Verifying QUIC implementations using Ivy

Christophe Crochet
UCLouvain
Louvain-la-Neuve, Belgium
christophe.crochet@uclouvain.be

Tom Rousseaux
UCLouvain
Louvain-la-Neuve, Belgium
tom.rousseaux@uclouvain.be

Maxime Piraux
UCLouvain
Louvain-la-Neuve, Belgium
maxime.piraux@uclouvain.be

Jean-François Sambon
UCLouvain
Louvain-la-Neuve, Belgium
jean-francois.sambon@uclouvain.be

Axel Legay
UCLouvain
Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

ABSTRACT

QUIC is a new transport protocol combining the reliability and congestion control features of TCP with the security features of TLS. One of the main challenges with QUIC is to guarantee that any of its implementation follows the IETF specification. This challenge is particularly appealing as the specification is written in textual language, and hence may contain ambiguities. In a recent work, McMillan and Zuck proposed a formal representation of part of draft-18 of the IETF specification. They also showed that this representation made it possible to efficiently generate tests to stress four implementations of QUIC. Our first contribution is to complete and extend the formal representation from draft-18 to draft-29. Our second contribution is to test seven implementations of both QUIC client and server. Our last contribution is to show that our tool can highlight ambiguities in the QUIC specification, for which we suggest paths to corrections.

CCS CONCEPTS

• **Networks → Formal specifications**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

QUIC, Formal specification, Ivy, Verification, RFC9000, draft-29, Interoperability, Testing

ACM Reference Format:
Christophe Crochet, Tom Rousseaux, Maxime Piraux, Jean-François Sambon, and Axel Legay. 2021. Verifying QUIC implementations using Ivy. In *Workshop on the Evolution, Performance and Interoperability of QUIC (EPIQ'21), December 7, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3488660.3493803

1 INTRODUCTION

QUIC is a new network protocol intended to make the Internet faster, more secure and more flexible. It is designed to replace the entire TCP/TLS/HTTP stack and is built above UDP. QUIC is now widely adopted and many other applications are made compatible with the protocol. This includes, for example, MQTT[15] or DNS[11]. As QUIC is getting deployed, ensuring that QUIC implementations meet the set of requirements defined by the QUIC specification is critical. This specification is an English text document describing the protocol. It is discussed by the Internet Engineering Task Force (IETF) and is split across several Request For Comments (RFC) documents. The main document is RFC9000, containing more than 250 MUST statements indicating properties that must be met by all implementations [13]. As RFCs lack a formal definition, they can lead to ambiguities of understanding [25].

Several approaches have been proposed to verify that QUIC implementations follow the specification requirements. The most common approach, called *interoperability testing*, manually generates sets of tests from the requirements and then compares QUIC implementations with respect to those sets. This approach has been used in the *QUIC-Tracker* test suite [22] and the *QUIC Interop Runner* [23]. Albeit such approach sounds appealing, it is limited by the capacity to manually produce interesting test suites from the requirements. Another approach [8] is to produce a mathematical model for the protocol and its requirements, and then use formal verification to automatically assess correctness. This approach is sound and precise. However, as it is limited to models only, it does not guarantee that subsequent implementations fulfil the requirements.

In a series of recent work [18, 19], a trade off between the two approaches has been proposed. The authors implemented a formal language to specify protocols requirements called Ivy. Any specification written in Ivy automatically generates various and well-distributed test cases that can be exercised on QUIC implementation. In their work, the authors verified several requirements of draft-18 of the QUIC specification. However, many requirements were left unimplemented, e.g. the management of the transport errors by QUIC implementations.

Our three contributions are summarized as follows.

(1) We contribute to the formal specification of QUIC in Ivy by implementing a significant part of draft-29 requirements, one of the latest versions of the QUIC specification. This comprises new requirements and requirements in draft-18 the authors did not model. We also update the existing requirements to the new specification. We modify Ivy to support wider variables, i.e. larger than 8 bytes, allowing the tool to explore more parts of the specification.

