# Prognosis: Closed-Box Analysis of Network Protocol Implementations

Tiago Ferreira
University College London

Harrison Brewton
University of Wisconsin–Madison

Loris D'Antoni
University of Wisconsin–Madison

Alexandra Silva
University College London

## ABSTRACT

We present Prognosis, a framework offering automated closed-box learning and analysis of models of network protocol implementations. Prognosis can learn models that vary in abstraction level from simple deterministic automata to models containing data operations, such as register updates, and can be used to unlock a variety of analysis techniques—model checking temporal properties, computing differences between models of two implementations of the same protocol, or improving testing via model-based test generation. Prognosis is modular and easily adaptable to different protocols (e.g., TCP and QUIC) and their implementations. We use Prognosis to learn models of (parts of) three QUIC implementations—Quiche (Cloudflare), Google QUIC, and Facebook mvfst—and use these models to analyze the differences between the various implementations. Our analysis provides insights into different design choices and uncovers potential bugs. Concretely, we have found critical bugs in multiple QUIC implementations, which have been acknowledged by the developers.

## CCS CONCEPTS

• **Theory of computation** → **Logic and verification**; **Verification by model checking**; **Abstraction**; *Transducers*; *Automata over infinite objects*;

## KEYWORDS

model learning, synthesis, varied abstraction modelling, bug finding, protocol state machines

**ACM Reference Format:**
Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: Closed-Box Analysis of Network Protocol Implementations. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452296.3472938

## 1 INTRODUCTION

Implementations of network and security protocols such as TCP/IP, SSH, TLS, DTLS, and QUIC are prominent components of many internet and server applications. Errors in the implementations of these protocols are common causes of security breaches and network failures (e.g., the Heartbleed OpenSSL vulnerability [4] that rendered SSL/TLS connections futile due to leaked secret keys, and others [6, 7]). Many approaches have been proposed to verify both specifications and implementations of the above protocols in an attempt to provide safety and correctness guarantees [18, 20–22, 27]. For example, Bhargavan et al. [14] are designing a formally verified (using a theorem prover) implementation of the TLS protocol that is provably free of many types of security vulnerabilities.

Most existing approaches suffer from being *monolithic*—small changes to the implementation require large changes to the verification process—and require *high expertise*—one needs to know the protocol very well and have a priori knowledge of what properties the implementation should satisfy. One way to avoid these limitations is to build *models* of the implementation, which provide abstractions of the critical components of an implementation and enable a number of powerful analyses such as model-checking and model-based test generation.

For example, instead of directly analyzing the binary or source code of a TCP implementation, one can analyze a model that only describes what types of packet flags (`SYN`, `ACK`, etc.) the implementation exchanges during a handshake. However, this approach is still monolithic. First, analyzing the model requires guessing what properties it must satisfy. Second, designing models still requires expertise, and whenever the implementation is modified, one has to manually update the model to retain its faithfulness to the implementation, updating not only the modified sections but also ensuring existing ones are truly unaffected. This problem is well-known in the model checking literature and recently McMillan and Zuck [27] showed, using Microsoft QUIC as an example, that creating a faithful model of a protocol implementation is a challenging, time-consuming problem that requires a number of iterations between the model designer and the protocol-implementation developers.

Orthogonally, several works have shown that one can often *learn* a model from implementations (e.g., for passport [10] and bank card [9] protocols). This idea is called *model learning* [32] and it builds on the fact that many classes of finite automata (often used as models) can be inferred by testing the implementation on a set of traces. Fiterău-Broştean et al. [22] applied model learning to detect an anomaly in a real TCP implementation, but their system is still monolithic—i.e., it requires one to *manually* design a mapper between the *abstract* traces of the model and the *concrete* traces of the implementation, a task requiring expert knowledge of the protocol logic. Thence, the system of Fiterău-Broştean et al. [22] is not reusable for different protocols and implementations.


