# Monitor-based Testing of Network Protocol Implementations Using Symbolic Execution


Hooman Asadian
hooman.asadian@it.uu.se
Uppsala University
Uppsala, Sweden

Paul Fiterău-Broştean
paul.fiterau_brostean@it.uu.se
Uppsala University
Uppsala, Sweden

Bengt Jonsson
bengt@it.uu.se
Uppsala University
Uppsala, Sweden

Konstantinos Sagonas
kostis@it.uu.se
Uppsala University
Uppsala, Sweden
National Technical University of Athens
Athens, Greece



## ABSTRACT

Implementations of network protocols must conform to their specifications in order to avoid security vulnerabilities and interoperability issues. To detect errors, testing must investigate an implementation's response to a wide range of inputs, including those that could be supplied by an attacker. This can be achieved by symbolic execution, but its application in testing network protocol implementations has so far been limited. One difficulty when testing such implementations is that the inputs and requirements for processing a packet depend on the sequence of previous packets. We present a novel technique to encode protocol requirements by *monitors*, and then employ symbolic execution to detect violations of these requirements in protocol implementations. A monitor is a component external to the SUT, that observes a sequence of packets exchanged between protocol parties, maintains information about the state of the interaction, and can thereby detect requirement violations. Using monitors, requirements for stateful network protocols can be tested with a wide variety of inputs, without intrusive modifications in the source code of the SUT. We have applied our technique on the most recent versions of several widely-used DTLS and QUIC protocol implementations, and have been able to detect twenty two previously unknown bugs in them, twenty one of which have already been fixed and the remaining one has been confirmed.



## CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software defect analysis**.

## KEYWORDS

Software security, security testing, network security, network protocols, symbolic execution, monitors, DTLS, QUIC





*ARES 2024, July 30–August 02, 2024, Vienna, Austria*



ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3664521

**ACM Reference Format:**
Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. 2024. Monitor-based Testing of Network Protocol Implementations Using Symbolic Execution. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30–August 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3664476.3664521


## 1 INTRODUCTION

Implementations of network protocols, such as TCP, TLS, DTLS, QUIC, etc., must conform to their specifications in order to avoid security vulnerabilities and interoperability issues. Even seemingly innocent deviations from the standard may open implementations up for security attacks. Examples include Heartbleed [10] and the TLS POODLE downgrade vulnerability [7, 22], enabled by insufficient checking of length fields or version numbers in input packets. In order to detect such errors, protocol testing techniques must be able to investigate an implementation's response to a wide range of inputs, not only inputs that are expected from protocol peers, but also inputs that could be supplied by malicious attackers.

A technique that explores an implementation's response to the full range of possible inputs is *symbolic execution (SE)* [19]. It analyzes programs for which some inputs are designated as *symbolic*, and explores the code paths that are possible for some values of these symbolic inputs, thereby implicitly considering all possible values of these inputs. In recent years, SE tools such as KLEE [8] and $S^2E$ [15] have demonstrated impressive results for testing programs, such as systems libraries, device drivers, and computer vision code, being able to find safety violations that have been difficult to expose using other techniques [9].

For implementations of network protocols and other programs that repeatedly interact with their environment, SE faces several difficulties. One of them is that most network protocols are *stateful*: to test the processing of a particular packet, the implementation must first be brought to a specific state by an appropriate packet sequence. Moreover, the requirements concerning that packet may depend on the contents of packets in the previously exchanged packet sequence. To illustrate, consider a requirement stating that if the sequence number field in any packet has the same value as the sequence number field of any previously received packet,

then an error packet must be output. An idea for an SE-based approach to check that a protocol implementation satisfies such a requirement for any sequence of input packets could be to devise techniques for designating the sequence number field in each input packet as symbolic (thereby implicitly considering any value in these fields), and for checking the uniqueness requirement on these fields by inserting instrumentation and assertions into the code of the SUT. Such techniques could be applied to a given SUT by appropriate modifications in its code, but such modifications need be re-applied when testing different implementations of a given protocol or different versions of an implementation.

**Challenge:** To thoroughly test several implementations of a protocol, we need a testing technique that can check requirements over sequences of exchanged packets, with the properties that it

① covers a wide range of inputs, including all potentially requirement-violating values of fields in sequences of packets,
② can automatically be applied to any implementation of a protocol, without requiring modification in a SUT's code.

**Current state of the art:** Unfortunately, existing SE techniques for testing protocol implementations do not achieve both ① and ②. Several combinations of SE and requirement checking (RC) have been presented, but they apply SE and RC separately. For instance, SymbexNet [32] tests protocol implementations by first employing SE to generate test inputs that exercise a wide range of code paths; these inputs are thereafter applied to the SUT to generate concrete executions that are subject to RC. This approach does not satisfy property ①, since its use of SE only explores code paths that can be exercised in a non-adversary environment, and furthermore considers only one test input for each such path. Consequently, the approach may miss requirement violations that are exposed by specific, possibly adversarial, inputs. Tempel et al. [34] show that SE adapted for protocols increases coverage, but they do not check requirements. Rath et al. [26] use SE to detect interoperability issues by provoking different failure scenarios, and Pedrosa et al. [24] use SE to check for mismatches in message communications. A combination of SE and RC which satisfies property ① has been presented by our group [3], however at the cost of subjecting each SUT to significant manual source code modifications (including to make it read input from a file, and injecting assertions) which are SUT-specific and not easily portable to its next version(s). Thus, we are still lacking a technique for testing stateful protocol implementations which achieves properties ① and ②.

**This paper:** We present a technique, based on symbolic execution, for testing network protocol implementations, which achieves both properties. The technique operates in a test environment which includes the parties of the protocol, and applies symbolic execution to check requirements against a protocol's implementation. Rather than relying on intrusive code modification in the SUT for checking requirements, our technique is based on concentrating the logic for checking each requirement in a software component which is external to the SUT; for this, we have developed a novel adaptation of the concept of *monitors* (e.g., [31]) for symbolic execution. During an interaction, a monitor observes the packets exchanged between the SUT and its peers to maintain relevant up-to-date information about the current state of the interaction. Using this information, it can check whether packets sent by the SUT conform to protocol requirements. To cover all potentially requirement-violating values in input fields, the monitor designates relevant fields in input packets as symbolic. A symbolic execution engine will then explore all code paths that can be reached for any values (including those that may induce requirement violations) of these fields. In addition, the SE engine will also detect memory errors, crashes, and other runtime problems. By being external to the SUT, monitors provide a uniform mechanism for checking requirements and detecting runtime errors, which must be implemented *only once* for each protocol, and can be automatically applied to any implementation of the protocol, *without* requiring modification in a SUT's code. In some cases, the effectiveness of our technique can be increased by small modifications in the SUT's code, e.g., to disable encryption (which is problematic for SE to handle).

We have implemented our technique by creating a generic infrastructure for SE-based testing of network protocol implementations which exchange packets over sockets. To evaluate the effectiveness of our technique and directly compare it with the state-of-the-art technique of our previous work, which is also based on requirement-driven symbolic execution [3], we have first applied it on four widely-used DTLS implementations. The technique we describe in this paper has reproduced all bugs detected by the other technique, but with less implementation effort and considerably faster. It has also detected several, previously unknown bugs, in newer versions of two of these DTLS implementations. We have also applied the technique of this paper to three implementations of the QUIC protocol, and have detected previously unknown bugs in all of them. We have reported these bugs to the developers, and the vast majority of them have already been fixed.

