Formal specification and testing of QUIC

Kenneth L. McMillan
Microsoft Research
Redmond, WA
kenmcmil@microsoft.com

Lenore D. Zuck
University of Illinois, Chicago
Chicago, IL
lenore@cs.uic.edu

ABSTRACT

QUIC is a new Internet secure transport protocol currently in the process of IETF standardization. It is intended as a replacement for the TLS/TCP stack and will be the basis of HTTP/3, the next official version of the hypertext transfer protocol. As a result, it is likely, in the near future, to carry a substantial fraction of traffic on the Internet. We describe our experience applying a methodology of compositional specification-based testing to QUIC. We develop a formal specification of the wire protocol, and use this specification to generate automated randomized testers for implementations of QUIC. The testers effectively take one role of the QUIC protocol, interacting with the other role to generate full protocol executions, and verifying that the implementations conform to the formal specification. This form of testing generates significantly more diverse stimuli and stronger correctness criteria than interoperability testing, the primary method used to date to validate QUIC and its implementations. As a result, numerous implementation errors have been found. These include some vulnerabilities at the protocol and implementation levels, such as an off-path denial of service scenario and an information leak similar to the "heartbleed" vulnerability in OpenSSL.

ACM Reference Format:
Kenneth L. McMillan and Lenore D. Zuck. 2019. Formal specification and testing of QUIC. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3341302.3342087

The work of L. Zuck was partially funded by NSF award CCF-1564296.

SIGCOMM'19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342087

1 INTRODUCTION

QUIC is a new Internet secure transport protocol, introduced by Google and currently in the process of IETF standardization [17]. It is intended as a replacement for the TLS/TCP stack for secure, authenticated communication over ordered channels. QUIC is designed to provide improved performance in various aspects, including initial latency, handling of multiple streams, mobility, data loss detection, and resistance to denial of service attacks. Currently, variants of the protocol are estimated to carry greater than 5% of traffic on the Internet, principally generated by video streaming services [34]. QUIC has recently been selected as the basis of HTTP/3. Thus, after standardization, it is reasonable to expect that the protocol will carry a significantly larger portion of Internet traffic.

The QUIC standard is being developed in the form of an RFC: an English-language document that provides extensive guidance for implementers of the protocol, but is nonetheless ambiguous and broadly open to interpretation. The primary mechanism for resolving these ambiguities and validating the correctness of the protocol design is to produce multiple independent implementations, and to test these implementations for interoperability [37]. As a result, these implementations represent a kind of commentary on the standard document, providing concrete interpretations where the language may be vague, unclear or contradictory.

While effective, this methodology leaves something to be desired from the point of view of clear standardization and of implementation compliance. First, the knowledge implicit in the implementations is not captured in any precise and rigorous way. Second, since the implementations do not represent the *full* diversity of behaviors that the protocol allows, interoperability testing provides very limited test coverage. Third, because actual protocol compliance is never tested, interoperability is insufficient to guarantee that current implementations will be interoperable with future ones.

The risks of interoperability testing are illustrated in the history of SSL/TLS. Consider for example the issue of "version intolerance." According to the protocol, servers presented with a request for an unsupported protocol version must refuse the connection. In practice, however, this behavior was often not properly tested, since existing clients did not generate future version numbers. Thus the server

implementations typically behaved incorrectly. For the same reason, middleboxes that interpreted protocol messages also exhibited version intolerance. To cope with this, browsers implemented voluntary downgrade strategies. These *ad-hoc* extensions to the protocol proved to be vulnerable, leading to downgrade attacks such as POODLE [13]. Attempts to mitigate these attacks were then sometimes thwarted by the fact that clients and servers did not fully implement their protocols. Various other attacks such as SMACK [3] and POODLE TLS [33] could have been prevented if the implementations were tested for compliance with the protocol.

From the experience of TLS, it is clear that more than interoperability is needed. It is important to have an *unambiguous statement* of the protocol standard, and to test implementations for actual *compliance to the standard*, and not just for interoperability. Moreover, it is necessary to test implementations in *adversarial environments* and not just in the benign environment of other existing implementations.

In this paper, we present a methodology that attempts to do this by applying light-weight formal methods. We develop a formal specification of QUIC based on the draft standards documents, and refine it by applying *specification-based testing* to the implementations. This simultaneously captures the protocol knowledge implicit in the implementations, and tests the implementations for compliance to the protocol.

Specification-based testing uses a formal specification to both generate test inputs for a system and validate its outputs. An example of this idea is the QuickCheck tool in the Haskell programming language [11]. Using a formal specification of a function f, QuickCheck randomly generates a small number of test inputs on which it applies f and verifies that the output satisfies the specification. The situation with protocols is, however, more complex than that of functions, since the tester must produce a *sequence* of inputs that are responsive to the prior system outputs. Moreover, a protocol consists of multiple roles, and one must verify that the *composition* of these roles complies with the overall protocol specification. This motivates a *compositional* approach to specification-based testing that applies ideas from a formal technique called *modular assume/guarantee reasoning* [16].

We will describe our new methodology and our experience in specifying and testing QUIC, including the positive results that were achieved as well as the limitations and difficulties we encountered. We found that it was possible to formalize a substantial part of the QUIC standard. Our specification is sufficiently complete that a randomized tester taking the client role can interact with the real servers, successfully exchanging data without the server detecting protocol errors.

In the process of testing we discovered many errors in the implementations, including internal errors such as code assertion failures and memory faults, and protocol compliance errors. As a side effect, compliance testing also discovered vulnerabilities at both the protocol and implementation levels. For example, an off-path denial-of-service scenario was discovered and mitigations were added to the standard. An information leak similar to the "heartbleed" error in OpenSSL [12] was also discovered in one implementation. These are described in Section 5.

On the negative side, we found that a large fraction of the draft standard could not be formalized in a testable way. A frequent cause of this is that the statements refer to internal events within the implementation that have no clear definition or that cannot be inferred by observations on the wire. We give examples of these and other issues in Sections 4.2 and 6.2, and consider possible solutions to these problems. In this case study we specify and test only safety properties. Extending the methodology to liveness and real-time properties is discussed in Section 6.1. QUIC has some requirements that are well-suited to external specification and testing, and some that are not. We think this situation is typical of Internet protocols.

1.1 Related work

There are numerous approaches to generating adversarial tests for network protocols. We consider first the approaches that are *not* based on formal specifications. One approach is to execute or simulate an existing implementation, and mutate the behaviors of some nodes, for example, by delaying, duplicating, or modifying messages. An example of this is Gatling [20] which systematically explores a space of mutations searching for performance bugs. This is an instance of "fuzz testing", a general approach that can be highly effective in discovering bugs and vulnerabilities. However, it tells us nothing about conformance to the protocol standard, and does not result in a formal specification of the protocol.