*SIGCOMM '21, August 23–27, 2021, Virtual Event, USA*

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472938

In this paper, we consider the following question:

*Can we design general and reusable techniques to detect logic errors affecting the interaction with protocol implementations without knowing a priori what properties these errors may violate?*

We present Prognosis, a *modular* and *reusable* framework for learning and analyzing models of network protocol implementations, specialising in bug finding and knowledge acquisition rather than providing verified guarantees. Unlike existing approaches, Prognosis can easily be adapted to handle different protocols and protocol implementations, and the programmer *does not* need to manually program the logic to map abstract traces to concrete traces of the protocol implementation, relying instead on a *reference implementation* that they trust by instrumenting its protocol logic components. Although any valid implementation of the protocol can be used as a reference implementation, different implementations will vary in their ease of modification depending on how they were designed. The Prognosis Adapter, generated from the instrumentation, creates the ideal separation of concerns between implementation-specific details and protocol-specific details. Crucially, the same Adapter can be reused for different implementations of the same protocol with a simple change, as reference implementations are, by design, able to communicate with any other valid implementation of the same protocol. Furthermore, Prognosis makes it easy to experiment with different levels of abstraction in the model precision and with several analyses that expose varying types of bugs.

We use Prognosis to analyze 1 implementation of TCP and 4 of QUIC—Quiche (Cloudflare), Google QUIC, Facebook mvfst, and QUIC-Tracker [28]. For QUIC, we experiment with varying levels of abstractions and learn models that: 1) only characterize the packet's frames, and 2) concrete data such as packet numbers. The first type of abstraction allows us to use existing decision procedures to compare whether the models learned from different implementations are equivalent. When dealing with protocols as big as QUIC for example, finding a difference in the models does not necessarily indicate a bug, it can also signal different design decisions as allowed by QUIC's flexible specification. Nonetheless, some of these differences the models capture provide greater insight into the consequences of specific design decisions that are sometimes taken lightly, or sections of the specification that should be stricter. We found one such case which triggered a change to the specification to better formalize the intended behavior. Abstract models also allow us to verify and test safety and liveness properties. By inspecting counterexamples produced by checking safety properties we identified 3 bugs in QUIC implementations, which have been acknowledged by the developers.

**Contributions and road map.** In a nutshell, the paper contains the following contributions:

(1) Prognosis, a framework with a modular architecture that enables reusability: different protocols and protocol implementations can easily be swapped without changes to the learning engine (Section 2).
(2) Configurable levels of abstraction in learned models. This helps with scalability and performance yet does not compromise correctness—once a bug is discovered at the model level, Prognosis creates concrete traces to check whether the bug is in the implementation or is a false positive (that can be used to refine the model). (Section 4).
(3) The various types of model Prognosis can learn and synthesize expose a range of analyses, e.g., model-based testing and decision procedures, which we use in finding bugs. (Section 5).
(4) Case studies: a TCP implementation and several QUIC implementations. The TCP case study shows that our framework can recover previous results [22], albeit with much less manual configuration. The QUIC case study highlights that the framework can be used to uncover bugs as well as to gain insight into different design choices and aid in RFC Improvements (Section 6). The bugs uncovered in two popular implementations of QUIC were confirmed by the developers. One of the bugs (if exploited) could make the QUIC server vulnerable to DoS attacks and it highlighted that in fact a part of the protocol was totally missing from the implementation.

We review relevant related work in Section 7 and conclude the paper in Section 8 with a discussion of the limitations of the framework, and directions for further work.

***Ethical statement*** Work does not raise any ethical issues.

## 2 ARCHITECTURE

In this section, we give an overview of the architecture of our framework. Prognosis is comprised of three modules (see Figure 1), which are parametrized by inputs provided by the user who is analyzing the protocol.

*System Under Learning (SUL).* The SUL is the protocol implementation we are analyzing—e.g., a TCP server. This implementation is accessed in a closed-box fashion—i.e., all we assume is that we can send packets to and receive packets from it. In general, implementations are complex, and we cannot hope to directly learn models of their entire behaviors. A user of Prognosis provides an Adapter pair $(\alpha, \gamma)$ containing an abstraction function $\alpha$ that maps concrete packet traces of the SUL to simplified abstract traces. For example, the function $\alpha$ might simplify TCP packets to only consider whether they are of type SYN, ACK, or SYN-ACK. We also need a concretization function $\gamma$ that maps simplified abstract traces to concrete ones accepted by the SUL. Unlike existing approaches that require the user to explicitly implement this function [22], Prognosis only requires the user to instrument an existing reference implementation of the same protocol we are analyzing to produce concrete packet traces from abstract ones. We describe this setup in detail in Section 3. Now that we have an Adapter that can map between abstract and concrete traces, we have the interface required to learn models and perform analysis.