**Contributions:** In short, the contributions of this paper are:

- A new technique to test protocol implementations against their requirements using symbolic execution and monitors which are external to the SUTs and, as such, can be directly reused by different versions and implementations of a particular protocol.
- An evaluation showing the technique's bug finding effectiveness, time performance and implementation effort when compared with another state-of-the-art SE technique.
- A publicly available infrastructure [5] with test harnesses and sets of monitors for two widely used protocols (DTLS and QUIC) to repeat our experiments or extend our work with more monitors for these or other network protocols.

**Overview:** The next section introduces the concept of monitors and their adaptation for SE, followed by §3 that presents the steps of our technique. Section 4 overviews the two protocols that we have applied our technique on, and presents representative monitors for their requirements. We then describe our implementation (§5), and evaluate its effectiveness (§6). The paper ends with discussing threats to validity (§7), related work (§8) and some final remarks.

## 2 MONITORS BY EXAMPLE

In this section, we present our adaptation of the concept of *monitors* [31] for symbolic execution of protocol implementations. We first explain monitors and, thereafter, our adaptation. of monitors for symbolic execution.

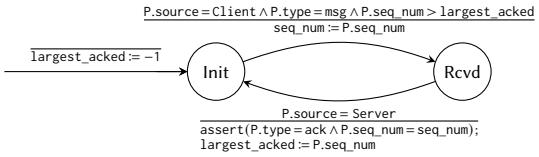


**Figure 1: A monitor for Req1.**

A monitor is a software component which is external to the parties of a protocol, and executes concurrently with them. It observes the sequence of packets exchanged between protocol parties and maintains information about relevant aspects of the current state of the protocol interaction; this information can be updated upon observing each new packet. Using this information, the monitor can detect whenever a packet sent by the SUT violates a protocol requirement, in which case it signals this violation.

As an illustration, consider a protocol for exchanging packets between a client and server, in which there are two types of packets: messages and acknowledgments, both of which contain a sequence number as a field. Each party strictly alternates between sending and receiving, i.e., packets are ping-ponged between the client and server. Suppose that a requirement for the protocol is that:

> ***(Req1)*** *When the server receives a message with a new sequence number (i.e., greater than the largest acknowledged sequence number so far), it MUST respond with an acknowledgment containing the same sequence number as the received message.*

A monitor for this requirement is shown in Fig. 1. For monitors, we find it convenient to use the format of state machines, in which their local state consists of a *control state*, which ranges over one of the shown circles, and additional local variables. The monitor in Fig. 1 maintains two local variables:

- `largest_acked` contains the largest sequence number acknowledged by the server;
- `seq_num` contains the sequence number of the most recent message with a new sequence number, sent by the client; it is used to determine whether the acknowledgment by the server has the required sequence number.

The monitor observes packets sent by the server or the client. It can perform a transition whenever a packet satisfies the condition above the bar in the label of an outgoing transition. We let `P` denote the observed packet, and let `P.source` denote the origin of `P`. When performing the transition, the monitor executes the statement below the bar. Thus, starting in state Init, if the monitor in Fig. 1 observes a packet from the client, which is of type message and whose sequence number is greater than `largest_acked`, it stores the packet's sequence number in variable `seq_num`, and transitions to state Rcvd. In state Rcvd, when the monitor observes a packet from the server, it checks using an `assert` statement that it is of type acknowledgment and that its sequence number is equal to that of the last message from the client (which was stored in `seq_num`). If that is the case, the monitor updates `largest_acked` and transitions back to Init. If the `assert` statement fails, then the execution will abort with an error.

The monitor in Fig. 1 can be used to check the requirement during testing. Alas, our monitor checks the server's response only to messages that are actually sent by the client during a testing session. It would be desirable to check the server's response to any sequence number for which the requirement is relevant, but generating an input message for each such sequence number is infeasible. However, we can leverage the power of symbolic execution to consider the range of all relevant sequence numbers. This is done by making selected input data *symbolic*. A symbolic execution engine will then explore all code paths that are reachable for some value of the symbolic input. In our example, we can test the server's response to any relevant value of the `seq_num` field by modifying the monitor in two ways:

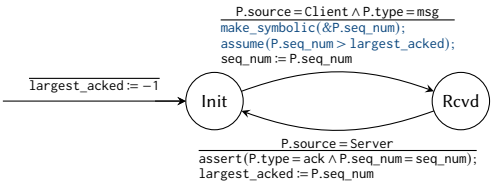


**Figure 2: Monitor for Req1, adapted for symbolic execution.**

(1) Making the `seq_num` field in the message from the client symbolic.
(2) Adding an assumption that the value in this `seq_num` field is larger than the previously largest acknowledged sequence number. This makes symbolic execution explore only code paths that are reachable on a new value of this field. In general, such assumptions can be used to restrict the considered values of symbolic fields to those for which a requirement violation is possible.

The bottom transition, where the server's response is checked, need not be modified. The result is shown in Fig. 2, with the additions marked in blue. A symbolic execution engine will then explore all code paths that can be reached for a value of the `seq_num` field, which is larger than the current value of `largest_acked`. If there is such a code path at which the `assert` statement on the bottom transition can be triggered, then the symbolic execution engine will stop executing this code path, and return a concrete value for the `seq_num` field which induces the assertion failure.

The monitor in Fig. 2 checks that packets from the server satisfy a constraint that depends only on the preceding packet. However, such constraints may also depend on several preceding packets. As an illustration, consider the following requirement.

> ***(Req2)*** *For each received packet of type `msg`, the server MUST verify that its sequence number does not duplicate the sequence number of any other packet of type `msg` received during the life of this session. The server MUST discard packets with duplicate sequence numbers and may generate an Alert packet.*

A monitor for this requirement is shown in Fig. 3. The monitor maintains the local variable `S`, which contains the sequence numbers of all packets sent by the client so far. Starting in state Init with `S` initialized to the empty set, when the monitor observes an incoming packet of type `msg` from the client, it can either (i) take the self-loop transition and add its sequence number to the set `S`, or (ii) take the transition to Rcvd and add the assumption that its sequence number is a duplicate (i.e., included in `S`), and thereafter check by an `assert` statement that the next packet from the server is of type Alert. If there is a code path for which the server responds

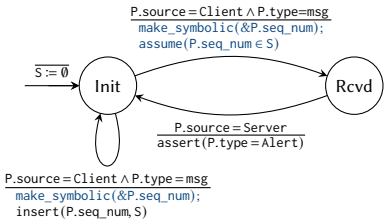


**Figure 3: Monitor for Req2, adapted for symbolic execution.**

with a packet other than an Alert, the assertion on the bottom transition is triggered. In this monitor, the choice between the two outgoing transitions from state Init is nondeterministic. This allows the monitor to make the transition to Rcvd on any packet from the client, assume it to have a duplicate sequence number, and check the server's response. The overall effect is to consider, given a sequence of packets from the client, all sequence number fields that exhibit a duplicate sequence number.

# 3 METHODOLOGY

Having introduced the concept of monitors, in this section we present a general methodology for using a set of such monitors to detect requirement violations in protocol implementations. Its steps are:

- creating a test harness to execute the SUT;
- encoding monitors as programs;
- invoking monitors at appropriate moments during SE;
- (optional) developing a protocol-specific parser/serializer;
- (optional) applying small modifications to the SUT.

We note that some of these steps are somewhat constrained by the SE engine which is employed. In §5, we will describe our implementation and infrastructure we have built for the two protocols to which we have applied our methodology.