Another such approach is "white-box" testing. Using tools such as KLEE [8] one traces the internal control flow paths of the system under test and uses symbolic execution and SMT solvers [2] to discover input values that stimulate branches not taken. In fact, there is some work applying white-box testing to QUIC [32]. That work does not address protocol specification or compliance, though it is not impossible in principle. Interoperability testing using this approach [29, 32] has the disadvantage noted above of being insufficiently adversarial.

Another alternative to formally specifying the wire protocol is to produce a *formally verified reference implementation* [4] or simply to prove properties of an existing implementation [10]. Again, there is no check of compliance to a common standard in these works.

Techniques that *do* address compliance include model-based testing (MBT) [7, 28, 35] and its precursors in the area of protocol conformance testing (see [26] for a survey). In MBT, an abstract model of the system is constructed, containing *controllable* events that are generated by a tester, and *observable* events that are generated by the system under test. The protocol specifications are given as finite-state machines (FSMs). This entails either abstracting away non-finite aspects of the protocol or restricting the model to some fixed sample of data values. The finite-state machine is systematically explored to generate test scenarios, either online or offline. This effectively provides a heuristic for generating adversarial tests. With an FSM model, some method is needed to fill in the concrete data parameters of messages. This may be done in a systematic way as in [35] or an ad-hoc way as in [7]. Generally, the need to extend FSMs in some way to account for data leads to significant complexity in these formalisms.

The compositional testing approach used here differs in several significant ways from existing work on MBT. First, *we do not model QUIC as an FSM*. A finite-state machine cannot communicate correctly with a QUIC node because of unbounded non-determinism in the node's responses. Instead, we take a constrained-random approach to generating tests based on a non-finite-state specification. Second, because we specify a closed system of communicating agents, the same specification is used to generate tests for both client and server roles. Compositional testing has the formal property that, if the system as a whole violates its specification, then some component must exhibit a failing test. This property is crucial for validating a specification. Without it, we may find that a legal output of one node is an illegal input for its peer. Thus, two nodes that pass all their tests may still fail when composed on the network. Compositional testing exposes such weaknesses in the specification. This is important when distilling a specification from implementations. Finally, work with finite-state models usually considers only some restricted aspect of the protocol. For example, [28] considers the connection state machine of TCP, but not TCP's data transmission. A similar approach is applied to TLS in [3, 7]. Here, we consider the full protocol state including the security handshake, with unbounded data, unbounded streams, unbounded connections and so forth.

Another effort that infers protocol specifications experimentally from implementations is the Network Semantics Project [5] that has developed a formal specification of TCP. In this work, the formal specification is used as a *test oracle*, that is, traces captured on the wire are checked for compliance. The specification is *not* used, however, for test generation. Checking traces can determine when the specification is too tight. By contrast, our compositional approach can also detect when the specification is too *loose*. That is, suppose the specification of an output of a protocol node is too weak. This same specification is used to generate inputs for its peer. Thus, we can detect the weakness by the fact that the peer misbehaves or flags a protocol error on a generated input. Without compositionality, we would almost certainly miss needed requirements, as QUIC is an order of magnitude more complex than TCP, and inferring formal requirements from the standard is challenging.

In other work, software API specifications are inferred automatically from run-time traces (*e.g.*, [1, 9, 30, 31]). This approach typically learns only a finite-state abstraction, which insufficient, as noted above. There has in addition been a great deal of work devoted to abstract modeling of network protocols that does not connect the models to implementations (*e.g.*, [19, 36, 38]).

The general approach of compositional or assume/guarantee testing that we take here was introduced for software verification [16] and has been applied to hardware [21]. Because the specification state and the messages contain significant amounts of data, these methods cannot be applied directly to QUIC. To solve this problem, we develop a specification methodology and corresponding optimized algorithms. This allows compositional testing to be applied in practice to complex, layered Internet protocols such as QUIC. The challenges in applying specification-based testing to QUIC are discussed in Sections 4 and 6. These were not insuperable, but they required us to adapt our approach to specification and randomized constraint solving in a number of ways.

To summarize, existing methods applicable to QUIC do not address the need we identify here to (1) distill a common, unambiguous wire specification from implementations and (2) test compliance of implementations in an adversarial environment.

Ethics statement: This work does not present ethical issues, as we handle no personal data.

2 AN INTRODUCTION TO QUIC

The QUIC protocol can be conceived approximately as a stack of layers, each of which provides one aspect of the overall transport service. These layers are depicted in fig. 1. At the bottom of the stack, UDP provides network services. Above this the packet protection layer provides secrecy by encrypting QUIC packets, which are encapsulated into UDP datagrams. Above this, the packet protocol provides loss detection using sequence numbers. Frames contain (among other things) ordered stream data and are encapsulated in packets. Each data frame carries a stream identifier, a sequence of bytes, and the offset of those bytes within the stream. This allows the stream data to be reconstructed at the receiving end in spite of datagram re-ordering and also

supports multiple independent streams. The security handshake protocol, a modified version of TLS 1.3, exchanges handshake messages using the frame protocol. Once a shared secret has been established by the handshake protocol, keys can be derived for encryption and decryption by the protection layer. At the top is the application layer, in which peers send and receive reliable, secure, authenticated data streams.

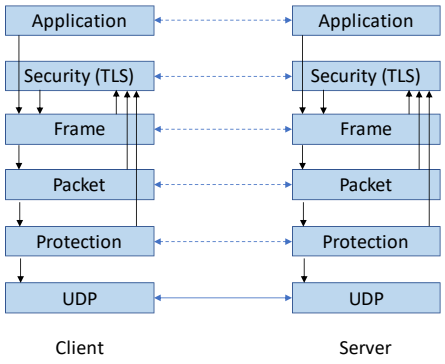

Figure 1: QUIC protocol layers. Arrows represent dependencies between layers.

This layered view of QUIC is useful, but somewhat misleading. We can think of each layer as providing a virtual connection between the peers, and relying on the layers below to implement this connection. However, the layering is not clean, as lower layers also rely on services of higher layers. Specification and testing might be easier if the protocols were cleanly layered. As we will see, it is still possible to exploit the decomposition of the protocol into events at different layers to improve the performance of testing.