*Learning Module.* We use existing active model learning algorithms [25] to learn a model of the SUL. Active learning algorithms work by directly interacting with the SUL instead of relying on passive data such as logged events or historic data. This allows us to do closed-box learning, where we don't have access to the inner workings of the SUL, and guarantees that the learner is able to get answers to every query it may formulate.

Specifically, we focus on algorithms based on the *Minimally Adequate Teacher Framework*, where the learning algorithm can

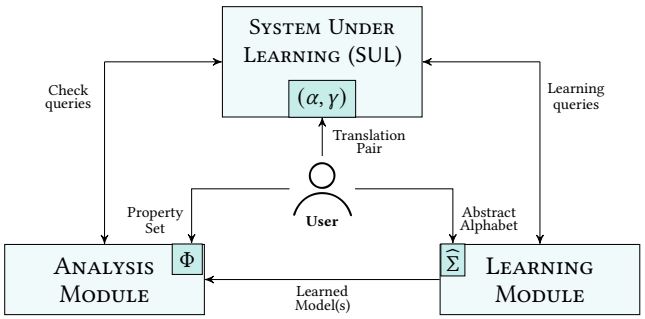


**Figure 1: Prognosis' modular architecture.**

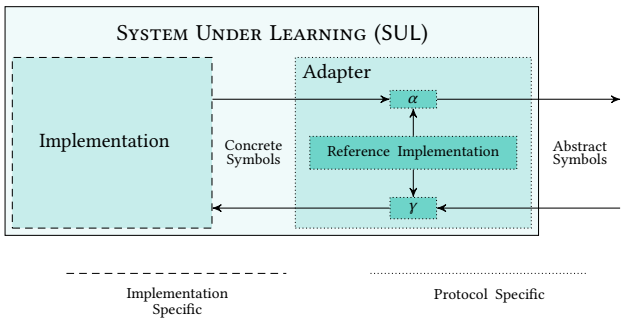


**Figure 2: System Under Learning components.**

perform 2 types of queries: membership and equivalence. Both types of queries are further explored in Section 4.

On an abstract level, the models operate over the simplified abstract traces and therefore abstract away much of the complexity of the SUL, allowing the user to focus on the properties of interest.

On a concrete level this involves sending sequences of input packets to the SUL, and registering the response obtained. For example, the learner might ask what happens when a SYN packet is sent and followed by an ACK. The SUL uses its Adapter to, acting as a client, produce and send these packets to the implementation, and return an answer the learner uses to build an accurate model.

In Section 4, we show how Prognosis can learn different models of varying levels of abstraction—from plain deterministic models to models that include registers to store e.g., packet numbers.

We show how Prognosis can be used to learn a detailed model of the TCP 3-way handshake shown in Figure 3.

*Analysis Module.* This module enables the use of the learned models to analyze the SUL, using a portfolio of techniques to unveil complex bugs and help the user gain insights about the implementation's behavior. For example, Prognosis can automatically compare whether the models learned for two different implementations are equivalent (for different notions of equivalence) and also supports simple visualizations of the learned models that allow a user to visually compare two models for differing behaviors. In Section 6, we show how the automated equivalence check and visualizations aided in detecting anomalies and explaining such anomalies to real developers in our evaluation. For example, Prognosis could detect that a supposedly variable value being transmitted was actually a constant. The full set of analysis exposed by Prognosis is discussed in Section 5.

## 3 SYSTEM UNDER LEARNING (SUL)

The SUL has two sub-components: An Implementation we want to analyze and learn a model of, and a protocol-specific Adapter that uses a translation pair to transform concrete packet traces into simplified abstract traces, and vice-versa. The Adapter is the interface with the learning module (Figure 2). In fact, the learner is completely oblivious to the existence of concrete traces, and communicates only directly with the Adapter, which then communicates with the Implementation. The learning module (described in Section 2) will use the abstract traces to build models of the implementation.

### 3.1 Implementation

The Implementation is the original system we are attempting to learn, for example a running TCP server. In this case we host the implementation being learned, but simply being able to connect to the implementation is enough for Prognosis to learn a model. At the high level, the implementation takes sequences of inputs in a domain $\Sigma$ and produces sequences of outputs in a domain $\Gamma$.