## 3.1 Creating a Test Harness

The first step is to construct a test harness in which the protocol parties execute, and which invokes a monitor whenever a packet is received by one of the parties. Such a test harness will then be executed symbolically. To minimize the effort spent, our methodology advocates creating a single uniform harness that can be equipped with an arbitrary monitor which tests one or several protocol requirements regardless of which protocol party they refer to. Depending on the monitor, the protocol interaction that the test harness performs ranges from a simple message exchange (in stateless protocols) to the parties completing a handshake and then exchanging some application data. When using KLEE as SE engine, the harness must consolidate all involved protocol parties in a single-threaded (Unix) process, and needs to coordinate interactions between them. Our experience is that this is possible for all protocol implementations we have tried so far, since protocol implementations (i.e., our SUTs) often come in the form of libraries that offer an API. Harnesses can utilize such an API to coordinate an interaction between protocol parties (e.g., between a client and a server) by letting parties take turns; in each turn a party processes the packets that have been sent to it so far, thereby producing packets for the other parties.

## 3.2 Encoding Monitors as Programs

Once specified, perhaps using a graphical notation similar to that of §2, the set of monitors needs to be translated into code. When using KLEE as SE engine, one is constrained to use a programming language that can be compiled to LLVM bitcode. Our implementation uses C. The translation of a monitor's state machine is roughly as follows:

- Control states are encoded using an enumeration (`enum`).
- The state of the state machine is kept in a data structure that records its current state and any associated local variables.
- State machine transitions are implemented using conditionals (`if-else` statements). In these `if-else`s, the conditions above a bar specify the condition to check, while the statements below the bar are the actions to perform before moving on to the target state of the transition.
- Whenever a choice between outgoing transitions is nondeterministic, we introduce a symbolic variable whose value ranges over the available alternatives, and determines which one is chosen. On encountering such a variable, the SE engine forks, effectively covering all possible outcomes of nondeterministic choices in the monitor.

## 3.3 Invoking Monitors

During testing of a protocol requirement, the SE engine must be able to invoke the monitor at appropriate points during the interaction that the harness involves. Since the monitor observes and possibly manipulates the packets exchanged during this interaction, it should be invoked when a packet is received and before it is processed by a protocol party. This can be realized by implementing the monitor as a callback function, which is called just before delivering the packet to the protocol party, with the data in the received packet passed as a parameter.

## 3.4 Developing Protocol-specific Infrastructure

In many protocol implementations, packets are exchanged as arrays of bytes. When a monitor is invoked immediately after a packet is received by a protocol entity, it perceives the packet as an array of bytes. Encoding monitors as programs becomes significantly easier if monitors receive packets as parsed objects instead. This can be done by implementing a parser for the protocol that the SUT implements (or reusing such a parser if it exists). This parser can be invoked just prior to calling the monitor. In this setup, instead of directly invoking the callback for the monitor, the callback triggers a protocol-specific infrastructure that performs the following operations:

- The received data is parsed into a structured data object.
- The monitor is called with this object as its parameter.
- After the monitor has completed its modifications to this object, a serializer converts the object back into a byte array format and passes it for processing to the harness.

This optional step streamlines the development of monitors.

## 3.5 Applying Small Modifications to the SUT

For certain SUTs, it can be advantageous to make small modifications to their code in order to streamline the testing process

using symbolic execution. Specifically, when a protocol involves the transmission of packets between parties in encrypted form, one may want to disable encryption. This not only avoids the encryption/decryption overhead, but also accelerates symbolic execution by removing the need to handle cryptographic functions. Another optional adjustment involves timing-based behavior. Some protocols initiate re-transmission or prematurely terminate an interaction if no response is received within a specified time frame. In most cases, this modification can be achieved by setting the timeout value to its maximum allowable value. Another option is to completely disable timeouts.

# 4 MONITORS FOR DTLS AND QUIC REQUIREMENTS

We have applied our methodology to test implementations of DTLS and QUIC, two widely-used complex security protocols. This section briefly describes these protocols and, for each of them, introduces two representative monitors for their requirements.

## 4.1 Datagram Transport Layer Security

DTLS is a client-server protocol based on TLS that is used to secure communication over datagram transport layer protocols such as UDP. Its lower overhead and reduced latency compared to TLS makes it suitable for resource-constrained environments. Our work is concerned with DTLS version 1.2 [27]. The DTLS protocol is structured in layers. The Record Layer lies at the base. It encrypts *messages* received from upper layers, encapsulates them into *records*, and sends them over the network. Dually, it receives records from the network, decapsulates and decrypts them, producing messages which it delivers to the appropriate upper layer. The Handshake Layer is engaged at the start of each DTLS session, and establishes encryption keys for the Record Layer, by performing a *DTLS handshake*. A handshake involves a complex exchange of messages between the client and server. As part of this exchange, the two parties agree on a key exchange algorithm, and use this algorithm to establish session keys, which are then deployed. The Handshake Layer delivers handshake messages to the Record Layer as fragments that can fit in a datagram. Conversely, the Handshake Layer assembles handshake messages from fragments received from the Record Layer.

Let us present two requirements for the DTLS protocol and their encoding using appropriate monitors.

*4.1.1 Matching Content Type.* According to the TLS RFC [16, p. 19], the content type in the Record Layer *"determines the higher-level protocol used to process the enclosed fragment."* The same RFC [16, p. 19] defines `ContentType` as a one-byte entity representing a set with four values, one of which is `alert`. Based on this, we can check whether the type of the content matches the enclosed fragment. In order to construct a monitor for checking that the server correctly responds to records with invalid `content_type` fields, we consult the DTLS RFC [27, p. 14], which prescribes a general rule for dealing with invalid records (e.g., records with invalid `content_type` values):

> *In general, invalid records SHOULD be silently discarded ... Implementations which choose to generate an alert instead, MUST generate fatal level alerts.*

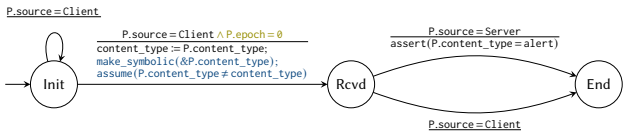


**Figure 4: Monitor for the Matching Content Type requirement of DTLS.**

We can now construct the monitor in Fig. 4. Let us analyze it, initially ignoring the optional conjunct colored olive. The monitor checks a server's reaction to records whose content type mismatches their enclosed fragment. When in state Init and observing a client-generated record, the monitor can nondeterministically choose to either (i) take the self-loop or (ii) take the transition to state Rcvd, where the `content_type` field in the record is made symbolic and assumed to be different from its original concrete value. The transition to Rcvd in effect transforms the record sent by the client, so that its `content_type` field is invalid, i.e., inconsistent with its payload. According to the RFC, the server must either discard an invalid record and continue the interaction or respond with an *Alert* message and terminate the connection. Thus, when in state Rcvd, if the monitor receives a packet from the server, it checks that the packet is of type `alert` and transitions to state End. If, instead, the monitor gets a packet from the client, it goes directly to state End, because in this case the server may have silently discarded the invalid record previously sent by the client.

Now let us consider the case where we enable the optional conjunct. Doing so makes the monitor invalidate only client-generated records whose `epoch` field is zero. In DTLS, such records enclose unencrypted fragments, whereas records with nonzero `epoch` fields enclose encrypted fragments. Although the `content_type` field is not encrypted, it is used in subsequent cryptographic functions by the server whenever `epoch` is nonzero. If we do not want to expose symbolic execution to cryptographic functions, we can choose to check this requirement only for records whose `epoch` field is zero.

