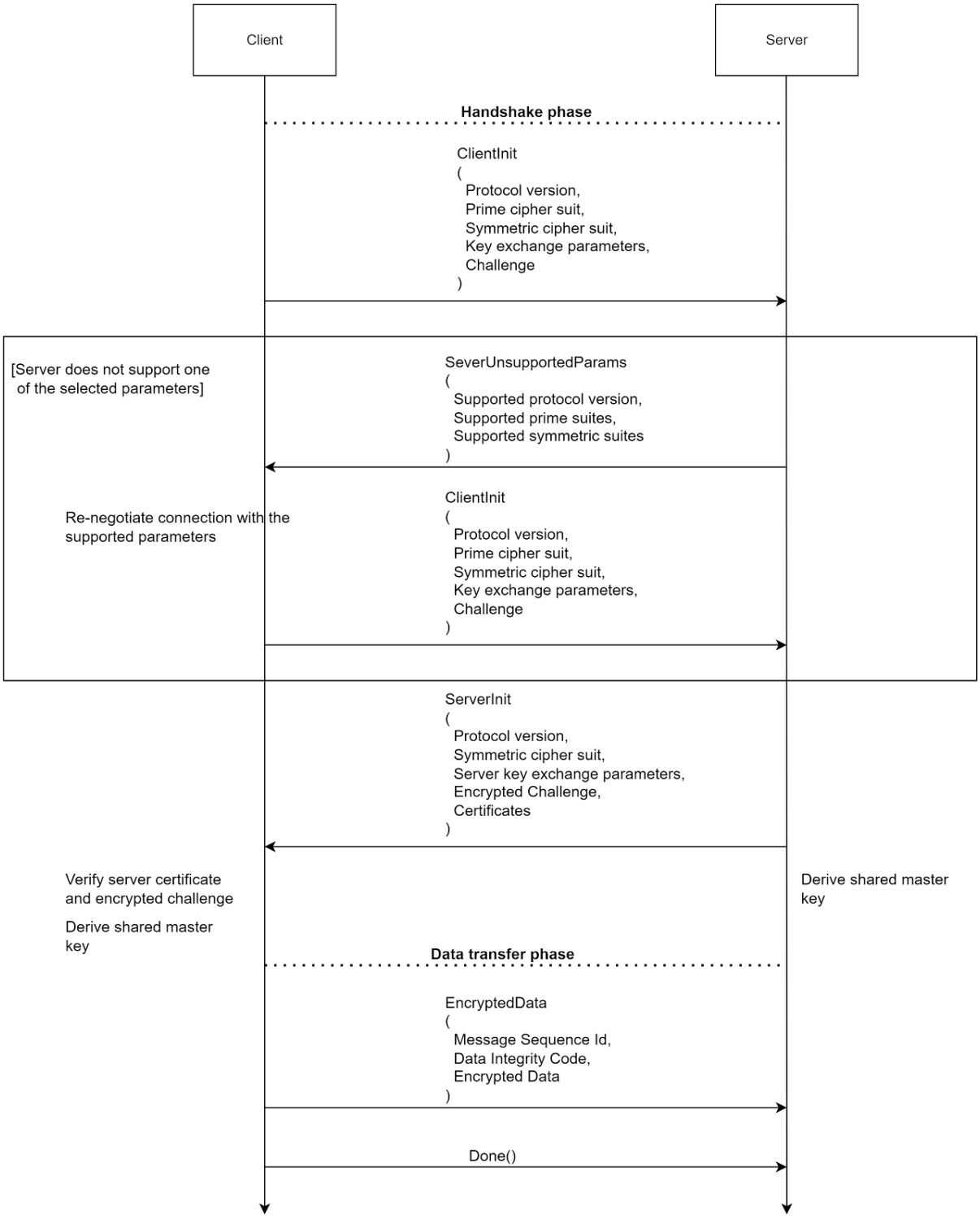

Figure 3. The proposed protocol schema

Another thing to note is that in the current protocol schema we do not consider the key renegotiation. This approach is suitable for short-lived communication protocols, such as HTTP, where only one message will be encrypted and passed in a short period of time. However, for long-living connections, it is important to be able to renegotiate symmetric

encryption keys as time passes by. This can be done either by using a master key as a seed for a random number generator which will be used to generate session keys, which then can be used for a certain number of encryptions and deterministically re-generated by a random number generator. This approach has little overhead and does not impose any extra complexity on the protocol structure, however, the underlying transport protocol must guarantee the message delivery to keep the message count in sync. If the underlying protocol does not guarantee message delivery, however, either extra messages for key renegotiation must be added or a sequence number appended to all data messages transferred by the protocol.