In what follows, we refer to these domains as *alphabets*, the usual terminology in the model learning literature. We will be using different alphabets that get gradually more abstract: the native alphabet, the concrete alphabet, and the abstract alphabet. Each of these alphabets serves its own purpose, and they are the way in which the Prognosis modules communicate. In general, the input and output alphabets of the implementations are complex packets. The following example shows the low-level alphabet used in typical network communications.

*Example 3.1 (Native Alphabet).* The *native alphabet* of a TCP implementation consists of all the possible TCP packets in their binary format, with all their complex fields and restrictions. The input and output alphabets are both $\mathbb{2}$, the binary alphabet, and the packets are sequences of these—i.e., sequences in $\mathbb{2}^*$. These sequences correspond to the binary representations of TCP packets that will be sent over the wire to the Implementation and received back as a response. For example, in our analysis of the TCP Implementation in Figure 3, we use the *Scapy* [8] Python library to send arbitrary TCP packets in $\mathbb{2}^*$ over a socket.

The native alphabet is only useful for creating packets to be communicated, but in general, it is more convenient to use structured alphabets that capture the fields and values of packets. Even though packets are communicated in binary, these same can be represented as a structured alphabet—e.g., a *JSON* object.

*Example 3.2 (Concrete Alphabet).* In our TCP example, the concrete alphabets can be JSON objects with the following structure:

```
{ "isNull": false,
  "sourcePort": 40965,
  "destinationPort": 44344,
  "seqNumber": 48108,
  "ackNumber": 0,
  "dataOffset": null,
  "reserved": 0,
  "flags": "S",
  "window": 8192,
  "checksum": null,
  "urgentPointer": 0 }
```

We will use concrete alphabets—which we below refer to as $\Sigma$ and $\Gamma$—as a machine-readable representation of the native alphabets. Concrete alphabets and native alphabets are necessary to trigger concrete executions of the SUL. However, these alphabets are simply too large (sometimes infinite). When learning a model of an implementation, one has to abstract some parts of the alphabets—e.g., specific packet formats and encrypted messages—to make model learning feasible and helpful. For example, while it is not feasible to learn a model describing an exact TCP implementation, it is possible to learn a model of what types (e.g., `SYN`, `ACK`, `SYN-ACK`) of packets are exchanged during a handshake (Figure 3(b)). This abstraction is handled by the Adapter, as described next.

## 3.2 Adapter

In the architecture of our system (Figure 1), the Adapter, which is part of the SUL (Figure 2), is by far the hardest component the user needs to provide. The Implementation itself is provided to us, and the Learner is abstract, but the Adapter has to translate inputs that the Implementation understands to inputs that the Learner can use, and vice-versa. The Adapter has to know the logic of the protocol to produce concrete packets with the right parameters, and it needs to know how to encode concrete symbols into their binary representation to interact with the Implementation, acting as the client in the connection. Instead of requiring the user to provide an implementation of the Adapter from scratch, Prognosis lets the user resort to a *reference implementation* as basis to build the Adapter. Before we describe this key idea, we set some notation.

The Adapter translates concrete input/output traces into simplified traces for which it is feasible to do model learning. We call the latter abstract traces as they are built from *abstract alphabets* $\widehat{\Sigma}$ and $\widehat{\Gamma}$, which allow us to focus on crucial aspects by abstracting away details that would make the alphabet otherwise too big or infinite. Let us now illustrate a candidate abstract alphabet to learn the TCP handshake of Figure 3.

*Example 3.3 (Abstract Alphabet).* In Figure 3(a) we have packet flags, sequence, and acknowledgment numbers. Because we are only interested in modeling what types of packets are exchanged, our abstract alphabets $\widehat{\Sigma}$ and $\widehat{\Gamma}$ will only contain packet flags. For example, a packet might have the following structure: `ACK+SYN(?,?,0)`. Here, each ? represents a parameter left unspecified. While it seems unnecessary to have these parameters if they are not used in learning, we will use them in Section 4.3 to synthesize a richer model.

*Remark* 3.1 (Nondeterminism). The abstract trace in the previous example does not incorporate the `seqNumber`. Because in TCP the Sequence Number is randomly determined at the start of the connection, we could have two different traces that represent the same 3-way TCP handshake. A choice the user has to make when providing an abstract alphabet is what they plan to model. In this example, we assume the user is trying to learn a model of the 3-way handshake and they know this process should be deterministic. Therefore, providing elements of the abstract alphabet that not only are irrelevant to the model but will cause nondeterminism would be a poor choice of abstract alphabet.