The basic unit of communication in QUIC is the *connection*. A connection is a point-to-point channel that provides multiple independent data streams. A connection is established when a client sends an *initial packet* to a server. This packet provides a *connection identifier* (CID) to the server, a string of bytes that uniquely identifies the connection. It also contains a frame with the first security handshake message. The server responds with its own initial packet, containing the server's CID and the second handshake message. Subsequent *handshake packets* are protected with handshake keys derived from the initial messages. Once the handshake is complete, session keys are available and transmission of session data commences. At this point, each side can also transmit additional secret CID's. These are used to prevent tracking of the connection by third parties, for example when one of the peers migrates to a different network location. A special 0-RTT packet type allows the client to send early session data using a pre-shared key from a previous connection (a feature of TLS 1.3). QUIC packet types are thus partitioned into four *encryption levels* using different keys: initial, handshake, 0-RTT (for early data) and 1-RTT (for normal data with forward secrecy).

In addition to CID's, QUIC packets contain unique sequence numbers that are used to detect packet loss. A peer sends acknowledgment (ACK) frames to indicate packet sequence numbers that have been received. A packet that is not acknowledged after some length of time is considered lost. Rather than retransmit the packet, the peer retransmits its frames as needed in subsequent packets with different sequence numbers.

The protocol has a variety of frame types that are used for data transmission, loss detection, flow control, connection state management, management of CID's and so forth. The STREAM and CRYPTO frame types contain stream data and security handshake message data respectively. As noted, ACK frames contain acknowledgments. The MAX_DATA and MAX_STREAM_DATA frames are used by receivers to limit the amount of data a sender may send on a connection or stream. Other frames allow the sender to indicate that it wishes to send data, but is blocked by flow control, or to terminate a stream or connection, to stop transmission on a stream, to provide fresh CID's or retire them, and to validate that a peer controls a particular IP address.

3 SPECIFICATION METHODOLOGY

We present the specification and testing methodology informally, using examples. For a formal treatment of the underlying theory and algorithms, see [24].

We will specify the QUIC wire protocol using an abstract machine that monitors protocol events. In concept, this machine tracks all the QUIC-related events on the Internet. When an event occurs (say, the transmission of a QUIC packet) the machine first consults its state to determine whether the event is in fact legal according to the protocol. If so, it updates its state to reflect the occurrence of the event.

The specification is coded in a language called Ivy [23, 27]. In Ivy, the events associated with transmission of packets are modeled by an *action* such as the following:

```
action packet_event(src:endpoint,dst:endpoint,content:packet)
```

This describes a collection of events having three parameters: as source endpoint, a destination endpoint, and packet contents. The 'endpoint' datatype might be, for example, a representation of an IP address and port number, while the 'packet' would a record type having fields representing the packet contents.

The protocol state is stored in a collection of mathematical functions and relations. For example, suppose our packets have sequence numbers and our wire specification has a requirement that sequence numbers are never re-used by a source endpoint. To determine whether a packet event is

legal, we record in the protocol state the set of sequence numbers used by each endpoint, using a relation such as:

```
relation seq_used(S:endpoint,N:seq_num)
```

The intention is that this predicate is true when endpoint S has transmitted a packet with sequence number N. To express our specification, we first give the initial state of the monitor, like this:

```
after init {
  seq_used(S,N) := false
}
```

This initially sets seq_used(S, N) to be false for all values of S and N. Now we give the monitor code that runs before each packet event:

```
before packet_event {
  require ¬seq_used(src,content.seq);
  seq_used(src,content.seq) := true
}
```

The 'require' statement says that, when a packet event occurs, its sequence number has not already been used by the source endpoint. If this requirement is met, we record the fact that the source used the given sequence number. As a result, if any endpoint sends two packets with the same sequence number, the condition in the 'require' statement will be false, and we say the protocol has been violated. Notice that our specification is not an abstract implementation of the protocol, but is instead an executable monitor that observes the behavior of all protocol nodes.

To monitor an actual execution of the protocol, we write a 'shim' that captures the packet events, for example by tracing packets on the wire. For each packet, the shim calls the 'packet_event' action, invoking the above monitor code and flagging an error if a 'require' condition is false. In this way, we can watch the behavior of a collection of processes interacting on the network and check whether their collective behavior complies with the protocol specification.

In addition, the Ivy tool can compile a *generator* that produces random sequences of events that conform to the specification. At each step, the generator chooses an action at random (say, 'protocol_action') and builds a constraint over the action's parameters that is true exactly when all the 'require' conditions hold. This can be seen as either symbolic execution [6] or a weakest precondition computation [15]. With the help of an SMT solver [2] the generator then randomly selects a valuation of the parameters satisfying the constraint. The result in our example will be a random sequence of packet events in which no sequence number is sent twice by any endpoint.

Now suppose we wish to test a running instance of our protocol located at network address a. We partition the packet events into those controlled by the instance under test, and those controlled by its environment. Since we want the tester to generate only the latter, we add this requirement:

```
before packet_event {
  require generating → src ≠ a
}
```

Suppose we now want to test our protocol instance through the network using UDP. We create a shim that connects abstract packet events to actual UDP packets. The output part of the shim might look like this:

```
after packet_event {
  if generating {
    call udp.send(src,dst,serialize(contents))
  }
}
```

Here 'udp.send' is a binding for an underlying operating system service. This code says that after any generated packet event, the serialized packet contents should be sent in a UDP datagram with the given source and destination. The input part of the shim might look like this:

```
implement udp.recv(src:endpoint,dst:endpoint,data:bytes) {
  if src = a {
    call packet_event(src,dst,deserialize(data))
  }
}
```

This implements a call-back from the UDP interface that occurs when a datagram is observed. If it originates from endpoint under test, we create a corresponding packet event. If this event violates the requirement that endpoint a not re-use a sequence number, then an error is reported and the test stops.

Of course, the above assumes that we have the capability to generate IP headers with any source endpoint and to observe all datagrams produced by the instance under test. If this is not the case, we might open just a few sockets at specific endpoints and constrain the generator to use only these. We could also, for example, communicate packets to the instance by 'mocking' the sockets API that it uses. The main point is that the shim is just an ad-hoc mechanism that connects abstract events in the specification to real events in a system. This allows to write the specification in a generic way, and to adapt it various testing scenarios. It lets us use resources such as operating systems, networks and compilers to test, rather than running in a simulated environment.

If our protocol has multiple roles, say, 'client' and 'server', we can constrain the environment to generate only events controlled by one role or the other. Thus we can use the same specification to test either clients or servers. This is an example of the usefulness of the assume/guarantee paradigm: When testing the server role, we effectively *assume* that the client-controlled events are correct, since they are

generated from the specification. However, we must *guarantee* that server-controlled events are correct (that is, we monitor these events and stop the test if they are incorrect). Conversely, when testing the client role, we assume correctness of server-controlled events, and guarantee correctness of client-controlled events. Assume/guarantee testing has an important formal property: if the composition of the client and server ever violates the specification, then there must exist a failing assume/guarantee test for one of the two roles.