*4.1.2 Invalid Epoch.* Records are equipped with the integer-typed field `epoch` which uniquely identifies the keys used to encrypt the record fragment. The DTLS RFC [27, p. 9] imposes the following requirement on `epoch`:

> *The* `epoch` *number is initially zero and is incremented each time a* ChangeCipherSpec *message is sent.*

Based on this requirement and the rule for dealing with invalid records, Fig. 5 shows a monitor which checks that the server appropriately responds to records with invalid `epoch` values, by either discarding them or by generating an *Alert* to report an error. This monitor uses the variable `counter` to keep track of the number of *ChangeCipherSpec* messages received from the client. In the initial

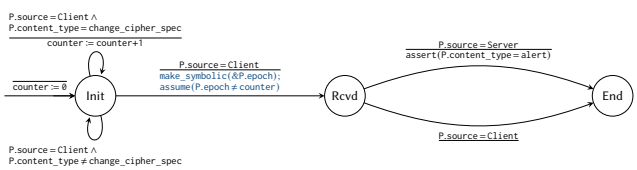


**Figure 5: Monitor for the Invalid Epoch requirement of DTLS.**

state, upon observing a client record, the monitor can either (i) take the self-loop transition enabled by the record type (indicated by `P.content_type`) and update `counter` accordingly, or (ii) take the transition to state Rcvd, making the `epoch` field symbolic and assuming its value to be invalid. Once in state Rcvd, the monitor checks that if the next record is received from the server, its content type is `alert`.

## 4.2 QUIC

QUIC [18] is a client-server transport layer protocol that provides secure and reliable stream-based communication. Operating over UDP, QUIC supports many of the services also provided by TCP, TLS and HTTP, but does so in a single protocol that is optimized to reduce latency. QUIC parties communicate by exchanging *QUIC packets*, with each packet containing a sequence of *frames*. Like TCP, QUIC is connection-based. To establish a connection, the parties perform a handshake in which they agree on encryption keys and the application protocol to invoke once the handshake is done.

*4.2.1 Unacceptable Version.* QUIC packets include an integer field that indicates the protocol's version. Through a version negotiation process, a server notifies a client that the version selected by the client is not supported. Regarding this process, the QUIC RFC [18, §6.1 and §17.2.1] requires that:

> *If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation (VN) packet.*
> ⋮
> *A Version Negotiation packet is inherently not version specific. Upon receipt by a client, it will be identified as a VN packet based on the Version field having a value of 0.*

This requirement is captured as a monitor in Fig. 6. We assume that the acceptable versions of QUIC are captured in the predicate `is_acceptable`. When in state Init and observing a client-generated

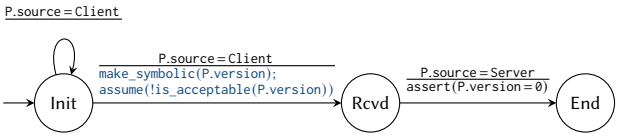


**Figure 6: Monitor for the Unacceptable Version requirement of QUIC.**

packet, the monitor can nondeterministically choose either to take the self-loop and do nothing, or take the transition to state Rcvd, during which the `version` field in the packet is made symbolic and assumed to be not acceptable. At this point, the only valid response from the server is a version (re)negotiation packet. While in the Rcvd state, the monitor reviews if the server responded with such a packet by checking that the `version` field in the next packet sent by the server equals to zero. If this is not the case, the symbolic execution engine will abort this code path and generate a concrete value for the `version` field in the packet sent by the client.

*4.2.2 Invalid New Connection ID Length.* With every QUIC connection is associated a set of connection IDs that ensure that packets for QUIC connections are not delivered to the wrong endpoint due to changes in addressing at lower protocol layers. Each connection possesses multiple connection IDs to prevent potentially malicious observers from correlating activities on multiple paths. Endpoints provide alternative connection IDs to their peers by sending frames of type `NEW_CONNECTION_ID`. Each such frame contains a `length` field, giving the length of the transmitted connection ID. For this field, the QUIC RFC [18, §19.15] states that:

> *NEW_CONNECTION_ID frames contain the following fields:*
> ⋮
> *Length: An 8-bit unsigned integer containing the length of the connection ID. Values less than 1 and greater than 20 are invalid and MUST be treated as a connection error of type FRAME_ENCODING_ERROR.*

This requirement is captured as a monitor in Fig. 7. Since each packet may contain an arbitrary number of frames, and since several of these may be of type `NEW_CONNECTION_ID`, the notation `P.frame` for accessing a frame within a packet is not well-defined. We rather want to refer to *some* frame of type `NEW_CONNECTION_ID` within a packet. For this purpose, transitions can be *parameterized* by a local variable, which ranges over a finite set. For this requirement,

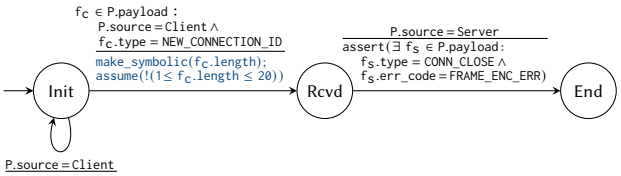


**Figure 7: Monitor for the Invalid New Connection ID Length QUIC requirement.**

the monitor contains a parameterized transition from state Init to state Rcvd, parameterized by the variable $f_C$, which ranges over frames in the observed packet. Such a parameterized transition represents a transition for each value of the parameter. Thus, the monitor in Fig. 7 can perform a transition from Init to Rcvd for each frame in the payload of the current packet, which is of type `NEW_CONNECTION_ID`. When performing the transition, the monitor will make the `length` field of this frame symbolic and assume that its value is invalid.

The check that an invalid `length` field is treated as a connection error is performed by the transition from Rcvd. The monitor of Fig. 7 checks that the next packet from the server contains a frame of type `CONNECTION_CLOSE` with error code `FRAME_ENCODING_ERROR`. Also here, we must take into consideration that a packet may contain several frames, and check that one of them reports the error. Such a check must be performed by a single transition, which checks all frames in the next packet from the server. This transition is labeled by an `assert` statement with an existential quantification over the frames in the payload of the observed packet.

# 5 IMPLEMENTATION

In this section, we describe our implementation. At its core one finds a generic infrastructure (§5.1) which suffices for all network protocols exchanging packets over POSIX sockets [33]. It mediates communication between protocol entities, and is responsible for "executing" the monitor on the packets exchanged between them. Besides the test harness, it needs to be supplied with the set of

monitors for the protocol's requirements and, optionally, a parser and a serializer (§5.2). With these components supplied and optional small changes to the SUT (§5.3), a protocol implementation can be tested using a symbolic execution engine (§5.4).

## 5.1 Generic Infrastructure

Many protocols exchange packets over POSIX sockets using an API provided by an OS library [33]. Our generic infrastructure is used as a proxy to this library and allows it to mediate, by redirecting API calls, the interaction between the SUT and its peers. For instance, library functions that send a packet to a socket (e.g., `sendto` in POSIX sockets) are replaced by functions that put the packet in a shared queue. Receive library functions (e.g., `recvfrom` in POSIX sockets) are replaced by functions that obtain the first packet in the shared queue, and use it to "execute" the monitor for a requirement of interest by performing the following sequence of steps:

(1) The first packet in the queue is parsed, using a protocol-specific parser (§5.2), and copied to a data structure $P$.
(2) The monitor is invoked with $P$ given as an argument.
(3) Once (the function implementing) the monitor returns, $P$ is serialized back and returned to the calling entity. Note that the returned $P$ may have some fields in the original $P$ manipulated or made symbolic.

A separate packet queue is used for each direction of the communication (e.g., from client to server, and from server to client). The `bind` function is redirected to a function which initializes packet queues. All other functions are left unchanged.

## 5.2 Protocol-Specific Infrastructure

Our implementation also comprises the following protocol-specific components.