Section 6 shows how Prognosis provides mechanisms to detect when a choice of abstract alphabet results in nondeterminism. In Section 6.2.4, we show a case in which nondeterminism was the result of an undesired protocol behavior.

Going from a native alphabet, to a concrete alphabet, to an abstract alphabet is somewhat simple as we merely remove information deemed unnecessary for learning. However, these details are essential for the learner to communicate with the Implementation, so this missing information will need to be recovered when sending concrete packets to the SUL. Learning algorithms (and our analysis) often need to perform queries to the SUL to decide how the model should be constructed. These queries, given an abstract trace $a$, require one to construct a concrete trace $c$ that is valid in the Implementation and that corresponds to $a$.

Formally, these translations are user-defined functions: An abstraction function $\alpha\colon \Sigma^* \times \Gamma^* \to \widehat{\Sigma}^* \times \widehat{\Gamma}^*$ that maps pairs of input/output traces to abstract traces; and a *concretization function* $\gamma\colon \hat{\Sigma}^* \times \hat{\Gamma}^* \to \Sigma^* \times \Gamma^*$ satisfying $\alpha(\gamma(a)) = a$.

As an example of an abstraction function, we could be removing the `SeqNumber` and `AckNumber` of TCP Packets: e.g., TCP{flags: `SYN`, Seq: `123`, Ack: `0`} would be translated to an abstract TCP Packet TCP{flags: `SYN`, Seq: $\top$, Ack: $\top$}.

Designing the reverse process, that is the concretization function, is a hard task that requires expert knowledge and stands in the way of modularity and reusability. Simplifying this problem is one of the aspects where our work significantly differs from prior work: Fiterău-Broştean et al. [22] have used an architecture reminiscent of ours to learn models of TCP implementations, but require the user to provide a Mapper, which is effectively an implementation of the concretization function $\gamma$. Directly implementing $\gamma$ requires a user to know the protocol logic in detail and to understand what concrete packets are valid and not valid. Essentially, the user needs to implement part of the protocol implementation itself, which not only is a hard task, but somewhat defeats the point of using a closed-box analysis based on model learning. Moreover, this explicit implementation of a concretization function is close to impossible if the protocol relies on a logic of high complexity (as is the case for QUIC), including aspects like key derivation, encryption, or symbols that contain a large number of fields.

Our solution to implementing concretization functions is inspired by the following common expression in cryptography: *Never roll your own Crypto.*—i.e., because cryptographic algorithms tend to be of great complexity, it is best to use existing implementations that have been widely tested instead of implementing them yourself. Building on this insight, Prognosis uses the following key idea.

***Reference implementation as a concretization oracle.***

Instead of manually implementing a concretization function from scratch (i.e., a version of the protocol logic that can produce concrete traces from abstract ones), we rely on a given *reference implementation* to provide ground truth information to the Adapter. Concretely, we rely on the reference implementation to both do the concretization logic and native formatting, as a normal implementation would. Given an abstract query symbol $a$ the Adapter needs to find a concrete packet $c$ to build that matches $a$. We modify the reference implementation so that it can "abstractly execute" abstract packets to identify what concrete packets they can yield. While this sounds like a mouthful, it boils down to identifying and

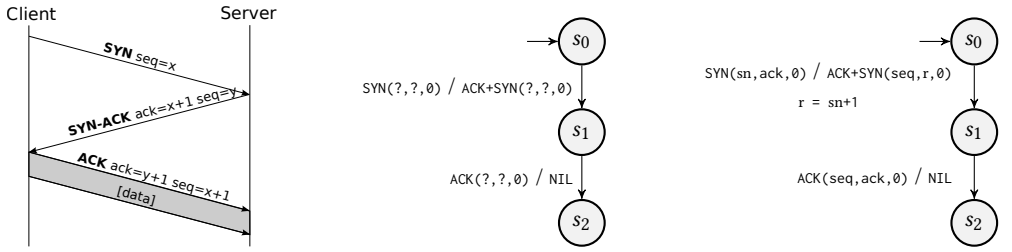


**Figure 3: (a) The TCP 3-way handshake. (b) Learned model (fragment) (c) Synthesized model (fragment) with register r**

using the parts of the reference implementation that operate over the symbols we are interested in.