4 QUIC PROTOCOL SPECIFICATION

We now consider the formal specification of the QUIC protocol in the Ivy. In the sequel, we will use 'our specification' to refer to this specification, and 'the standard' to refer to the draft RFCs. Our specification and all code needed to run tests are available in open source [22].

4.1 Extensional and Intentional Specs

We wish to specify the protocol here in terms of behaviors visible on the wire, a style of specification we will refer to as *extensional*. This differs markedly from the approach of the RFC, which tends to describe the protocol in terms of how it should be implemented internally, a style we will refer to as *intentional*. An intentional specification is not well suited to testing because it refers to internal events that are not visible to the tester. On the other hand, an extensional specification is weaker, since it considers a behavior to be a protocol violation only if an external observer can *prove* that the protocol was violated.

As an illustration, the specification of TCP in [28] is intentional. In monitoring network traces, it was found that inferring non-deterministic internal events was too costly, and thus 'debug trace' events had to be used to resolve these choices [28, p. 12].

Here, we do not monitor any internal implementation events. Having said this, our specification does have some intentional aspects, for reasons of tractability of test generation. We formally specify actions at each layer of the protocol, from the application layer down to the packet layer. Some of these events (at the frame and security layers) are not externally visible. They are, however, easily inferred from the visible events. In Section 6 we will revisit the question of extensional specification and consider some aspects of the standard that cannot be captured in this way.

4.2 The packet protocol

The packet protocol is specified in terms of a single action corresponding to transmission of a QUIC packet:

```
action packet_event(src:endpoint,dst:endpoint,content:packet)
```

The packet protocol has four primary functions: (1) identifying connections, (2) generating sequence numbers for use in loss detection (3) determining the encryption level and (4) carrying frames. Note that actual retransmission in response to packet loss is performed at the frame layer. Packets are never retransmitted in QUIC.

```
1  before packet_event {
2    var src_cid := content.src_cid;
3    var dst_cid := content.dst_cid;
4    require connected(dst_cid) ∨ dst_cid = nonce_cid(src_cid);
5    require happens_after_init(contents) → connected(dst_cid);
6    if content.long {
7      require connected(dst_cid) → connected_to(dst_cid) = src_cid;
8    } else {
9      src_cid := connected_to(dst_cid)
10   }
11   var enc_lev := rtt1 if ¬ content.long
12                  else initial if content.ptype = pinitial
13                  else handshake;
14   require ¬seq_used(src_cid,enc_lev,content.seq);
15   seq_used(src_cid,enc_lev,content.seq) := true;
16   require frame_queue.count(src_cid,enc_lev) > 0;
17   require content.payload = frame_queue.frames(src_cid,enc_lev);
18   frame_queue.frames(src_cid,enc_lev) := frame_array.empty
19 }
```

Figure 2: Simplified specification of packet events

A simplified specification of 'packet_event' is shown in Fig. 2. The first task of a QUIC endpoint on receiving a packet is to associate it with a *connection*. We model a connection as a pair of unique identifiers associated with the connected application-layer processes. Since these are in one-to-one correspondence with initial CID's, we equate these two types of identifiers. We represent the set of established connections in the network with the following function and relation:

```
relation connected(C:cid)
function connected_to(C:cid) : cid
```

A CID is 'connected' if it is an end of some connection. In this case, the function 'connected_to' tells us the CID of the other end. These state variables are updated by an application layer event called 'establish' which establishes a connection.

The code that infers the connection from the packet contents is in lines 2 to 10. The long-format packets have both a source and a destination CID, while the short-format packets contain only a destination CID. A client initiates a connection by sending an initial packet. At this point, the client does not know the server's CID and it therefore uses a nonce value for the destination CID. However, the standard says that after receiving an initial packet from the server the client must switch to the CID provided by the server.

Here, we observe a difference between an intentional and an extensional specification: The moment when the client

'receives' (processes) the server initial packet is not observable on the wire. If we observe a client packet, we can't tell whether it was sent before or after a given server initial packet was processed unless there is a definite causal relationship between the two packets. Therefore, in our extensional specification, we require this: if a packet logically *must happen after* the peer's initial packet, then it uses the peer's CID. This is expressed at line 5 using a predicate 'happens_after_init', which is defined in Ivy as follows:

```
definition happens_after_init(P) = P.ptype ≠ initial ∨ P.contains_ack
```

That is, the only packets that are not causally dependent on the peer's initial packet are packets of initial type that do not contain ACK frames. Overall, the specification states that the destination CID is either the nonce or the peer's CID, and must be the latter if the packet happens after the peer initial packet. For long packets, if the peer's CID is used, the source CID must be connected to it. For short packets, the source CID is inferred from the destination CID. We have simplified here, since we do not consider that new CID's may be issued as aliases for the originals, and we do no consider RETRY packets.

Next, we determine the packet's encryption level based on its type (lines 11 to 13). For simplicity, we ignore 0-RTT packets. Then we consider packet sequence numbers. For correct packet loss detection, we require that the packet numbers are not re-used within a given encryption level. This requirement is reflected in lines 14–15.

The standard also states the sequence numbers of initial packets must be begin with zero and be strictly increasing. Here again, we see a difference between intentional and extensional specifications: Packet zero may be dropped, and packets observed on the wire may be re-ordered at the network layer, either because the network itself re-orders them, or because the implementation packs them into UDP datagrams out-of-order. For this reason, we can only infer that the packet numbers are non-increasing in the case of two packets that are necessarily causally ordered. There is no clear reason for this requirement, however. In testing, the implementations were found to be insensitive to arbitrary ordering of the sequence numbers of initial packets. Thus, we chose to drop this requirement as not usefully testable. This is one of many such intentional statements in the standard. Another example is the statement that the nonce CID must be 'unpredictable'. Clearly there is no general test for unpredictability, so compliance to this statement must be verified by some means other than testing.

Finally, we connect the packet and the frame protocols. Frame protocol events generate frames and enqueue them for transmission in packets. There is a frame queue for each connection end and encryption level. In lines 16–18, we require that there is at least one queued frame and that the packet payload is equal to the sequence of queued frames. Finally, we clear the queue, so that each frame is transmitted in at most one packet.

In this simplified description, we have omitted some additional requirements in the packet protocol relating to acknowledgments of acknowledgments and to detection of migration of the peer.