EPIQ'21, December 7, 2021, Virtual Event, Germany
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9135-1/21/12…$15.00
https://doi.org/10.1145/3488660.3493803

(2) We automate the use of the resulting Ivy model. We use the generated tests against seven server implementations and seven client implementations. This demonstrates that the tool can be used beyond the four original implementations tested by the authors.
(3) We discuss the results we obtained and identify different types of errors in each implementation. The diversity of the results highlights contradictions and ambiguities in the QUIC specification. These findings can be used to improve the QUIC specification. We also highlight a significant disparity in the test results and hence a difference in maturity of the protocol implementations. We provide hypothesises on the causes of the tests failures.

2 OVERVIEW OF THE IVY MODEL FOR QUIC

A formal Ivy model of the protocol is defined as a set of components linked by the relationship between their input/output. Such model represents an *abstraction* of the specification entities as a whole. Each component represents a part or a layer of the QUIC stack. This includes, for example, the frame layer ① or the packet layer ② presented in Figure 1. The shim component ③ is used to send and receive packets over the network. For each received packet, the shim calls the packet_event action. It is an Ivy procedure containing all the requirements directly linked to the QUIC packet specification and raises an error if a requirement is violated. It checks, for instance, that the packet number always increases. The frames are also handled similarly with their corresponding action. In Figure 1, one can see that the set of requirements is linked to the packet component ②.

In their original work, the authors model most of the QUIC specification requirements of draft-18 except for the Retry, Version Negotiation and 0-RTT packets and one frame type, the RETIRE_CONNECTION_ID. The transport error codes were not implemented either. A limitation of the methodology is that quantitative-time requirements cannot be modelled. This includes all the requirements for congestion control and recovery.

We now summarize our contributions to the Ivy model of QUIC's specification. Our first contribution is to update this model to draft-29 of the protocol, in order to implement its new requirements. For instance, we updated the shim to the new wire format. We chose draft-29 as it was the latest version of the QUIC specification when starting this work, and remained widely supported by implementations during our study.

Our second contribution is to extend the model with additional requirements for existing and new transport parameters. As an example, the preferred_address transport parameter was not entirely defined in the original work, lacking a check for forbidden migration. We also add the transport error codes management tests. For this, we create a different model which deliberately does not follow a given QUIC requirement in order to generate illegal frames and packets. They are used to check whether an implementation reacts as expected, i.e. returning a specific transport error code, when receiving these frames and packets. Testing this feature is important since one can use the error code as an indication to fall back on TCP. Finally, we modify Ivy to support Connection IDs (CIDs) up to 16 bytes, increasing the domain of values that can be tested and thus the coverage of our test tool. It allows testing more implementations without modifying their source code to cope with this restriction. It allows us to relax the limitation of CIDs from exactly 8 bytes to any length up to 16 bytes. Due to space constraints, our Ivy specification is available online [4].[1]

In Figure 2, we estimate the coverage of requirements that can be handled with our Ivy model. We distinguish five categories. The first one contains requirements that cannot be verified because they refer to internal state, lack of formal definition or cannot be modelled in Ivy due to the technique limitations. The second category contains requirements defined in the original work that we updated to draft-29. The third category contains requirements for which we implemented a complete model in the tool. The fourth category contains requirements for which we implemented a partial model, i.e. requirements for which only a part can be verified. Finally, the last category contains requirements that we leave as future work. We will now detail how these requirements are verified.

The extent of our additional coverage is thus defined by the third and fourth category of requirements. For some new requirements, new tests were created such as for requirements on transport error codes. For each tested error, we added a specific test variant exercising it.

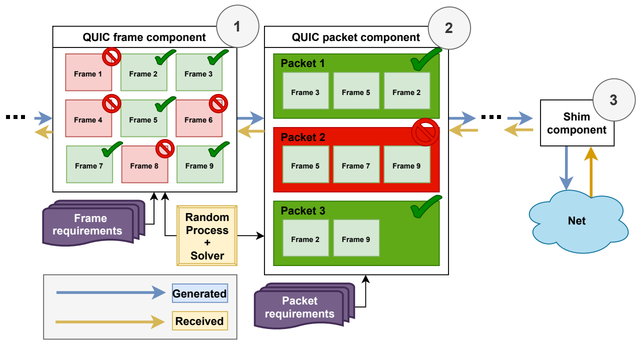

Figure 1: The QUIC frame component (resp. QUIC packet component) randomly generates frames (resp. packets) meeting its requirements, selects and transmits some to the next component. The shim is the interface between the model abstract representation and the network concrete wire format.