*Sets of Monitors.* Monitors are encoded as C functions using the translation described in §3.2. These functions take a pointer to the data structure as a parameter and access packet fields by selecting the corresponding fields in this structure. Monitors have persistent states implemented by static variables. For example, the monitor in Fig. 5 maintains two such variables, one for storing the current control state of the monitor, and the other for the `counter` variable.

Figure 8 contains the C code implementing the monitor for the Matching Content Type DTLS requirement for servers (Fig. 4). The code first declares an enumeration for the monitor's states and a static variable for storing the current state of the monitor (lines 1 and 3). The function implementing the monitor takes as parameter a pointer to the data structure storing the currently observed packet. Its body is quite straightforward and closely follows the preconditions and actions of Fig. 4. The `kleener_make_symbolic()` function is our wrapper around KLEE's `klee_make_symbolic()` that also allows for treating only *part* of a buffer as symbolic. Its first argument is a pointer to the relevant packet field, the second is its size, and the third is a name. The only aspect of the monitor of Fig. 4 which is not included in the code of Fig. 8 is the self-loop in its Init state. This is handled by infrastructure that supports nondeterminism between two transitions, one of which is a self-loop in the initial control state. It does so by assigning to each packet `P` an index (i.e., the $i$th packet in the interaction has index $i$), and declaring a symbolic variable `trigger_index` ranging over possible values of this index. The infrastructure then allows the non-self-loop transition to be taken if its guard is true *and* the index of the current packet is equal to `trigger_index`, otherwise it allows the self-loop to be taken. Since `trigger_index` is symbolic, the SE engine will explore all feasible code paths in which the monitor performs an arbitrary number of self-loops followed by a non-self-loop transition; each such path is feasible for some specific value of `trigger_index`. Transitions parameterized on frames, as in the monitor of Fig. 7 are handled in analogous manner: our infrastructure implements a loop which iterates over frames in a packet, and exits when the index of the current frame equals the value of a symbolic variable.

```
1  enum STATE { INIT, RCVD, END };
2
3  static STATE curr_state = INIT;
4
5  void matching_content_type_server(RECORD *P) {
6    if (curr_state == INIT && P.source == CLIENT &&
7        byte_to_int(P->epoch, sizeof(P->epoch)) == 0) {
8      uint8_t content_type = P->content_type;
9
10     kleener_make_symbolic(&P->content_type,
11                           sizeof(P->content_type), "content-type");
12     klee_assume(P->content_type != content_type);
13     curr_state = RCVD;
14   }
15   else if (curr_state == RCVD && P.source == SERVER) {
16     if (P->content_type != Alert_REC)
17       assert(0 && "Mismatching Content Type");
18     curr_state = END;
19   }
20   else if (curr_state == RCVD && P.source == CLIENT)
21     curr_state = END;
22   else if (curr_state == END) return;
23 }
```

**Figure 8: The C implementation of the monitor of Fig. 4.**

*Parser & Serializer.* These two components are optional, but we have implemented them nonetheless for the two protocols we have tested. Based on the protocols' RFC, we have constructed a parser which parses protocol packets into data structures; this allows monitors to access specific packet fields (e.g., `P->epoch` in Fig. 8). Conversely, the serializer serializes into protocol packets the data structures, whose fields may have been modified or made symbolic by the monitors. Without these two components, monitors would need to directly manipulate the bytes at the appropriate field offsets within the byte array that stores the protocol packet.

## 5.3 Optional SUT-Specific Preparations

As mentioned in §3, for some network protocols, it may be beneficial to make small modifications to their implementations that may speed up SE. For example, if the protocol transmits packets between entities in encrypted form, we can replace the concerned cryptographic functions (e.g., encryption, decryption, hashing, etc.) by simplified ones (e.g., identity). Another optional modification is to disable timing-based behavior. For example, in DTLS, if no response to a flight of messages is received within a certain amount of time, the flight may be re-transmitted or the entire interaction may be aborted. While such SUT modifications are optional, disabling timeouts or increasing their values simplifies monitor construction.

### 5.4 Testing Using Symbolic Execution

Testing is performed by symbolically executing the constructed test harness, in which the protocol parties and monitors execute (cf. §3.1). We have used KLEE [8] as symbolic execution engine. For each explored path, KLEE produces concrete values for the data that has been made symbolic by a specific monitor. Some of these values cause crashes in the SUT or assertion failures in the monitor's code. We have constructed test cases that exhibit the bugs we have detected by substituting these values in the corresponding exchanged messages, and reported these bugs to the SUT's developers.

## 6 EVALUATION

In this section, we evaluate our technique on a variety of widely-used DTLS and QUIC implementations. With this evaluation, we intend to answer the following questions:

- Q.1 Is our technique, which is based on a SUT-agnostic set of monitors, as *effective* as another state-of-the-art SE-based testing technique that requires intrusive modifications (e.g., declaring packet fields symbolic, adding assertions, etc.) to each SUT for testing the same protocol requirements?
- Q.2 If it is equally effective, does it require *less or more effort?* And is it *faster or slower*?
- Q.3 Can it detect *previously unknown bugs* in the most recent versions of the two network protocols we consider?
- Q.4 Can it do this *without any* modifications to the SUTs?

### 6.1 Comparison with a Related Technique

To answer Q.1 and Q.2, we compared the approach described in this paper with another technique that our group has proposed [3], which is also based on requirement-driven symbolic execution. The corresponding paper of our prior work comes with a replication package [4], which we used for the comparison of the two techniques. We implemented as monitors, in a set of new files, all the DTLS protocol requirements that were checked in that work, and applied our new technique on the same versions of the four DTLS implementations that were tested, starting from the original code base of these SUTs (i.e., *without* any modifications). The technique we describe in this paper was able to reproduce *all* the vulnerabilities and non-conformance bugs that were previously identified. Moreover, the new technique did this:

*With less effort.* Both approaches require having a test harness and, optionally, making modifications to the code of the SUT (e.g., to de-randomize it and disable encryption). However, in the old approach [3], the code that checks the requirements needs to be added as calls to a shared library directly in the code of the SUT, which requires understanding and modifying a complex code base. In contrast, in the approach of this paper, the requirements are encoded as monitors in a set of new files (i.e., outside the SUTs), which can be shared when testing various versions of the same protocol implementation (or even different ones), and require no code interventions. Even disregarding this particular advantage, and concentrating on the effort required to test just one SUT and only one of its versions, the coding effort is overall smaller in our approach. For example, for OpenSSL 3.0.0-alpha12, whose code base consists of ≈ 289 160 lines of code (LOC), the test harness required ≈ 1 250 LOC for the old approach, but only ≈ 670 for the one of this paper. In addition, the code for the 16 server and 14 client DTLS protocol requirements (i.e., 30 requirements in total) comprises ≈ 1 700 LOC in their approach, while the same requirements need ≈ 1 200 LOC to be implemented as a set of monitors. (Thus, monitors require about 40 lines on average.) Observe that the total amount of code that both approaches require is relatively small compared to the code size of the (quite complex) SUT which is tested. An additional drawback of the old approach [3] is that checking different protocol configurations relies on capturing a new set of packets, whereas in the new approach this is just a matter of adjusting the desired configuration through the SUT's API within the test harness.