4.3 The frame protocol

Most of the content and complexity of QUIC is in the frame protocol. There are 20 frame types, each of which is modeled in Ivy using a record type. As an example, Figure 3 gives a simplified specification for STREAM frames, which carry application data. The stream frame action is defined as follows:

```
action frame.stream.event(f:frame.stream, src_cid:cid, dst_cid:cid,
                          e:encryption_level)
```

For each frame type, the parameters of the corresponding action are the frame content, the source and destination CID's and the encryption level.

```
1  before frame.stream.event {
2    require connected(dst_cid) ∧ connected_to(dst_cid) = src_cid;
3    require e = rtt1 ∧ sec.established_1rtt_keys(src_cid);
4    require f.offset + f.length ≤ app.data_end(src_cid,f.id);
5    require f.data = app.data(src_cid,f.id).segment(f.offset,f.offset+f.length);
6    require f.fin ↔ app.closed(src_cid,f.id)
7                ∧ f.offset+f.length = app.data_end(src_cid,f.id);
8    require f.offset + f.length ≤ stream_max_data(dst_cid,f.id);
9    require stream_id_allowed(dst_cid,f.id);
10 }
11
12 after frame.stream.event {
13   call enqueue_frame(src_cid,f,e);
14 }
```

Figure 3: Specification of STREAM frame events

A stream frame 'f' transmits 'f.length' application bytes at offset 'f.offset' from stream id 'f.id'. Lines 2 to 3 say that the source and destination CID's must be connected, and the encryption level must be '1-RTT' (that is, STREAM frames must be sent only with 1-RTT encryption). When a frame event occurs, we also require that the necessary encryption keys have been generated by the security layer. Lines 4 to 7 relate the STREAM frames to the application layer. They state that the bytes transmitted must be in the range sent by the application, that the data transmitted must be equal to the segment of the application data of the given length at the given offset. The frame contains a 'fin' bit indicating the end of the stream. It is true if and only if the application has finished transmitting and the frame contains the end of the stream (line 6). Lines 8 to 9 relate to flow control. They

state that the transmitted bytes do not exceed the maximum allowed by the receiver and that the sender does not attempt to use a steam ID larger than the receiver allows or attempt to use a new stream id that it is not allowed to open. After a frame event the action 'enqueue_frame' is called to enqueue the frame for eventual transmission in a packet. In this simplified description, we have omitted some additional bookkeeping related the total number of transmitted bytes on a connection and recording of finished streams.

We briefly consider the other frame types. The ACK frame acknowledges receipt of ranges of packet sequence numbers. The specification requires that each acknowledged sequence number has been used by the peer. There are frames that affect stream and connection state (such as CONNECTION_CLOSE, RST_STREAM and STOP_SENDING). Various flow control frames (MAX_DATA, MAX_STREAM_DATA, MAX_STEAM) affect the flow control state that is used in the specification of STREAM frames. CRYPTO frames are similar to STREAM frames but carry cryptographic handshake messages generated by the security layer. Other frames allow distribution of fresh CID's (to prevent tracking of mobile clients) and verification of network paths, among other functions. Each of these is specified in the same style as the STREAM frame above (and may include various updates of the protocol state). The standard states that all frames are *idempotent* which we take to mean that a frame may always be repeated, and repeating it has no effect on the protocol state. Our specification reflects this, in the sense that state updates can never disable a frame once it is enabled.

4.4 Security and application layers

The security layer has events corresponding to sending and receiving of cryptographic handshake messages and the establishment of encryption keys, which are used at the protection layer. The standard states that the TLS 1.3 protocol is used for the cryptographic handshake. We generate events at the security layer during testing by an actual *implementation* of TLS. This illustrates an important practical aspect of compositional testing. If some component of a system lacks a formal specification, or if specification-based generation for a component is intractable, we can treat it as a 'black box' (see [24] for details). There is some cost in generality of testing in this approach, but it allows us to circumscribe the formal specification task. For various technical reasons, we we also treat the encryption and decryption services, the operating system and the network itself as black boxes. There is one specification of TLS that we do rely upon however. That is, TLS allows user data to be encapsulated into handshake messages. QUIC uses this to exchange initial values of various transport parameters. We do formally specify the behavior of this interface. In the future, it may be possible to improve the generality of testing by generating TLS 1.3 behavior from a formal model, such as [14]. However, this must be done in such a way that it does not require inverting hash functions or other intractable operations that cannot be handled by SMT solvers.

At the application layer, there are a few simple actions related to requesting, establishing and closing of connections and streams, and to sending and receiving of stream data. The corresponding events at the API of a QUIC library may be visible or invisible, depending on the test set-up. When invisible (for example, because the QUIC library is linked to an actual application in the test) they can be easily inferred from the packet traffic.

4.5 The test shim

As in section 3, we wrote a test shim in Ivy that connects packet events in the specification to actual packets on the wire. This shim is complex because it includes packet encoding and decoding, that depend on QUIC's fairly elaborate encryption and decryption scheme.

The test shim also connects the security layer events to real instances of TLS. When (the real) TLS outputs a message, the shim translates it to a send event in the security layer of the specification. Correspondingly, when a receive event occurs in the security layer, the shim calls the API of TLS to deliver the received messages. The net effect is that the instances of TLS in our tester exchange messages with the TLS instances inside a QUIC implementation via the QUIC frame protocol.

Finally, events at higher layers of the protocol in the implementation under test are not visible to the tester. We designed the specification, however, so that these invisible events could be easily inferred. On observation of a packet, the shim scans it for indications of frame, security and application layer events, calling the appropriate actions. This is a small overhead, requiring only 28 lines of Ivy code.

5 RESULTS

We now consider some of the results of testing actual implementations of the evolving QUIC draft standard, using our evolving formal specification. We tested four implementations of QUIC, which were chosen because they had the best results in the interoperability testing matrix. One was only briefly tested early in the development of the specification and was abandoned due to TLS 1.3 compatibility issues. For each implementation, we tested a demonstration HTTP server based on a QUIC library, communicating with the server via UDP (testing clients is also possible in the framework). Somewhat arbitrarily, test runs were limited to 100 protocol events and a single QUIC connection. We also ran these tests in batches of 1000 on the same server instance, to

test its long-run behavior. In addition to specification violations, we recorded crashes and failures to make progress (as indicated by anomalously low transfer of data).

Testing revealed 27 errors in the server implementations, some of which reflect issues in the draft standard. We detail here only issues that we believe to have been fixed. Table 1 shows a breakdown of the issues found, categorized by our estimation of their root causes, based on developer feedback. These categories are: standard requirement not implemented, unexpected message order, unexpected parameter value, messages racing timers, arithmetic overflow, and code paths that had not been exercised for various reasons. A last category is 'unknown' reflecting errors which were fixed, but whose root cause was not communicated to us by the developers. For each category, the table gives the number of errors detected and the mode of detection: crashes (including internal assertion failures), specification violations, and lack of progress. We also give the number of errors that were deemed to be exploitable by an attacker, the number that were caused at least in part by an ambiguity or contradiction in the standard, and the number that were caused by an adverse stimulus — a message sequence that would occur only in unusual or adverse conditions and not in the expected flow of the protocol. We do not count crashes *per se* as exploitable, though some might be exploitable for denial of service. We describe some of the errors detected in further detail below.