3 GENERATING TEST CASES

Our objective is to test various implementations of a QUIC server and client. This is done by generating tests, i.e. sequences of frames and packets, from the formal Ivy model of the specification. The series of tests are generated following the *Network-Centric Compositional Testing* [18] methodology. This approach starts from a formal representation of the requirements given in the specification, that we described in Section 2, and generates traces, i.e. sequences of frames and packets, which are sent to tested implementations.

The tests are distinguished by the type of frames they can generate and by the distributions of those frames in the sent packets. The tester can manually put weights on the different frame types to influence their distributions. The weights have a default value

[1]https://github.com/ElNiak/QUIC-Ivy/tree/quic_29/doc/examples/quic

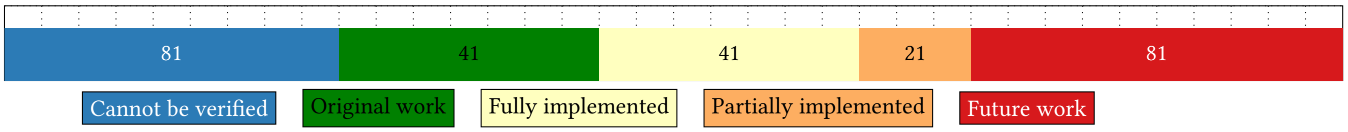

Figure 2: Tool coverage of requirements in the QUIC specification

of 1. A higher (resp. lower) weight will increase (resp. reduce) the relative probability of the associated type of frame to be generated. For the distributions, Ivy uses a Monte Carlo sampling method [24] relying on these weights.

By influencing the distributions of frames and packets and letting the solver choose them rather than spelling out the exact frames and packets that are sent, our approach guarantees that the generated frames and packets are valid, i.e. they follow the specification requirements.

Each test has a set of parameters that include the destination IP address and port, and the transport parameters sent.

When executing a test with Ivy, a sequence of frames and packets are generated and sent to the implementation under test. The test stops when a requirement is not true (failing test), or when the connection closed as expected, i.e. when all the data is successfully sent (passing test).

Apart from setting the type of generated frames and the input parameters of the test, one can also add new requirements to the formal specification of QUIC. By moving some of these requirements from the general model to the tests, one has more flexibility in the tests. For example, one could fix the version of QUIC used in the base formal specification but this is inconvenient when one wants to change the version in different tests, for instance for version negotiation. Other requirements can be used to create illegal packets. In this case we add requirements such that the model is not conforming anymore to the QUIC specification. As the solver guarantees that the generated frames and packets follow these added requirements it will produce non-conform frames/packets.

3.1 Example

We illustrate how a test is created with an example where we generate sequences of frames and packets. As explained in Section 3, each test is based on our formal specification of QUIC. We first manually fix the input parameters of the test. In this case, we set the IP addresses and the port used during the test. The tested implementation is a server.

```
# Tester address
parameter client_addr : ip.addr = 0x7f000001
parameter client_port : ip.port = 4987
# Tested implementation address
parameter server_addr : ip.addr = 0x7f000001
parameter server_port : ip.port = 4443
```

Then, we choose the set of frames generated and the distribution of these frames in the packets. In the following example, we decide to generate ACK, STREAM, CRYPTO and PATH_RESPONSE frames. Then we set the relative weight for the PATH_RESPONSE to five. It increases the relative probability to generate this frame. This ensures that the path validation during migration has a high chance of passing.

```
# Allow generation of a frame
export frame.ack.handle
export frame.stream.handle
export frame.crypto.handle
export frame.path_response.handle
# Relative weight (all other weights = 1)
attribute frame.path_response.handle.weight = "5"
```