*Considerably faster.* The old approach [3] relies on completing a protocol interaction. (For example, in the case of DTLS, establishing a handshake or definitely failing to do so.) This means that symbolic execution may need to explore many paths unnecessarily. In contrast, in the approach that this paper advocates, a monitor that detects a requirement violation can signal the error and stop symbolic execution long before it reaches code "deep" in the protocol interaction. As concrete evidence for what this means in practice, the old approach detects the violation of the "Handshake Version" requirement for the OpenSSL 3.0.0-alpha12 DTLS server after exploring eight paths in 1h18m36s (cf. [3, Table II]), while the technique of this paper detects the violation after exploring six valid plus one erroneous (i.e., a total of seven) paths in just 1m39s. The same requirement for OpenSSL 3.0.0-alpha12 DTLS clients explores 6 paths in 43m49s [3, Table III], while the approach of this paper explores $4+1=5$ paths in 1m48s. Similar differences exist for many other requirements and SUTs. For example, the violation of the "Record Length" requirement for Eclipse's TinyDTLS (commit 7068882) requires exploring 112 paths in 2h06m13s [3, Table IV], while the approach of this paper explores just 17 paths in only 5 seconds.

### 6.2 Effectiveness on DTLS Implementations

To answer Q.3 for the DTLS protocol, we represented a total of 21 requirements as monitors, and checked them in a scenario where the test harness completes an interaction between the client and server and then exchanges some application data. As SUTs we used four widely-used DTLS implementations (MbedTLS 3.1.0, OpenSSL 3.0.2, Eclipse's TinyDTLS at commit e4d547ce, and WolfSSL 5.3.0). For all SUTs, the versions or commit points we used were the latest at the time when our experiments were conducted. Our technique did not detect any bugs in these versions of MbedTLS and OpenSSL. However, it detected a total of thirteen previously unknown bugs in TinyDTLS and WolfSSL, which we describe below. We mention that all the bugs we report in this paper for DTLS implementations have been communicated to their developers and have *all been fixed by now*; more information about them and what changes their fixes required can be found by following the hyperlinks in the last column of Table 1. We analyze them below.

In Eclipse's TinyDTLS implementation, the monitor-based technique of this paper detected five previously unknown bugs: (1) If a TinyDTLS client receives a *CertificateRequest* with more certificate types than the number specified in the `certificate_types_count` field, two out-of-bounds pointers are accessed. (2) Similarly, an

**Table 1: The discovered, previously unknown, bugs in the most recent versions of two DTLS and three QUIC implementations. The entries in the "Status" column contain links to GitHub issues that were used for reporting and discussing the bugs.**

| SUT | Role(s) | Short Description of the Bug(s) | Status |
|---|---|---|---|
| TinyDTLS | Client | Out-of-bound pointers when a client receives a *CertificateRequest* with an invalid value for the `certificate_types_count` field. | Fixed |
| | Client | Out-of-bounds pointer when a client receives a *CertificateRequest* with an invalid value for the `signature_hash_algorithms_length` field. | Fixed |
| | Client&Server | Out-of-bounds pointers caused by a handshake message other than *ClientHello* whose `msg_type` field has been set to `ClientHello`. | Fixed |
| | Client | Assertion failure when a client expects a *ServerHello* message but instead receives a *ServerKeyExchange*. | Fixed |
| | Client&Server | Out-of-bounds pointers caused by a *ChangeCipherSpec* message whose `content_type` field has been set to `Alert`. | Fixed |
| WolfSSL | Client&Server | Four (4) non-conformance bugs due to insufficient checking of length-related fields. | All Fixed |
| | Server | Non-conformance bug when the message sequence numbers (SNs) in the initial *ClientHello* and *HelloVerifyRequest* differ. | Fixed |
| | Server | Non-conformance bug when the message sequence numbers (SNs) in the second *ClientHello* and *ServerHello* differ. | Fixed |
| | Server | Non-conformance bug when a server proceeds with the interaction despite receiving a *ClientHello* message with a non-existing version. | Fixed |
| | Client | Non-conformance bug when a client fails to update the highest validated record SN; thus, accepting messages with outdated SNs. | Fixed |
| LSQUIC | Client&Server | Non-conformance bug when a client (or server) fails to discard a packet with the fixed bit assigned to zero. | Confirmed |
| | Client&Server | Three (3) non-conformance bugs occur where the incorrect error code is returned by a client (or server) in the presence of invalid input. | All Fixed |
| Quicly | Server | Non-conformance bug when a server fails to discard a `NEW_TOKEN` frame generated by a client and proceeds with the interaction. | Fixed |
| Picoquic | Server | Non-conformance bug when a server fails to discard a `NEW_TOKEN` frame generated by a client and proceeds with the interaction. | Fixed |
| | Client&Server | Three (3) non-conformance bugs occur where the incorrect error code is returned by a client (or server) in the presence of invalid input. | All Fixed |

out-of-bounds pointer is dereferenced in some cases when the `signature_hash_algorithms_length` field is inconsistent with the actual size of the `signature_hash_algorithms` field. (3) During the DTLS handshake, if a TinyDTLS client (or server) is fed a message for which the handshake type is advertised as *ClientHello* but its size is smaller than the minimum size for a valid *ClientHello* message, the receiving party initiates the cookie generation process. In this case, the cookie generation function tries to dereference pointers past the boundary of the message in two different locations. (4) Similarly, when a TinyDTLS client expects a *ServerHello* message, if the handshake type is advertised as *ServerKeyExchange*, an incorrect function for processing the message is invoked which, later, causes an assertion to fail. (5) If a TinyDTLS server (or client) expects a *ChangeCipherSpec*, but is sent a record for which the content type is advertised as *Alert*, the server mistakenly invokes the wrong function for processing that record as an *Alert*, which causes accessing an out-of-bounds pointer of the record. This bug was caught with the monitor of Fig. 4 whose implementation was shown in Fig. 8.

In addition, we detected a total of eight previously unknown bugs in WolfSSL's implementation. Four of them are non-conformance bugs due to insufficient checking of length-related fields. The remaining four are as follows: (1) A non-conformance bug where the implementation fails to ensure that the message sequence in a *ClientHello* message sent to the server should be reused by the server in a *HelloVerifyRequest* message. (2) A similar non-conformance occurs for the message sequence in the second *ClientHello* and the *ServerHello* message. These two non-conformances make a WolfSSL DTLS server susceptible to message sequence duplication in the case of multiple *HelloVerifyRequests* and multiple cookie exchanges, respectively. (3) A non-conformance bug occurs if the server is fooled to proceed with the handshake interaction despite receiving a *ClientHello* message with a non-existing protocol version. (4) An insufficient check in the anti-replay mechanism in WolfSSL's implementation creates a situation where the client fails to update the highest validated record sequence number. The failure makes the client susceptible to replay attacks where one party is fed outdated messages.

To answer Q.4, we mention that all these bugs were detected in *completely unmodified* SUTs. However, to also see whether *some* modifications in the SUTs would somehow improve the effectiveness of our technique, we also performed an experiment where we disabled the checks of the hash values in the *Finished* messages. We did not detect any more bugs in that setting.

## 6.3 Effectiveness on QUIC Implementations

To answer Q.3 for the QUIC protocol, we represented a total of 16 requirements from its RFC [18] as monitors, and checked them in a scenario where in addition to completing the underlying TLS handshake, two parties of the protocol (client and server) exchange various transport parameters. As SUTs we used the most recent versions or commits of three C-based open-source QUIC implementations: LiteSpeed QUIC 3.2.0, quicly 482610bf, and picoquic 7f2fbdfb. The LiteSpeed QUIC (LSQUIC) library is a commercial implementation of QUIC and HTTP/3 functionality for servers and clients. Quicly is a modular implementation of the QUIC protocol designed to be used within the H2O HTTP server. Picoquic is a minimal implementation of QUIC aligned with the IETF specifications.