Message serialization

While not as complex as application-level protocols, presentation-level protocols tend to have some complexity. Another noticeable detail is that such protocols tend to also be used in proxies and embedded devices, so updating them is an extremely cumbersome task. This enforces strict requirements on backward compatibility. For example, while migrating from TLS 1.2 to TLS 1.3, many proxies would refuse to work with the new protocol version [6]. Thus, to enable new features an extension mechanism had to be implemented, which further complicates the protocol. Another problem is that each implementation of the protocol has to rely on the custom deserializer, which can introduce additional bugs and vulnerabilities.

Thus, leveraging serialization protocol to define the message template would enable easier serialization and deserialization, less room for error in each implementation, and faster serialization and deserialization.

Several popular open source serialization protocols exist nowadays, but the most prominent among them are Protocol Buffers, Fast Buffers, and Apache Thrift. Their comparison can be seen in Table 1. The performance measurement details are introduced in [7].

Table 1.

Serialization protocol comparison

Characteristic	Protocol Buffers	Fast Buffers	Apache Thrift
Licence	BSD	GNU V3	Apache V2
Deserialization, 200 bytes	< 1 microsecond	<1 microsecond	~1 microsecond
Deserialization, 1000 bytes	~3 microseconds	~1 microsecond	~8 microseconds
IDL flexibility	High	Low	Moderate
Language support	C, C++, Rust, Go, Java, .NET, PHP, JavaScript, etc.	C++	C, C++, Rust, Go, Java, .NET, PHP, JavaScript, etc.

 ISSN 1560-8956

As we can see, Apache Thrift is less suitable for over-the-wire due to its poor marshaling performance that comes from several nested non-inlinable calls in the implementation. Apache Thrift displays worse asymptotic growth in comparison to Protocol Buffers and Fast Buffers, which will get worse when the expected message size goes up. The expected payload size is around 2 kilobytes due to the bigger public keys used by post-quantum encryption algorithms.

Protocol Buffers and Fast Buffers offer low serialized payload size, high serialization and deserialization speed, and open source license. Fast Buffers offer higher serialization and deserialization speed, however, they lack flexible schema definition and wide programming language support which is provided by Protocol Buffers.

While C++ is interoperable with most existing languages and can be linked as a dynamic library, having an opportunity to implement the protocol in a wider range of languages natively brings the benefit of easier dependency management and less environment overhead. Another useful feature of Protocol Buffers is a *oneof* keyword, which allows messages to imitate discriminated unions, which would be a useful feature since the handshake step can respond with two message types - successful negotiation or unsupported “prime” cipher suite.

The proposed protocol architecture

Based on the protocol schema and requirements we discussed above we now propose an architecture for the implementation of the protocol. The architecture we propose is aimed at object-oriented languages, however functional implementations are possible too, although they require complex state management. Also, it’s worth noting that we propose a blocking version of the implementation, as its implementation would be more consistent across a wide range of languages.

A simplified UML diagram of the proposed architecture is presented in Figure 4. In the UML diagram, we skip generated Protobuff models and some helper classes and mostly concentrate on public API. We also violate some best practices for the sake of simplicity. For example, the serialization provider can be split into several subinterfaces as per the interface segregation principle [8], which would improve the testability of the implementation, but would make the UML diagram extremely bloated. However, both client and server connections will need access to all of the serialization capabilities, so we suggest using a facade interface [9] that would delegate execution to a subset of specialized interfaces.

The architecture is mostly stateless, so we heavily leverage the composition of the components while maintaining thread safety. One exceptional case is EncryptedConnection class, which uses a decorator pattern to wrap the provided transport layer connection. The transport layer connection lifetime is not bound to the resulting encrypted connection, so the caller must be careful in order not to transfer data while the encrypted connection is taking

ownership over the channel. This is a deliberate design decision so that the underlying connection can be re-used for further secured connection over and over again, thus decreasing the overhead required for consecutive connections.