The main advantage of using a well-tested reference implementation is that it already encodes the desired protocol logic—i.e., in what order packets are transmitted, how they are built, etc. For example, a natural choice for the QUIC reference implementation is QUIC-tracker [28], which is widely used by QUIC implementers.

It is worth noting that this implementation does not have to be 100% correct. It is merely a reference point for modelling the interaction with the target implementation. While correct reference implementations allow us to narrow detected bugs to the target implementation, finding any modelled bugs is useful as it can detect issues in both implementations.

Naturally, although a reference implementation allows us to communicate with the target implementation, it also has behaviour that is not fit for being modelled. Namely, via our modification of the reference implementation we aim to enforce the following properties:

**(1) No unrequested packets are sent to the target Implementation.** We must ensure that any output symbols registered are indeed caused only by the input packets requested.

**(2) All concrete packets sent match the requested abstract packets.** Concrete packets must fulfil the requested abstract symbols fully.

**(3) Both reference and target implementations can be reset on request, returning them to their initial state.** The learner must be able to perform a series of independent queries, as such we must have a way to fully reset the connection to its initial state, ready for a new query to be done.

**(4) Concrete packets constructed or received as a response are saved with their abstract counterparts in a historic Oracle Table.** The Oracle Table is a critical data structure used in synthesising more detailed models of the implementation. It is further explored in Section 4.3.

**(5) Response packets from the target implementation to the reference implementation are abstracted and sent back to the learner.** Just as the adapter must be able to concretise packets, it must be able to abstract them back to the same abstraction level as the original request it received, so that they can be sent back to the learner in a matching abstraction level.

Although the specific ways of achieving these properties vary from an implementation to another, we have found this requirement set is often enough to strike the balance in allowing the learner to interact with the SUL while minimising behaviour introduced by the reference implementation.

In some more complex protocols, it may also be useful to store in a queue formulated payloads that would break rule (1) by reacting to a received response packet, so that they can be sent later if requested by a matching abstract counterpart, as exemplified in Listing 1.

Our modified code will have instrumentations at all the points where packet types and packet frames are constructed, and ensure through the instrumented conditional branching that only matching concrete packets are sent to the target implementation. We also add code to be able to hook into functionality allowing us to create packets from scratch.

As an example, consider the Adapter receives an abstract query `INITIAL[ACK]` for a QUIC packet (where `INITIAL` is one of the 7 packet types and `ACK` is one of the 20 frames carried in packets).

Firstly, the queue is searched for a matching packet to be sent. If there is no match for this packet, one is made from scratch, with the valid current state of the reference implementation (Packet Number, connection details, and others), and sent instead. The reference implementation may then receive packets as a response, which are processed by the reference implementation as usual to make sure its state is updated, and if the reference implementation logic would cause a new packet to be sent in response, it is instead stored in our queue waiting for the learner, as demonstrated below.

**Listing 1: Enforcing (1) and (2) in ACK sending logic**

```
  if (packet.shouldBeAcked) {
    // Received a packet to be acked
-   connection.sendPacket(INITIAL, ACK)
+   AckQueue += packet
  }
+ if (requestedInitialAck) {
+   // Abstract Symbol was requested
+   connection.sendPacket(AckQueue.find(INITIAL, ACK))
+ }
```

After this, the original response from the target implementation is abstracted, and sent to the learner:

**Listing 2: Implementing (4) and (5) in packet send/receive hooks**

```
+ // Define abstract symbol according to set abstraction.
+ abstractSymbol = newAbstractSymbol(
+   packetType, version, packetNumber, frameTypes)
  (...)
+ // Save symbol exchange in Oracle Table.
+ oracleTable.addIOs(abstractInputs, abstractOutputs,
+                    concreteInputs, concreteOutputs)
+ // Return abstract response to learner.
+ learner.send(abstractOutputs)
```

Finally, at the end of each query the Learner requests that the SUL reset its state for a new query to be made. In the specific case of QUIC, it is enough to reset the Adapter (client), and start a new connection to the Implementation:

**Listing 3: Implementing (3) in main Adapter body**

```
+ // Define reset routine.
+ func reset() {
+   agents.stopAll()
+   connection.close()
+   connection = newConnection(conncetion.getConfig())
+   agents = defaultAgentsWithConnection(connection)
+ }
```

Most importantly, by using a reference implementation we can rely on it to maintain the desired state in between multiple packets—i.e., when a new abstract symbol request comes in, we can simply resume from the state we left at the previous abstract symbol. This last aspect is what allows us to not have to explicitly model any of the protocol-specific logic, as it was done in prior work [22]. To quantify this aspect, our instrumentation of the reference implementation of the TCP protocol only required an additional 300 lines of mostly boilerplate code (which we envision can be automated in the future), while the mapper implemented Fiterău-Broştean et al. [22] to model the concretization function consisted of 2,700 lines of code modeling the complex logic of TCP. For QUIC, our instrumentation required just over 2,000 lines of again mostly boilerplate code. Given the complexity of QUIC (which involve cryptography and other complex features), it is hard to imagine replicating the approach from [22].