In contrast to the DTLS experiments, and because QUIC is a network protocol whose interaction is almost fully encrypted, for QUIC implementations we made two small modifications to the SUTs: we disabled encryption and timeouts. These changes were quite small though; less than 100 LOC for each of these three implementations.

Our technique detected a total of nine bugs in them, with all but one of them fixed by now. Six of these bugs (second lines for LSQUIC and picoquic in Table 1) are non-conformance bugs that manifest as incorrect error codes returned by a client (or server) in the presence of invalid input. We mention that one of these six bugs was caught with the monitor of Fig. 7. The remaining three bugs, one per implementation, are: (1) A non-conformance bug in LSQUIC when a client (or server) fails to discard a packet with the fixed bit assigned to zero. The LSQUIC developers confirmed the issue as a non-conformance to the QUIC specification but noted that, for performance and simplicity, their implementation currently does not come with logic to check the fixed bit. (2&3) The QUIC protocol mandates that only servers are allowed to generate `NEW_TOKEN` frames. Servers receiving `NEW_TOKEN` frames are expected to discard them while generating an appropriate error. In both the quicly and picoquic implementations, the server failed to generate an error

upon receiving a NEW_TOKEN frame. Both implementations have by now been fixed.

## 7 THREATS TO VALIDITY

In this section, we briefly discuss limitations of our technique. The main limitation is related to the scope of requirements to be tested. As our monitors can only observe and manipulate the (fields of) messages exchanged during a protocol interaction, they are limited to testing requirements directly observable from these interactions. Consequently, our technique cannot effectively capture protocol requirements that involve the SUT's internal state and do not result in change in externally observable behaviour. Luckily, for both DTLS and QUIC, there are only few such requirements.

Unsurprisingly, our technique is sensitive to the definition and accuracy of the monitors used. A monitor that is flawed or that inadequately captures the intended requirements can result in erroneous test outcomes, manifesting as false positives or false negatives. During our experiments, we did not encounter any false positives; this is corroborated by the fact that all issues we discovered were confirmed (and most fixed) by the protocol implementors.

On a related point, the extraction of protocol requirements from natural language documents (the RFC and all its errata, which are in most cases separate documents) and the definition of monitors is currently not automated. This is an orthogonal issue, which is outside the scope of this paper. Work along these lines exists in the literature (e.g., by Chen et al. [14]), but how to take advantage of such work for protocols such as DTLS whose errata are separate documents is a challenge left for future work.

## 8 RELATED WORK

Symbolic execution was introduced already in the 70's [19], In the last fifteen years, it has developed into a powerful software testing technique, being able to find safety violations that have been difficult to expose using other techniques [9].

Applications of symbolic execution (SE) for testing stateful software such as network protocol implementations are not so numerous. Several approaches, including [24, 26, 29, 30, 32, 34], allow a test setup to designate fields in incoming packets as symbolic, in order to explore code paths reachable for different values of such fields. Tempel et al. [34] show that the test coverage of SE can be increased if statefulness is taken into account by specifying a protocol's state machine. Wen et al. [35] use model learning to generate a protocol's state machine, which is then supplied as an additional input in SE and the approach is applied to the FTP protocol. These approaches use SE to detect generic run-time errors (e.g., in memory accesses), but are not applied to check protocol requirements.

KleeNet [30] supports checking of correctness properties after manually inserting assertions into the SUT's source code. An extension of KleeNet with a simulated network model [29], supports SE of a distributed application executing on several nodes. Occurrences of packet loss and node failures can be represented by symbolic variables, prompting SE to explore different error scenarios. Assertions can be inserted into the SUT's code to check consistency of the distributed state. The use of symbolic variables to represent non-deterministically occurring errors is analogous to our use of symbolic variables for nondeterminism in monitors. Ideas of KleeNet are used in SymNet [28], which employs the $S^2E$ platform [15], to perform integration testing of different synchronization scenarios and apply it to test the Host Identity Protocol [23]. These lines of work aim to uncover interoperability issues under different synchronization and failure scenarios. They do not test that packet data is manipulated according to protocol requirements; in particular they have limited abilities to detect flaws and requirement violations in processing of adversarial (i.e., non-valid) inputs. Interoperability is also addressed by Rath et al. [26] who present a SE-based methodology applied to QUIC implementations, and by Pedrosa et al. [24] whose work uses symbolic execution to characterize messages that during a session can be sent by one party but rejected as non-compliant by the other, and constraint solving to find interoperability violations.

Checking correctness of packet data processing is addressed by SymbexNet [32]. However, SymbexNet's technique applies symbolic execution and requirement checking separately. It first symbolically executes the protocol parties jointly on partly symbolic input, in order to generate test inputs that explore a variety of code paths. Only code paths that can be exercised in a non-adversary environment are explored. The generated test inputs are thereafter supplied to the party under test, while monitoring the resulting execution to check against requirements which have been encoded as rules over packet sequences. SymbexNet's approach exercises a wide variety of code paths in a protocol party, but may still miss specific requirement-violating inputs. This can happen e.g., when the same code path can be exercised by both violating and non-violating inputs, and the symbolic execution engine happens to generate a non-violating input for this particular path. This problem was addressed by our prior work [3] for testing DTLS implementations against the requirements using symbolic execution. That work addresses statefulness by presenting the sequence of all input packets sent to the SUT during an interaction as input before SE starts, which requires pre-recording such as sequence of packets. Since SE will replay the pre-recorded session, randomization and other sources of nondeterminism must first be manually removed from the SUT's source code. Each protocol party is tested separately. Moreover, for testing different protocol configurations, additional sequences of packets must be pre-recorded. In contrast, our use of protocol parties (e.g., client/server) to perform the interaction allows to reuse monitors for checking the same requirement in different configurations simply by adjusting the desired configuration through the SUT's API within the test harness. In the technique of this paper, all protocol parties can be tested using the same test harness. Furthermore, our prior work [3] represented stateful protocol requirements by manually inserting assumptions and assertions into the source code of the SUT, whereas our use of monitors provides a structured way of representing requirements for processing packet data and does not require modification to the SUT's code. Thus, monitors can be reused out-of-the-box for different implementations of the same protocol.

To uncover deep bugs in stateful protocol implementations, Chau et al. [12, 13] use symbolic execution for a form of differential testing of non-stateful libraries that classify certificate chains as valid or invalid. Using symbolic execution, path constraints for the "valid"

and "invalid" outcomes are generated. Path constraints from different implementations are compared, and discrepancies are investigated. This approach can be suitable for non-stateful components, if requirements are difficult to formalize precisely. In our approach, we can check individual protocol requirements separately, without needing to access other implementations than the one under test.

Greybox fuzzing is a highly successful testing technique for detecting crashes and vulnerabilities in non-stateful programs [36]. Adaptations to testing stateful programs include LTL-Fuzzer [21], which employs greybox fuzzing to detect violations of requirements expressed in temporal logic. LTL-Fuzzer requires manually instrumenting the SUT's code with requirement-specific checks. AFLNET [25], further developed in SnapFuzz [1], aims to increase the effectiveness of greybox testing on stateful programs, such as network protocol implementations. Like many other applications of greybox fuzzing, AFLNET mainly finds run-time errors (e.g., crashes); it does not check protocol requirements.

The well-known concept of monitors [31] is commonly used to check requirements. Monitors have been used to check a rich range of properties, including properties expressed in fragments of first-order temporal logic (e.g., [6, 17]). Run-time verification (RV) [11] checks properties of concrete individual executions, whereas our adaption of monitors for SE allows to check such properties for all possible values of symbolic inputs. Previous combinations of RV and SE (e.g., the work of Artho et al. [2]) exhibit the same separation between symbolic execution and requirements checking as in SymbexNet.