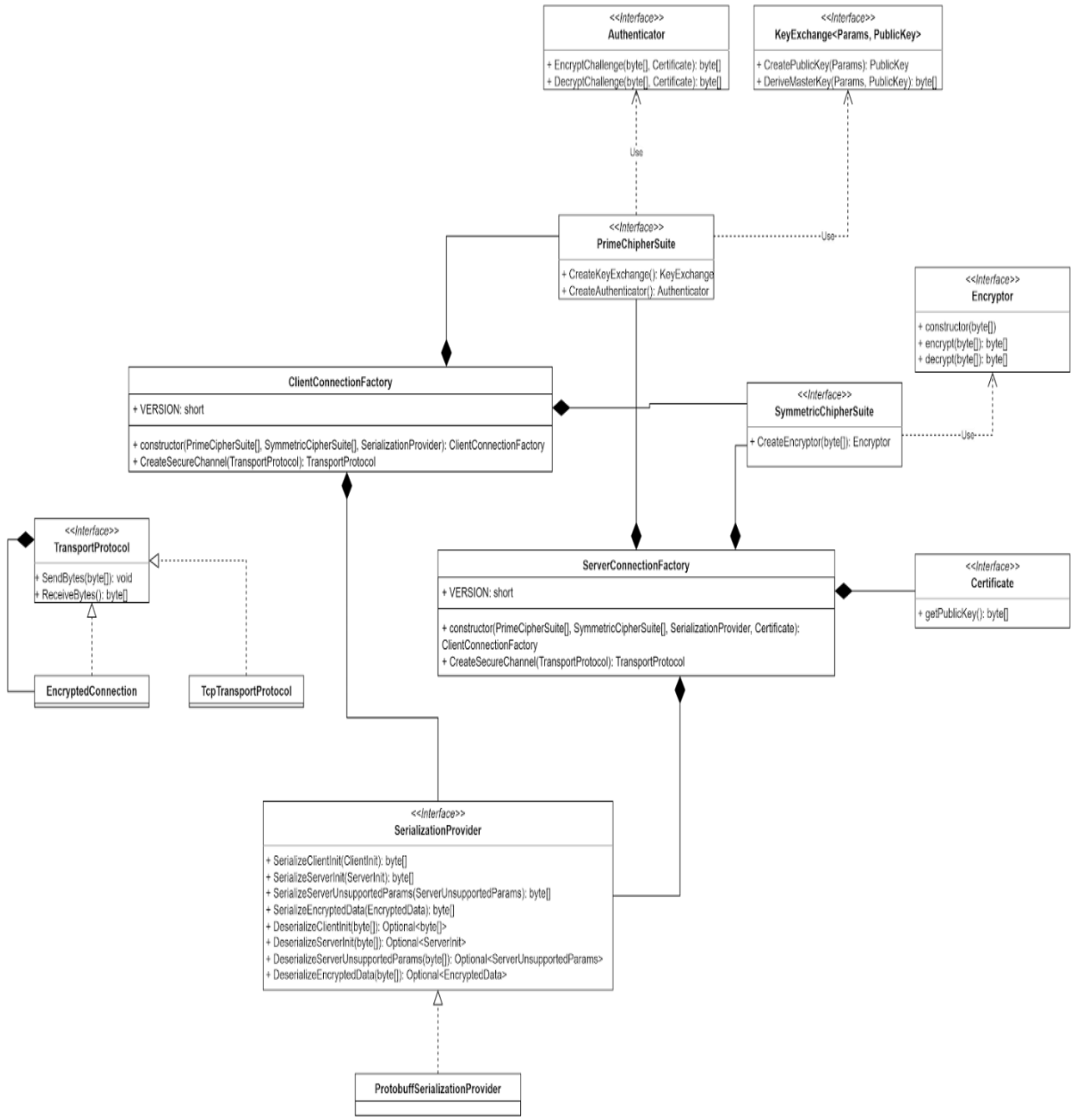

Figure 4. A UML diagram of the proposed architecture

The ClientConnectionFactory and ServerConnectionFactory are responsible for orchestrating the handshake and instantiating an EncryptedConnection that can either encrypt or decrypt data passed to it before sending it via the wrapped transport layer connection. SymmetricChipherSuite represents a cipher suite that can be used during the data transfer phase and is essentially a factory for the symmetric encryption algorithm.

Overall, this provides a flexible framework that is easy to use. Flow charts for blocking client-side and server-side communication are shown in Figure 5.