Finally, we also add specific requirements to the generated frames and packets by refining events already present in the model (i.e. packet_event). A requirement is indicated in Ivy with the keyword require and is followed by a condition. Below, we find a simple example where we randomly generate an Initial packet (packet_event) with an *invalid* "Token" field, in this case we expect a non-zero length token (cf. line 6).

```
1  # Action already present in the model
2  before packet_event(src:ip,dst:ip,pkt:quic_packet){
3      if _generating {  # Wrong field generation
4          # Applied only on generated packet
5          # [...]  new features
6          require ~(pkt.token.end = 0);
7      };
8      # new feature (for configuration purpose)
9      require pkt.long -> pkt.pversion = 0xff00001d
10 }
```

As this situation is a non-conform one, we expect to receive a PROTOCOL_VIOLATION error code in a CONNECTION_CLOSE frame or no connection at all. This can be modelled with the special instruction _finalize allowing to check whether some requirements are eventually fulfilled.

```
export action _finalize = {
    require is_protocol_violation |  ~handshake_done;
}
```

We now give some details on tests made to stress the implementations of the server and the client. Observe that the main reason to make a distinction between those two entities is that some parameters of the client and of the server are incompatible. As an example, the preferred_address transport parameter is only available at the server side. Moreover, the generation of some frame such as the HANDSHAKE_DONE is forbidden to the client.

3.2 Server implementations tests

We have generated 23 tests to stress seven implementations of QUIC servers. Some of those tests are briefly described in Table 1.

The majority of the separate tests files concerns transport error codes. We developed 16 error code tests as each requirement requires a separate test file. As illustrated in Section 3.1, testing a specific property requires a separate test. The other tests differ by the generated frames.

quic_server_test_stream	
Generated frames:	STREAM, ACK, PATH_RESPONSE, CRYPTO
Test output:	Requests `/index.html`. Stops once the response stream is closed. Graceful connection closure is expected. Multiple connection migration are allowed. Several bidirectional streams are opened to test the streams mechanics.
quic_server_test_unknown	
Generated frames:	STREAM, ACK, PATH_RESPONSE, CRYPTO, UNKNOWN
Test output:	Verifies handling of UNKNOWN frame types by the implementation in application encryption level. The test expects to receive a FRAME_ENCODING_ERROR
quic_server_test_unknown_tp	
Generated frames:	STREAM, ACK, PATH_RESPONSE, CRYPTO
Test output:	Verifies handling of UNKNOWN transport parameters by the implementation. The implementation is expected to ignore the TP and to continue the connection.

Table 1: Server tests description

Let us illustrate the approach on the `quic_server_test_stream` test. In this test, we generate `STREAM` frames until the number of requests is reached or until the maximal data defined during the handshake is exceeded. Each time a generation or reception of a packet occurs, the whole specification will be tested. Moreover, as soon as one of the requirements is not satisfied, the test fails.

3.3 Client implementations tests

We implemented 14 tests to stress seven clients' implementations. Most of these tests are similar to their server counterpart. Due to space limitations, we only detail in Table 2 two errors handling tests specific to the client's implementation.

For the first test, we set the "Length" field of the token to zero, which is not allowed by the specification. For the second one, we verify that the `preferred_address` transport parameter contains zero-length connection ID and the tested implementation throws a `TRANSPORT_PARAMETER_ERROR`.

We do not test connection migration. Indeed, as it is triggered by the client, testing such feature requires to change manually the tested implementation, which we decided not to do in this work.

quic_client_test_new_token_error	
Generated frames:	STREAM, ACK, HANDSHAKE_DONE, CRYPTO, NEW_TOKEN
Test output:	NEW_TOKEN frames with length field set to 0 are sent and the test expects to receive a PROTOCOL_VIOLATION transport error in return.
quic_client_test_prefadd_error	
Generated frames:	STREAM, ACK, HANDSHAKE_DONE, CRYPTO
Test output:	The preferred_address transport parameter is set with a zero length CID field and the test expects to receive a TRANSPORT_PARAMETER_ERROR in return.

Table 2: Client tests description

4 TESTED IMPLEMENTATIONS

We selected several implementations written using different programming languages. These implementations often use different `TLS` implementations. Some of them are mature implementations while others are newer. Some are intended for production while some for research and specification feedback purposes. We found them through the `QUIC` Working Group wiki page listing known `QUIC` implementations [10].