For *black-box* testing, McMillan and Zuck [20] propose a monitor-based approach, which is applied to QUIC implementations. In their approach, a monitor-like machinery which can detect requirement violations is constructed and applied during interoperability testing. Unlike the approach of this paper, their approach solely evaluates how implementations respond to specification-conforming inputs, and therefore achieves limited coverage of adversarial inputs. Their paper acknowledges this as something limiting the coverage achieved, and leaves combination of their technique with symbolic execution as future work.

## 9 CONCLUSIONS

In this paper, we presented a technique that employs symbolic execution for testing stateful network protocol implementations. The technique is based on the use of monitors to concentrate the logic for testing each requirement into a software component external to the SUT. Monitors provide a uniform mechanism for checking requirements and obviate the need for modifying the SUT. To evaluate the effectiveness of our technique, we applied it to four widely-used DTLS implementations and three implementations of the QUIC protocol. Our technique was able to reproduce all bugs detected by another recently proposed related technique, but with less implementation effort and considerably faster. It was also able to detect several previously unknown bugs in newer versions of these implementations. With a single exception, due to a specific design choice for performance reasons in the LSQUIC implementation, all the requirement violations and bugs that our technique detected have since then been fixed in the corresponding implementations.

We argue that our monitor-based technique is a suitable basis for building environments that perform testing of protocol implementations based on symbolic execution. Such test environments can automatically check requirements in different implementations of the protocol. Moreover, they can easily be extended to check additional requirements by constructing corresponding monitors. In this way, common test infrastructure can be maintained and used by the community of developers for a specific network protocol.

## REPLICATION MATERIAL

We provide replication material [5] for all our experiments.

## ACKNOWLEDGMENTS

This research was partially funded by the Swedish Foundation for Strategic Research (SSF) through project aSSIsT and the Swedish Research Council (Vetenskapsrådet).

## REFERENCES

[1] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-throughput Fuzzing of Network Applications. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. ACM, New York, NY, USA, 340–351. https://doi.org/10.1145/3533767.3534376

[2] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Richard Washington. 2005. Combining test case generation and runtime verification. *Theor. Comput. Sci.* 336, 2-3 (2005), 209–234. https://doi.org/10.1016/j.tcs.2004.11.007

[3] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. 2022. Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification. In *IEEE Conference on Software Testing, Verification and Validation (ICST 2022)*. IEEE, 70–81. https://doi.org/10.1109/ICST53961.2022.00019

[4] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. 2022. Replication Material for the ICST 2022 Paper: Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification. https://doi.org/10.5281/zenodo.5929867

[5] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. 2024. Replication material for the ARES 2024 paper: Monitor-based Testing of Network Protocol Implementations Using Symbolic Execution. https://doi.org/10.5281/zenodo.11945614

[6] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (2015), 15:1–15:45. https://doi.org/10.1145/2699444

[7] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2017. A Messy State of the Union: Taming the Composite State Machines of TLS. *Commun. ACM* 60, 2 (Feb. 2017), 99–107. https://doi.org/10.1145/3023357

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI '08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[9] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[10] Marco M. Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. 2014. Heartbleed 101. *IEEE Secur. Priv.* 12, 4 (2014), 63–67. https://doi.org/10.1109/MSP.2014.66

[11] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A Survey of Runtime Monitoring Instrumentation Techniques. In *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, Pre-Post@iFM 2017* (Torino, Italy) *(EPTCS, Vol. 254)*, Adrian Francalanza and Gordon J. Pace (Eds.). 15–28. https://doi.org/10.4204/EPTCS.254.2

[12] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy* (San Jose, CA, USA) *(SP 2017)*. IEEE Computer Society, 503–520. https://doi.org/10.1109/SP.2017.40

[13] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study

on PKCS#1 v1.5 Signature Verification. In *26th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS 2019)*. The Internet Society. https://doi.org/10.14722/ndss.2019.23430

[14] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. 2018. RFC-directed differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE 2018)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 859–870. https://doi.org/10.1145/3180155.3180226

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1 (2012), 2:1–2:49. https://doi.org/10.1145/2110356.2110358

[16] Tim Dierks and Eric Rescorla. 2008. The Transport Layer Security TLS Protocol Version 1.2. RFC 5246. http://www.rfc-editor.org/rfc/rfc5246.txt

[17] Klaus Havelund and Doron Peled. 2018. Efficient Runtime Verification of First-Order Temporal Properties. In *Model Checking Software - 25th International Symposium, SPIN 2018 (LNCS, Vol. 10869)*, María-del-Mar Gallardo and Pedro Merino (Eds.). Springer, 26–47. https://doi.org/10.1007/978-3-319-94111-0_2

[18] Janardhan Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. , 151 pages. https://www.rfc-editor.org/rfc/rfc9000.txt

[19] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[20] 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 2019)*. ACM, 227–240. https://doi.org/10.1145/3341302.3342087

[21] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE' 22)*. ACM, New York, NY, USA, 1343–1355. https://doi.org/10.1145/3510003.3510082

[22] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf

[23] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. 2008. Host Identity Protocol. RFC 5201. https://www.ietd.org/rfc/rfc5201.txt updated by RFC 6253.

[24] 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* (Oakland, CA, USA) *(NSDI 15)*. USENIX Association, 485–498. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa

[25] Van-Thuam Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST 2020)*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[26] Felix Rath, Daniel Schemmel, and Klaus Wehrle. 2018. Interoperability-Guided Testing of QUIC Implementations using Symbolic Execution. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Heraklion, Greece) *(EPIQ@CoNEXT 2018)*. ACM, 15–21. https://doi.org/10.1145/3284850.3284853

[27] Eric Rescorla and Nagendra Modadugu. 2012. Datagram Transport Layer Security Version 1.2. RFC 6347. , 32 pages. http://www.rfc-editor.org/rfc/rfc6347.txt

[28] Raimondas Sasnauskas, Philipp Kaiser, Russ Lucas Jukic, and Klaus Wehrle. 2012. Integration testing of protocol implementations using symbolic distributed execution. In *20th IEEE International Conference on Network Protocols* (Austin, TX, USA) *(ICNP 2012)*. IEEE Computer Society, 1–6. https://doi.org/10.1109/ICNP.2012.6459940

[29] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks* (Stockholm, Sweden) *(IPSN 2010)*. ACM, 186–196. https://doi.org/10.1145/1791212.1791235

[30] Raimondas Sasnauskas, Jó Ágila Bitsch Link, Muhammad Hamad Alizai, and Klaus Wehrle. 2008. KleeNet: automatic bug hunting in sensor network applications. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems* (Raleigh, NC, USA) *(SenSys 2008)*. ACM, 425–426. https://doi.org/10.1145/1460412.1460485

[31] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[32] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. 2014. SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications. *IEEE Trans. Software Eng.* 40, 7 (2014), 695–709. https://doi.org/10.1109/TSE.2014.2323977

[33] W. Richard Stevens. 1990. *Unix network programming*. Prentice Hall.

[34] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. 2023. Specification-based Symbolic Execution for Stateful Network Protocol Implementations in the IoT. *IEEE Internet of Things Journal* (2023). https://doi.org/10.1109/JIOT.2023.3236694

[35] Shameng Wen, Qingkun Meng, Chao Feng, and Chaojing Tang. 2017. A model-guided symbolic execution approach for network protocol implementations and vulnerability detection. *PloS one* 12, 11 (2017), e0188229. https://doi.org/10.1371/journal.pone.0188229

[36] Michał Zalewski. 2013. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.