 ISSN 1560-8956

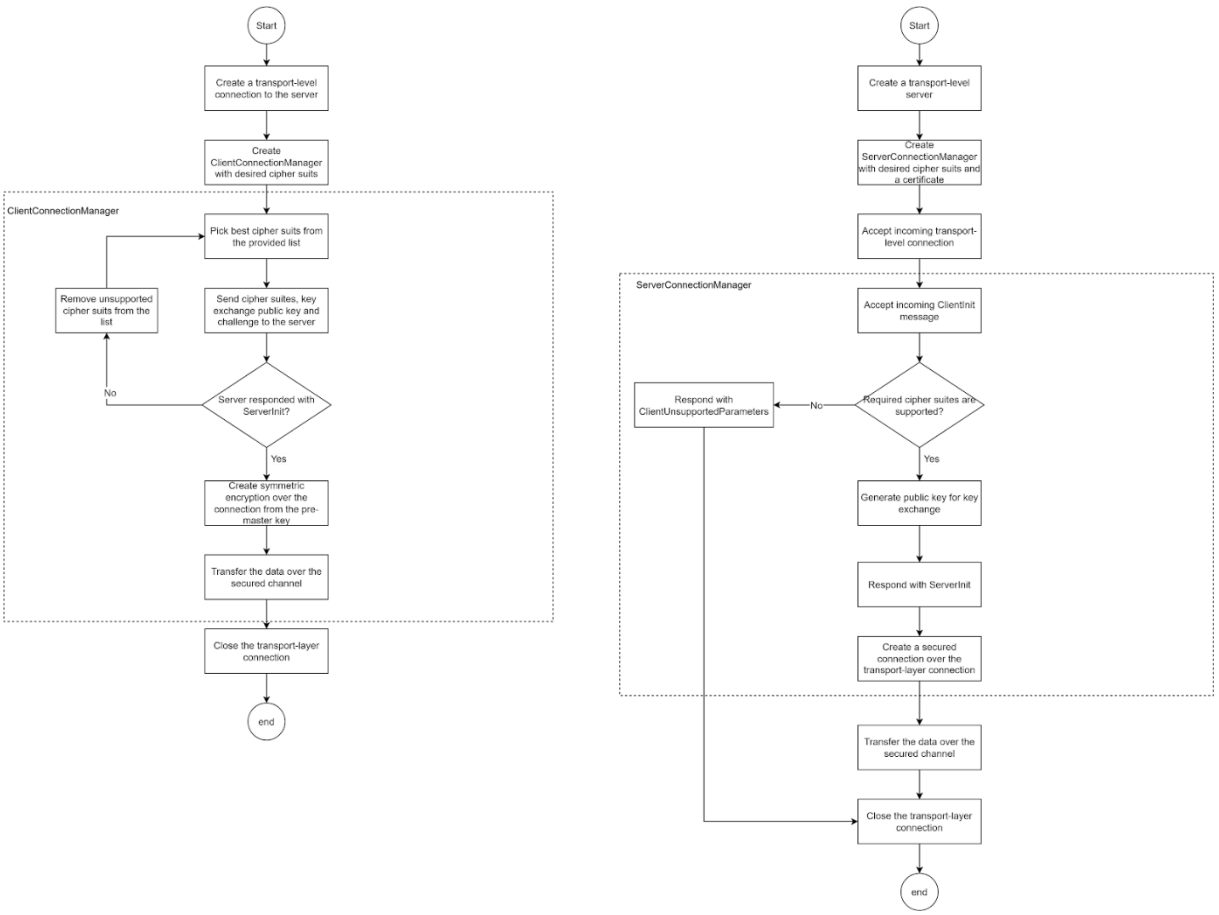

Figure 5. – Flow chart for client data transfer

Conclusion

Existing cryptographic protocols are slowly migrated toward post-quantum encryption schemes, but still have problems, such as backward compatibility and dependency on lower-level transport protocols, often having two different specifications for different transport-level protocols.

In this article, we summarized the requirements for general-purpose encryption protocols, analyzed the most efficient handshake schema, researched the potential use of existing serialization protocols to improve backward compatibility and decrease the complexity of the implementation, and proposed an architecture for the protocol that would meet all of the requirements and enable easy evolution over time while providing easy-to-use API.

As a future work, improving symmetric encryption security by adopting symmetrical key change capability can be considered. Another area of improvement could be improving composability of prime chipher suites by splitting it into independent key exchange algorithm and authentication algorithm.

REFERENCES

1. HTTP 3.0 draft proposal URL: https://datatracker.ietf.org/doc/html/draft-ietf -quic-http-34 (date of use: 11.05.2022)

2. TLS 1.2 RFC, section 7.4 handshake URL: https://datatracker.ietf.org/doc/html/rfc5246#section-7.4 (date of use: 11.05.2022)

3. TLS 1.3 RFC, section 4 handshake URL: https://datatracker.ietf.org/ doc/html/rfc8446#section-4 (date of use: 11.05.2022)

4. TLS 1.3 Performance Analysis - Full Handshake URL: https://www.wolfssl.com/tls-1-3 -performance -part -2-full-handshake-2/ (date of use: 11.05.2022)

5. Initial recommendations of long-term secure post-quantum systems, section 4 URL: http://pqcrypto.eu.org/docs/initial-recommendations.pdf (date of use: 11.05.2022)

6. TLS 1.3 and Proxies URL: https://www.imperialviolet.org/2018/ 03/10/tls13.html (date of use: 11.05.2022)

7. Apache Thrift vs Protocol Buffers vs Fast Buffers URL: https://www.eprosima.com/index.php/resources-all/performance/apache-thrift-vs-protocol-buffers-vs-fast-buffers (date of use: 11.05.2022)

8. M. Fowler, M. Foemmel, E. Hieatt, R. Mee, R. Stafford, Patterns of Enterprise Application Architecture, p. 476 // Pearson

9. The Facade Pattern, URL: https://www.pearsonhighered.com/assets /samplechapter/0/3/2/1/0321247140.pdf (date of use: 11.05.2022)

 ISSN 1560-8956