Implementation	Language	SLOC	Company	Version
picoquic [2] picotls [14]	C	84k	Private Octopus H2O	ad23e6c 47327f8
lsquic [12] boringssl [9]	C	129k	LiteSpeed Tech. Google	v2.29.4 a2278d4
quic-go [17]	Go	73k	-	v0.20.0
quinn [1]	Rust	41k	-	0.7.0
aioquic [16]	Python	19k	-	0.9.3
quiche [3]	Rust	58k	Cloudflare	0.7.0
quant [6]	C	18k	NetApp	29
mvfst [7]	C++	105k	Facebook	36111c1

Table 3: Tested implementations and their versions

5 RESULTS

We run 100 iterations of each test on each implementation. This is to allow diverse behaviour to be captured and evaluated by the tool. Each iteration generates different packets traces following the random generation approach described in Section 3. Each test was run on the loopback interface of a single machine, for which we assume this link to be perfect. The results obtained against `QUIC` servers are summarised in Table 4. Those for `QUIC` clients are summarised in Table 5. In these two tables, each column contains the results of one `QUIC` implementation. Each row contains the results of a test. The success rate in each cell represents the percentage of test runs satisfying all the specification requirements.

Observe that the interpretation of statistical models based on our results could be biased by the distribution of some errors. Indeed, a few requirements could be violated for a large number of test iterations. This would lead the test to have a very low success rate while the implementation could meet all the other requirements verified by this test. For example, the `picoquic` implementation fails one requirement during connection migration. It lowers the success rate of the `stream` test to around 50% while only one requirement is violated.

Rather, these success rates depict an insight of the implementations maturity. It fulfils our objective of detecting errors in implementations rather than reasoning statistically. Indeed, the detailed results express the requirements which are not met, which gives clues for the debugging.

We will now present the two main categories of problems found. The first one concerns all problems related to migration and the second one concerns those related to the transport errors. Note that due to the page limitation, we did not include all the results but these will be available in a technical version of the paper.

5.1 Migration issues

For the tests with only legal frames/packets, we do not detect many different errors. Most of the problems are linked to acknowledgement of acknowledgements and to the path validation process during a migration.

	quinn [1]	mvfst [7]	picoquic [2]	quic-go [17]	aioquic [16]	quant [6]	quiche [3]
stream	79%	6%	56%	95%	18%	12%	97%
max	85%	3%	47%	39%	27%	21%	96%
reset_stream	29%	7%	61%	100%	24%	5%	98%
connection_close	95%	37%	81%	63%	78%	40%	100%
stop_sending	100%	4%	48%	33%	33%	8%	96%
accept_maxdata	77%	12%	50%	68%	43%	21%	96%
unknown	95%	99%	99%	96%	0%	0%	100%
unkown_tp	84%	59%	98%	100%	68%	100%	96%
double_tp_err	0%	0%	100%	100%	0%	3%	100%
tp_err	100%	100%	0%	100%	0%	0%	0%
tp_acticoid_err	100%	0%	0%	0%	0%	100%	0%
no_icid_err	100%	100%	100%	100%	0%	0%	0%
token_err	100%	98%	100%	100%	100%	100%	99%
new_token_err	100%	0%	0%	84%	100%	0%	0%
handshake_done_err	100%	92%	89%	0%	86%	2%	77%
newcid_err	81%	85%	100%	9%	68%	93%	91%
max_limit_err	49%	41%	100%	0%	41%	16%	0%
blocked_err	70%	0%	0%	75%	0%	0%	100%
retirecid_err	87%	0%	86%	85%	0%	0%	0%
stream_limit_err	100%	63%	99%	98%	99%	10%	0%
newcid_length_err	84%	0%	2%	81%	0%	0%	91%
newcid_rtp_err	91%	0%	0%	90%	0%	0%	0%
max_err	0%	90%	100%	0%	0%	0%	0%