To make the approach more effective, we introduce a few optimizations. First, whenever the reference implementation sends new packets (as part of an intermediate step) that do not match our abstract query, we store such packets in a list and use them to try to answer future queries. That is, when an abstract query comes in, we first look into this list to check if any of the packets has the desired abstract value and, if this is the case, we send that packet to the Implementation. Second, we cache all intermediate pairs between abstract I/Os and concrete I/Os in a data structure called an Oracle Table $O \subseteq \{(a, c) \mid a \in (\widehat{\Sigma}^* \times \widehat{\Gamma}^*), c \in (\Sigma^* \times \Gamma^*)\}$.

In Section 4, we use this cache to synthesize richer models that go beyond finite abstract alphabets and can capture concrete packet numbers and other numerical quantities. We keep track of received packets for each query and also use this record to detect retransmitted packets that should not be part of the response due to nondeterminism.

Now that we have constructed an abstraction that produces a simplified abstract alphabet, and we have effective ways of querying over this alphabet, we have built the interface necessary to interact with the SUL module. We could then, for example perform our 3-way TCP handshake by sending the input trace `SYN(?,?,0)` `ACK(?,?,0)` and we would get the output trace `ACK+SYN(?,?,0)` `NIL`, which accurately represents the flags of our 3-way TCP handshake. However, because it uses our abstract alphabet the Adapter cannot tell us the exact sequence or ack numbers it picked as this would cause nondeterminism.

# 4 LEARNING MODULE

We now describe how the learning module interacts with the SUL to learn models of the implementation. In Section 4.1, we formally define the queries the learner is allowed to ask to construct a model. In Section 4.2, we recall existing synthesis techniques for learning Mealy Machines (i.e., automata with outputs). In Section 4.3 we use program synthesis to learn a more detailed model that is capable of recovering register values and changes, like sequence and acknowledgement numbers in the TCP protocol.

## 4.1 Learner Interface

Thanks to the techniques presented in Section 3, the SUL can be treated as a *query oracle* that can answer the question "*If I send this input sequence, what will the implementation return?*". With this deterministic query oracle we have all we need to create an interface for many model-learning algorithms for finite state machines. In particular, we can use the query oracle to implement (or at least approximate) two types of queries that a learner can ask:

**Membership Queries:** $a_O = mq(a_I)$? where $a_I \in \widehat{\Sigma}^*$ and $a_O \in \widehat{\Gamma}^*$. These are single traces $a_I/a_O \in \widehat{\Sigma}^* \times \widehat{\Gamma}^*$ of the system that give the Learner knowledge about the specific traces produced by the SUL so that it can build a hypothesis model $H$.

**Equivalence Queries:** $eq(H)$? where $H$ is a hypothesis model the Learner believes to be correct and wants to know if it is equivalent to the SUL; the answer is either $a_I/a_O$, a counterexample trace that distinguishes the SUL from the hypothesis $H$, or no counterexample is returned, and the model is considered a correct abstraction of the SUL, terminating the learning process.

In practice, Equivalence Queries require an oracle omniscient of the SUL, and if we had that, we would not have to learn the system in the first place. Instead, we can use heuristic Equivalence Oracles such that when a counterexample is returned, it is guaranteed to be a valid counterexample, but the absence of a counterexample no longer guarantees equivalence. This approach still gives us approximation guarantees—i.e., the model is accurate with high probability with respect to the set of inputs we use to test equivalence. It is now a good point to remind the reader that the goal of this paper is to unveil and discover potential incorrect behaviors of the SUL, rather than provide behavior guarantees. Even if the learned models might not be 100% accurate, they will still be helpful to analyze and detect anomalies as we will show in our evaluation.