Protocol errors. Of the 13 protocol errors we detected, five were due to a protocol requirement simply not being implemented. One server failed to respect a prohibition on sending a packet solely to acknowledge an ACK-only packet. Two failed to obey a rule that allows client migration to be detected only if a packet with the highest-observed sequence number arrives from a new address. One did not respect a rule on error codes in CONNECTION_CLOSE frames and one ignored STOP_SENDING frames. We attribute the fact that these omissions were not previously detected to the fact that previous testing had not been based on a thorough wire specification (though ad-hoc compliance tests were done).

Four violations were caused by unexpected messages or unusual parameter combinations. One server failed to open a stream on receipt of a frame other than a STREAM frame (a result of packet re-ordering). Another error resulted from an initial packet arriving after the connection was closed. One server failed to handle a STREAM frame without an offset field, and one failed to handle a MAX_STREAM_ID frame with an unexpected value. In a similar vein, a race between an arriving CONNECTION_CLOSE and a timer resulted in illegal sent messages.

One violation was due to an integer overflow in the value of the MAX_STREAM_DATA that would have caused a deadlock after transfer of 4GB on a stream. In two further cases, we discovered errors because randomized tests exercised code that was not well tested previously. On detecting migration, one server sent a PATH_CHALLENGE to the wrong address. One handled flow control incorrectly, resulting in a data leak described below.

Crashes. Twelve of the errors resulted in crashes, due either to memory faults or internal assertion failures. The most common cause of these was a frame arriving out of the expected order. All crashes were fixed, but we are aware of the root causes of only eight. In two cases a message arriving early caused the use of uninitialized data. In three cases an early-arriving message caused freeing of resources that were later used. In one case, an incoming message raced with a timer. In two cases, previously unexercised code was executed.

Ambiguities and contradictions. Four of the errors were caused at least in part by ambiguities or contradictions in the standard. For example, one server crashed because of a negative deadline given to a timer library. This resulted from the failure of pseudo-code in the loss recovery standard [18] to handle a case of unsigned integer underflow. This was later corrected in the standard. Another error (noted above) related to a state machine for stream senders in the standard that did not enumerate all the conditions that can open a stream. An ambiguity respecting the argument of MAX_STREAM_ID frames resulted in another possible violation (this was cleared up in version 17, unrelated to this work). In a possible contradiction, the standard stated that an APPLICATION_CLOSE packet must appear in every frame after the application closes the connection, but elsewhere it said that APPLICATION_CLOSE may not appear in initial frames. A violation of the latter condition resulted in a (minor) information leak in one server. These are cases where testing of implementations revealed information about the standard and helped to disambiguate it.

Vulnerabilities. In four cases, protocol or progress violations revealed as a side effect possible vulnerabilities in the implementations or the standard itself. One of these is a possible denial-of-service (DoS) attack by an off-path attacker. We discovered a trace in which a server ceased at some point to send any packets. Further analysis revealed that this was caused by a rapid switching of the client IP address between two values.

The specification states that when a client migration to a new IP address is detected, the server should validate the new network path by sending a PATH_CHALLENGE frame and waiting for a corresponding PATH_RESPONSE frame, indicating that the client in fact controls the new IP address. Before sending the challenge, however, the server waited for a timer to expire. Since the timer was reset each time

Table 1: Summary of errors detected

		Detection					
Root cause	Errors	Crash	Spec	Progress	Exploitable	Ambiguous	Adverse
Not implemented	5	0	5	0	0	0	2
Message order	7	5	2	0	1	3	7
Parameter	3	0	2	1	1	1	3
Race	3	1	1	1	1	0	3
Overflow	1	0	1	0	0	0	0
Unexercised	4	2	2	0	1	0	3
Unknown	4	4	0	0	0	0	?
Total	27	12	13	2	4	4	18/23

the client switched address, the challenge was never sent, resulting in a starvation scenario. This was detected only because the source IP address was randomized by the tester. In a non-adversarial test environment, this case would occur with negligible frequency.

This starvation scenario in turn suggested a vulnerability of the protocol itself to DoS. The attacker is assumed to be able to capture packets on the network and replay them from a different IP address in such a way that some fraction of the copies arrive before the originals. This causes the server to repeatedly detect migrations, and thus prevents any transfer of data. Mitigations for this off-path migration attack were included in a later version of the standard.

Another interesting case was a data leak similar to the "heartbleed" vulnerability discovered in SSL/TLS [12]. This was detected when a server sent incorrect bytes in a retransmission of a stream frame, violating the specification. The root cause was related to an error in the handling of flow control and the packing of frames in packets, which may have been exposed by adverse generation of flow control by the tester. The result was a STREAM frame that was longer than intended, causing arbitrary server memory contents to be leaked to the network. This was easily observed in the transcript of the test, as the ASCII text of an HTML page abruptly changed to binary data. The reason it was detected as protocol violation, however, is that the extra bytes did not match bytes sent in a previous stream frame at the given offset.

Another possible data leak was an APPLICATION_CLOSE frame sent in response to the unexpected initial packet mentioned above. This violated the protocol and also sent application data in the clear. Finally, a progress issue revealed a case in which a server allocated a number of records in memory proportional to the gap in packet sequence numbers, allowing an attacker to effectively stop the server. This issue was known but had not been repaired and was discovered because of a random choice of packet numbers.

Tester performance. As mentioned, the SMT-based tester generated protocol events at approximately 10Hz. Most of the errors we found occurred within a few minutes of testing on a single core. The most infrequently occurring error was the heartbleed-style data leak described above, which occurred roughly every 20,000 events (about 40 minutes of testing). The most time-consuming aspect of testing was triage of the errors, refining the specification, and communicating with the implementation developers to diagnose and repair the issues. To give a sense of the time scale, the total amount of test engineer time we spent on actual testing was approximately four weeks. The largest number of true errors discovered by a single test engineer in one day was three, with a typical value being one. Using cloud resources might produce further results as the rate of error discovery declines.

Analysis. Despite some limitations discussed in the next section, we found that the specification-based testing approach was effective in meeting the objectives we outlined. First, testing allowed us to significantly refine our formal specification. In many instances, we either loosened the specification (because of an unexpected message sent by an implementation) or tightened it (because an implementation flagged a protocol error due to a specification-generated message). Moreover, the specification-based tester was quite effective in uncovering implementation errors.