Table 4: Server - Successful test ratio

	quinn [1]	picoquic [2]	quic-go [17]	aioquic [16]	quant [6]	quiche [3]	lsquic [12]
stream	99%	51%	100%	97%	85%	52%	92%
max	100%	15%	100%	98%	85%	34%	100%
accept_maxdata	100%	93%	100%	97%	95%	82%	83%
unkown	100%	96%	99%	0%	0%	100%	0%
tp_unkown	100%	34%	99%	99%	100%	99%	96%
double_tp_error	0%	100%	100%	0%	0%	0%	0%
tp_error	0%	0%	100%	0%	0%	0%	0%
tp_acticoid_error	0%	0%	0%	0%	100%	0%	0%
no_ocid	0%	100%	100%	0%	0%	0%	0%
tp_prefadd_error	0%	100%	0%	0%	0%	0%	0%
blocked_error	99%	0%	97%	0%	0%	91%	98%
retirecoid_error	99%	99%	100%	0%	0%	0%	98%
new_token_error	98%	94%	96%	1%	0%	87%	100%
limit_max_error	0%	88%	0%	0%	81%	0%	0%

Table 5: Client - Successful test ratio

The first problem never leads to a crash and no major security issues were detected. We supposed that it is also linked to migration since this error amplifies when migration is allowed. This is enhanced when we compare the result of the client tests and the server tests. This error is less present when the client is tested. As a reminder, the server tests allow multiple connection migrations while the client tests disable it.

However the migration problems are more serious. Many security considerations are involved in guaranteeing authentication, confidentiality and integrity of the messages exchanged between endpoints with the migration in general. Many attacks are linked to the migration such as the "Peer Address Spoofing", "On-Path Address Spoofing" or the "Off-Path Packet Forwarding" describe in the QUIC specification.

5.1.1 *Migration issues: Case study I.*

We found one problem involving connection migration in the `mvfst` implementation. It considers the connection migration as invalid when it starts just after sending the HANDSHAKE_DONE. Note that the `disable_active_migration` transport parameter is not set according to packet traces. In this test the HANDSHAKE_DONE frame is acknowledged in a migrated connection.

According to Section 9 of `draft-29`, an endpoint cannot migrate before the handshake is confirmed.

The design of QUIC *relies on endpoints retaining a stable address for the duration of the handshake. An endpoint MUST NOT initiate connection migration before the handshake is confirmed, as defined in section 4.1.2 of* [QUIC-TLS]. .

QUIC specification `draft-29` section 9.

From the client point of view, the handshake is confirmed when it receives the HANDSHAKE_DONE frame.

In this document, the TLS *handshake is considered confirmed at the server when the handshake completes. At the client, the handshake is considered confirmed when a* HANDSHAKE_DONE *frame is received.* .

QUIC-TLS specification `draft-29` section 4.1.2.

From the server point of view, it is confirmed when the TLS handshake is complete, i.e. when the TLS endpoints exchanged a "Finished" message and verified the peer's "Finished" message. Thus, if a client migrates just after it has received a HANDSHAKE_DONE frame, it respects the specification. The server must accept the migration before it receives an ACK for the HANDSHAKE_DONE.

In this document, the TLS *handshake is considered* ***complete*** *when the* TLS *stack has reported that the handshake is complete. This happens when the* TLS *stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion [also confirmation for server] of the handshake depends on the perspective of the endpoint in question.* .

QUIC-TLS specification `draft-29` section 4.1.1.

However the `mvfst` server prevents the migration until it knows that the client considers the handshake as confirmed, i.e. when the server receives the ACK for HANDSHAKE_DONE. We suppose that `mvfst` wants to be sure that the client has received the HANDSHAKE_DONE before allowing it to migrate. In the case where the frame is acknowledged in the original connection, this problem does not arise. This is consistent with our hypothesis.

5.1.2 *Migration issues: Case study II.*

Another problem in the specification highlighted by our model is that polysemous requirements lead to different valid formal interpretations. This means that they are ambiguous. The best example is the address of the highest-numbered non-probing packet to which an endpoint should send its packets. Consider the following statement:

An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures *that an endpoint does not send packets to an old peer address in the case that it receives reordered packets .*

QUIC specification draft-29 section 9.3.

According to this rule, one should send the packet to the address of the highest-numbered non-probing packet. A non-probing packet is a packet containing any other frame than the probing frames which are PATH_CHALLENGE, PATH_RESPONSE, NEW_CONNECTION_ID, and PADDING frames. The requirement does not indicate the packet number space to consider. Thus one valid interpretation is to consider the highest-numbered non-probing packet among all possible ones. Since migration is allowed only after the handshake completed, one should probably consider only the application data space. However it is not written explicitly.

Moreover, the same paragraph also contains the following indication:

Receiving a packet from a new peer address containing a non-probing frame indicates that the peer has migrated to that address. In response to such a packet, an endpoint MUST start sending subsequent packets to the new peer address and MUST initiate path validation (Section 8.2) to verify the peer's ownership of the unvalidated address.

QUIC specification draft-29 section 9.3.

When this rule is used, responses of non-probing frames are sent to the new peer address. When we consider the highest-numbered non-probing packet among all packet number spaces, most implementations do not pass the requirement. This is the case with picoquic. However when we consider the highest-numbered non-probing packet among the application data space, some implementations failing with the above interpretation do pass the requirement. This is also the case for picoquic. This result demonstrate that both chosen interpretations of the requirement, which are valid from a formal point of view, lead to different results. Even if the most appropriate interpretation may seem trivial for an implementer, the ambiguity should be removed.

5.2 Transport error code issues

Concerning the management of the transport error codes, we observed their requirements are not widely respected among the tested implementations. Various types of errors have been observed. We faced some implementations that do not implement one of these requirement. For example, the transport error codes management is almost completely implemented in some implementations (e.g quinn) and barely implemented in other ones (e.g. quant) as reflected in Table 4. As the application layer may use these error codes to fall back on a TCP connection, the transport error codes management is important.

Another type of problem is when implementations use the wrong encryption level. This is mostly the case for aioquic or quinn. Those two implementations use the 1-RTT encryption scheme to return the error. However this encryption level is not allowed at this time.

Some implementations accept frames restricted to their role, e.g. a server accepting server-sent frames. For example, the NEW_TOKEN frame is accepted by servers of picoquic and mvfst, meaning that they do not check whether the frame can be sent by their peer. These implementations do not report any PROTOCOL_VIOLATION error nor a local error.

We also observed some implementations to send error codes with inaccurate error messages. As an example, when the quic-go server receives a server-specific frame, such as HANDSHAKE_DONE, it closes the connection with an error message reporting that this frame is not allowed because of the encryption level. This is not accurate in this case.

Some implementations send an inaccurate error code. For instance, the mvfst implementation returns a PROTOCOL_VIOLATION when receiving a NEW_CONNECTION_ID frame with an invalid field, while a FRAME_ENCODING_ERROR is expected. Our manual analysis reported that it detects the problem as the error message reports the field as invalid.

Some implementations such as quant detect the error locally without reporting an error. This is not conforming to the specification. However, when we analyse a later version of quant, that problem has been solved as presented in Table 6. We can also observe that the management of invalid transport parameters is correctly handled in this version, i.e errors are detected and the correct transport error codes are sent.

	quant *29*	**quant *master***
double_tp_error	🟥 3%	🟩 100%
tp_error	🟥 0%	🟩 100%
tp_acticoid_error	🟩 100%	🟩 100%
no_icid_error	🟥 0%	🟩 100%

Table 6: Quant transport parameter: before/after

CONCLUSION AND FUTURE WORK

In this paper, we applied the Ivy modelization framework to QUIC and produced a model verifying a large part of the requirements of the QUIC specification. The model contains new and updated requirements from the original work. We automated the use of the model in a series of tests against QUIC clients and servers. We showed the diversity of results we obtained with our tool and the variety of errors encountered. In addition, the results we obtained illustrate the ability of the tool to highlight ambiguities in the specification and generate executions following different interpretations of these ambiguities. This shall be exploited to improve the IETF specification of QUIC.

We see several directions for future work. One is to update our model to the RFC9000 specification. The differences between draft-29 and RFC9000 are minor and thus we expect this work to be relatively small. The RFC defines additional security considerations and transport error codes, adds a new subsection on flow control performance and removes one regarding the QUIC connection life cycle. Only a few MUST requirements were added. The second one is to model the features of QUIC that are missing in our Ivy model, such as the Retry and Version Negotiation packets. The third one is to consider extensions of QUIC such as Multipath QUIC [5], Forward Erasure Correction (FEC) [20], the Unreliable Datagram extension [21].