In the errors discovered we can observe two important advantages of the methodology. In 13 cases the errors were discovered because we monitor for compliance to a specification. Further, of the 23 to which we can assign root causes, 18 were found because of adverse stimulus (the remaining five were compliance violations occurring in normal protocol cases). Only two errors were previously known (but unfixed) issues and these had not to our knowledge been discovered by previous testing. This is a good indication that the approaches of interoperability testing and manual directed

or fuzz testing can be usefully augmented by specification-based methods.

As a point of comparison, interoperability testing of two QUIC implementations using symbolic execution [32] discovered two client API bugs and three crashes, based on manually created scenarios. No protocol violations were discovered, and only one error required more than a single message. A related approach applied to different protocols [29] considered only single-message errors. On the other hand, all of the bugs we found required a sequence of messages to be exchanged. This may indicate the difficulty of exploring deep protocol interactions with symbolic execution. Moreover, driving one implementation to provide a diverse stimulus for another implementation may be more difficult than using an abstract specification. More work is needed to answer these questions. Symbolic execution could in principle be combined with the present method and may improve coverage. We leave this for future work.

6 EXPERIENCES

In this section we consider some of difficulties we faced in applying the methodology, limitations of the current work, and lessons learned.

6.1 Completeness of the specification

We formally specified most features of the protocol, omitting 0-RTT data, version negotiation, retry messages, and one frame type (RETIRE_CONNECTION_ID). We focused on messages and features needed to produce basic protocol functionality or produced by the implementations. Overall, we found the effort required to create the specification and track changes through nine drafts to be manageable. The most time-consuming aspect was tracking changes in low-level formatting and encryption.

Our formal specification captures only safety properties of the protocol. These are properties whose violation can be detected by observing the system for finite time. An important class of properties that we do not cover is *liveness*. A liveness property requires that some condition must *eventually* hold, but do not specify a finite deadline. By their nature, such properties are difficult to test, as a message that is not observed may simply have been dropped or delayed. There is some work in this area [25] but detecting liveness violations by testing is heuristic at best. There are nonetheless many statements in the standard that can only be interpreted as liveness properties and we do not capture these. A good example is retransmission: our specification *allows* any data to be retransmitted, but does not *require* that lost data is eventually retransmitted.

Quantitative-time properties are another important class. We do not cover these because our tester does not run in real time. In particular, we do not cover any statement in the QUIC document on recovery and congestion control [18]. In principle, we could solve this problem by testing the implementations in virtual time. This would require creating a specialized test harness for each implementation.

Finally, we do not capture protocol error handling requirements and rather focus on the implementations' response to legal inputs. In principle, we could use the legal behaviors generated by our specification as seeds for generating illegal behaviors (*i.e.*, fuzzing) and use this to test a separate specification of error handling. We have left this for future work.

6.2 Extensional specification

We found that specifying extensional properties sufficed to produce a tester that successfully interacts with the real implementations. We also found that a large portion (perhaps most) of the statements in the standard were not usefully testable. Externally detectable violations of such statements occur only in rare cases, and these do not affect interoperation with the existing implementations. For example, in the QUIC standard there are cases when a received packet disables a given behavior. Since packets may be dropped internally (say, because the needed decryption keys have not yet been computed) we cannot definitely infer on observing the behavior that the protocol has been violated. Nonetheless, such statements may have heuristic value, for reasons of performance, obfuscation, or resistance to denial of service. A case in point is cryptographic randomness. This is crucial for security but not testable externally. Validating such specifications would require a different methodology (perhaps based on formal proof of refinement rather than testing). Our view is that it is important to state clearly and unambiguously the external view, separately from any advice on internal implementation. This is important for testing purposes, and also so that implementers clearly understand what is required for correct operation of the protocol.

6.3 Test coverage issues

Sampling randomly from *all* legal inputs tends to produce poor coverage of implementation behaviors. As an example, imagine generating IP addresses completely at random. The chance of generating the same address twice would be extremely low. Or, consider the CONNECTION_CLOSE frame in QUIC, indicating a protocol error. If this frame were generated too frequently, it would cancel all activity, and little of the implementation's behavior would be observed. In practice, we found it necessary to add constraints to our test shim designed to improve test coverage. This is done to increase or decrease the probability of certain events occurring in testing, and not to manage the tractability of constraint solving.

For example, we limit the client to use a bounded number of IP addresses, and to create a bounded number of streams. We experimented with restricting the tests to certain combinations of frames, and also adjusted the probability of certain frames. For example, to stress the servers, we tried test runs with many CONNECTION_CLOSE or RST_STREAM frames, which would be unusual in normal interoperability testing. In other tests, we set the probability of these frames to a low value, to allow effective testing of other frame types. In any test methodology it is important to adjust the test set to obtain good coverage. A constraint-based approach provides us with a powerful tool to do this. Standard metrics such as code and branch coverage could be used to guide this process.

A further test coverage issue is that the implementation behaviors cannot be fully covered without controlling their client API's and the underlying system API's. To do this would require a specialized test harness (instead of treating the application as a black box). We expect that many more errors could be found by this approach.

6.4 Performance of generation

We initially found the performance of random packet generation using an SMT solver to be far too slow for practical testing purposes. Analysis revealed that the primary cause of this was the encapsulation of large arrays of bytes into frames, and the encapsulation of many frames into packets. When compared to typical software or hardware interfaces, protocol messages have more complex structure and typically require more data transfer to exercise functionality. We solved this problem in two steps. First, we subdivided the problem by creating separate actions for events at the frame and security layers. Second, we eliminated the constraints involving large arrays of bytes (such as the content of the data and crypto streams) by factoring out the definitions of certain message fields. This can be seen in Fig. 2 line 17, where the packet payload is defined, and Fig. 3 line 5, where the STREAM frame data field is defined. We modified the Ivy tool so that these dependent fields are computed after solving for other parameters (see [24] for details). This separation also required some redundant state variables to store certain facts about the large arrays independently of the arrays themselves. These measures allowed us to generate events at a satisfactory rate (about 10Hz). However, they do require the specifier to pay careful attention to the form of the specification. It may be difficult in practice to determine why test generation is slow. A related problem is that test generation may deadlock because a requirement is unsatisfiable. This is also difficult to diagnose. We do not know if these methods will scale to more complex protocols that QUIC.

7 CONCLUSION

It is well recognized that Internet standards in the form of RFCs are ambiguous and difficult to interpret [5]. In this work, we tested an approach that uses light-weight formal methods to simultaneously achieve two goals: (1) distill an unambiguous wire specification from the knowledge implicit in implementations of QUIC and (2) test implementations against the wire specification in an adversarial manner.