ACKNOWLEDGMENTS

This work was partially supported by funding from the Walloon Government (DGO6) within the MQUIC project.

REFERENCES

[1] Jean-Christophe Begue & al. Benjamin Saunders, Dirkjan Ochtman. 2018. quinn. https://github.com/quinn-rs/quinn/tree/0.7.0

[2] Bastian Köcher & al. Christian Huitema, steschu77. 2017. picoquic. https://github.com/private-octopus/picoquic/tree/ad23e6c3593bd987dcd8d74fc9f528f2676fedf4

[3] Cloudflare. 2018. quiche. https://github.com/cloudflare/quiche

[4] Christophe Crochet and Jean-François Sambon. 2021. Towards verification of QUIC and its extensions. (2021). http://hdl.handle.net/2078.1/thesis:30559

[5] Quentin De Coninck and Olivier Bonaventure. 2017. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (Incheon, Republic of Korea) *(CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 160–166. https://doi.org/10.1145/3143361.3143370

[6] Lars Eggert. 2016. quant. https://github.com/NTAP/quant/tree/29

[7] Facebook. 2019. mvfst. https://github.com/facebookincubator/mvfst

[8] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: Closed-Box Analysis of Network Protocol Implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/3452296.3472938

[9] Google. 2014. boringSSL. https://boringssl.googlesource.com/boringssl/

[10] QUIC Working Group. 2016. quicwg/base-drafts. https://github.com/quicwg/base-drafts/wiki/Implementations

[11] Christian Huitema, Melinda Shore, Allison Mankin, Sara Dickinson, and Jana Iyengar. 2019. *Specification of DNS over Dedicated QUIC Connections*. Internet-Draft draft-huitema-quic-dnsoquic-07. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-huitema-quic-dnsoquic-07 Work in Progress.

[12] LiteSpeed Technologies Inc. 2017. lsquic. https://github.com/litespeedtech/lsquic/tree/v2.29.4

[13] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[14] steschu77 Bastian Köcher & al. Kazuho Oku, Christian Huitema. 2016. picotls. https://github.com/h2o/picotls/tree/47327f8d032f6bc2093a15c32e666ab6384ecca2

[15] Puneet Kumar and Behnam Dezfouli. 2019. Implementation and analysis of QUIC for MQTT. *Computer Networks* 150 (2019), 28–45.

[16] Jeremy Lainé. 2019. aioquic. https://github.com/aiortc/aioquic/tree/0.9.3

[17] Lucas Clemente & al. Marten Seemann. 2016. quic-go. https://github.com/lucas-clemente/quic-go

[18] Kenneth L McMillan and Lenore D Zuck. 2019. Compositional Testing of Internet Protocols. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 161–174.

[19] Kenneth L. McMillan and Lenore D. Zuck. 2019. Formal Specification and Testing of QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 227–240. https://doi.org/10.1145/3341302.3342087

[20] François Michel, Quentin De Coninck, and Olivier Bonaventure. 2019. QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC. In *2019 IFIP Networking Conference (IFIP Networking)*. IEEE, 1–9.

[21] Tommy Pauly, Eric Kinnear, and David Schinazi. 2018. An Unreliable Datagram Extension to QUIC. *Internet Engineering Task Force.(September 2018). draft-pauly-quicdatagram-00* (2018).

[22] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. 2018. Observing the Evolution of QUIC Implementations. *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Dec 2018). https://doi.org/10.1145/3284850.3284852

[23] Marten Seemann and Jana Iyengar. 2020. Automating QUIC Interoperability Testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3405796.3405826

[24] Alexander Shapiro. 2003. Monte Carlo sampling methods. *Handbooks in operations research and management science* 10 (2003), 353–425.

[25] Jane Yen, Ramesh Govindan, and Barath Raghavan. 2021. Tools for Disambiguating RFCs. In *Proceedings of the Applied Networking Research Workshop* (Virtual Event, USA) *(ANRW '21)*. Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/3472305.3472314