The approach we used is compositional specification-based testing. The compositional aspect is crucial because it allows us to detect not only when the specification is too strong, but also when it is too weak. We found that the approach is effective on both fronts: it allowed us to extract knowledge from the implementations, and also to detect a significant number of errors in the implementations and standard.

To apply compositional testing to complex network protocols, we developed a methodology with two novel aspects:

(1) Allowing testable compositional specification of arbitrary collections of processes communicating over a network, as opposed to a fixed set of channels or interfaces.
(2) Exploiting layering of the protocol to decompose the test generation problem into tractable subproblems that can be handled by modern SMT solvers.

We are currently working on extending this methodology to address some of the difficulties we encountered, including the specification of real-time and liveness properties (especially related to loss detection, recovery and congestion control).

It is important to understand that our formal specification is not simply a formal expression of the informal statements in the RFC. The RFC primarily describes how the protocol is to be implemented internally, while our formal specification describes only what is visible externally. This is crucial in order to use the specification for testing and also as a clear and unambiguous distillation of the essence of the protocol. The two forms of specification are complementary and both are necessary.

As we noted, the SSL/TLS ecosystem suffered many difficulties owing to the lack of compliance of implementations in the wild to an unambiguous common protocol specification. Our hope is that providing such a specification in a *testable* form will be a step in preventing such difficulties in QUIC. We also hope that the form of the specification is simple enough that future developers of QUIC can use it as a reference, though this remains to be seen. This consideration in part motivates the form of the specification as an executable monitor. In general, we see this work as a step in the process of integrating formal specifications as a *complement* to other approaches in Internet standardization.

REFERENCES

[1] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 98–109.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885.

[3] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 535–552. https://doi.org/10.1109/SP.2015.39

[4] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE, 445–459.

[5] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. 2018. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *JACM* 1, 66 (12 2018), 1–77.

[6] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*. ACM, 234–245.

[7] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. 2018. A Formal TLS Handshake Model in LNT. In *Proceedings Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation, MARS/VPT@ETAPS 2018, and Sixth International Workshop on Verification and Program Transformation, Thessaloniki, Greece, 20th April 2018*. To be published in EPCT, 1–40.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 209–224.

[9] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. 2010. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 426–439.

[10] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification - 30th International Conference Part II*, Vol. 10982. Springer, 430–446.

[11] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. 2002. Testing and Tracing Lazy Functional Programs Using QuickCheck and Hat. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, Vol. 2638. Springer, 59–99.

[12] The Mitre Corporation. 2014. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[13] The Mitre Corporation. 2014. CVE-2014-3566. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566.

[14] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1773–1788. https://doi.org/10.1145/3133956.3134063

[15] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

[16] Dimitra Giannakopoulou, Corina S. Pasareanu, and Colin Blundell. 2008. Assume-guarantee testing for software components. *IET Software* 2, 6 (2008), 547–562.

[17] Internet-Draft. 2019. QUIC: A UDP-Based Multiplexed and Secure Transport (Version 18). https://tools.ietf.org/id/draft-ietf-quic-transport-18.

[18] Internet-Draft. 2019. QUIC: QUIC Loss Detection and Congestion Control. https://tools.ietf.org/html/draft-ietf-quic-recovery-18.

[19] James E. Johnson, David E. Langworthy, Leslie Lamport, and Friedrich H. Vogt. 2007. Formal specification of a Web services protocol. *J. Log. Algebr. Program.* 70, 1 (2007), 34–52.

[20] Hyojeong Lee, Jeff Seibert, Dylan Fistrovic, Charles Edwin Killian, and Cristina Nita-Rotaru. 2015. Gatling: Automatic Performance Attack Discovery in Large-Scale Distributed Systems. *ACM Trans. Inf. Syst. Secur.* 17, 4 (2015), 13:1–13:34. https://doi.org/10.1145/2714565

[21] Kenneth L. McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. IEEE, 109–116.

[22] Kenneth L. McMillan. 2019. Mechanized Specification of QUIC. https://github.com/microsoft/ivy/tree/master/doc/examples/quic.

[23] Kenneth L. McMillan. Last updated 2019. Ivy. http://microsoft.github.io/ivy.

[24] K. L. McMillan and L. D. Zuck. 2019. Compositional Testing of Network Protocols. http://mcmil.net/pubs/SECDEV19.pdf. In *IEEE Secure Development Conference (SecDev 2019)*. To appear.

[25] Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. 2017. Lasso detection using partial-state caching. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 84–91. https://doi.org/10.23919/FMCAD.2017.8102245

[26] B. Neelakantan and S. V.Raghavan. 1995. Protocol Conformance Testing – A Survey. In *Computer Networks, Architecture and Applications*, S. V. Raghavan et al. (Ed.). Springer, Chapter 1, 175–191.

[27] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 614–630.

[28] Javier Paris and Thomas Arts. 2009. Automatic testing of TCP/IP implementations using QuickCheck. In *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang, Edinburgh, Scotland, UK, September 5, 2009*. ACM, 83–92.

[29] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. Analyzing Protocol Implementations for Interoperability. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 485–498. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa

[30] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. 2015. Protocol State Machines and Session Languages: Specification, implementation, and Security Flaws. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE, 125–133.

https://doi.org/10.1109/SPW.2015.32

[31] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. 2019. State machine inference of QUIC. *CoRR* abs/1903.04384 (2019). arXiv:1903.04384 http://arxiv.org/abs/1903.04384

[32] Felix Rath, Daniel Schemmel, and Klaus Wehrle. 2018. Interoperability-Guided Testing of QUIC Implementations using Symbolic Execution. In *Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ 2018)*. ACM, 15–21.

[33] Ivan Ristic. 2014. POODLE Bites TLS. https://blog.qualys.com/ssllabs/2014/12/08/poodle-bites-tls.

[34] Jan Rüth. 2018. How much of the Internet is using QUIC? https://blog.apnic.net/2018/05/15/how-much-of-the-internet-is-using-quic/.

[35] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. 2008. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Lecture Notes in Computer Science, Vol. 4949. Springer Verlag, 39–76.

[36] Gregor von Bochmann. 1989. Protocol Specification for OSI. *Computer Networks and ISDN Systems* 18, 3 (1989), 167–184.

[37] QUIC working group. 2019. QUIC Working Group. https://quicwg.org/.

[38] Pamela Zave. 2015. How to Make Chord Correct (Using a Stable Base). *CoRR* abs/1502.06461 (2015). http://arxiv.org/abs/1502.06461