Learners that depend only on these two types of queries were first studied in [11] to learn deterministic automata and have since been extended to many types of state machines.

## 4.2 Learning Mealy Machines

With oracles capable of answering membership and equivalence queries, we can use existing algorithms [25] to learn Mealy machines of the abstract behavior of the SUL. Intuitively, a Mealy machine is a finite automaton that for every input symbol it reads, it also produces an output.

*Definition* 4.1 (Mealy Machine). A Mealy Machine is a tuple $(S, S_0, \widehat{\Sigma}, \widehat{\Gamma}, T, G)$, such that: $S$ is a finite set of states, $S_0 \in S$ is the initial state, $\widehat{\Sigma}$ is the abstract input alphabet, $\widehat{\Gamma}$ is the abstract output alphabet, $T$ is the transition function $T\colon S \times \widehat{\Sigma} \to S$, and $G$ is the output function $G\colon S \times \widehat{\Sigma} \to \widehat{\Gamma}$.

*Example 4.1.* The Mealy machine in Fig. 3(b) is a model of the TCP 3-way handshake over input and output alphabets:

$\widehat{\Sigma}$ = {`SYN(?,?,0)`, `ACK(?,?,0)`} $\widehat{\Gamma}$ = {`ACK+SYN(?,?,0)`, `NIL`}

Given the input sequence [SYN(?,?,0), ACK(?,?,0)], this machine outputs the sequence [ACK+SYN(?,?,0), NIL]—starts in state $s_0$, when reading SYN(?,?,0) it transitions to state $s_1$ and outputs ACK+SYN(?,?,0), and then reading ACK(?,?,0), it transitions to $s_2$ and outputs NIL (no packet).

Mealy machines have been studied extensively and there are many algorithms that can learn them using membership and equivalence oracles [25]. At the high level, these algorithms issue membership queries to discover the behavior of the machine until they can find a machine that is consistent—i.e., it correctly matches all the traces for which it has issued membership queries. At this point, the algorithm issues an equivalence query, which can either end the learning process (in case of a yes answer) or cause the learning process to ask more membership queries and repeat this process.

Prognosis uses the TTT algorithm [25] which is guaranteed to learn a Mealy machine in time polynomial in the size of the machine. For example, when we run TTT on the TCP implementation using the abstract alphabet in Example 3.3, TTT learns the model in Fig. 3(b) (we depict only transitions relevant to the handshake, though the learned model is deterministic and total).

## 4.3 Synthesizing Rich Models

Mealy Machines can only model operations involving finite alphabets and cannot reason about numerical values—e.g., sequence numbers. To capture the packet exchange in Figure 3(a) we need not only the TCP flags currently learned in the model depicted in Figure 3(b) but also certain quantities[1]—sequence and acknowledgement numbers.

In this section, we present an extension of Mealy machines that adds registers, and numerical inputs and outputs. We then show how the ideas in Section 4.2 can be combined with constraint-based synthesis techniques to learn these extended models. While we could consider a wide range of enhancements to traditional automata, we limit ourselves to extending automata to read and write integer values from packets, and to increment or set to input values one of a finite number of registers $\vec{x}$. These extensions capture common features of the protocols we are interested in. A transition of an extended Mealy machine looks like the following:

$$p \xrightarrow[\vec{x}=\mathbf{u}(\vec{i},\vec{x})]{I(\vec{i})/O(\mathbf{o}(\vec{x}))} q$$

Informally, if the machine is in state $p$ and reads an abstract symbol $I$, possibly parametric on concrete numerical values $\vec{i}$, it updates the registers $\vec{x}$ with values determined by $\mathbf{u}(\vec{i},\vec{x})$, and it outputs an abstract symbol $O$ parametric on the values determined by $\mathbf{o}(\vec{x})$. The update function $\mathbf{u}(-)$ can take on quite complex values, but in the following we consider each register is updated with either a copy of a register, or an input value, or one of these incremented by 1. Similarly, the output function $\mathbf{o}(-)$ can, for each parameter, output the value of a register or that value plus 1.

We start from a Mealy machine where the register updates and outputs are missing (as in Figure 4(left)) and our goal is to find concrete terms $\mathbf{u}_1, \cdots, \mathbf{u}_9$, and $\mathbf{o}_1, \mathbf{o}_2$ for each transition that result in an extended machine correctly modelling the SUL with respect to a given set of concrete traces.

We use the set of traces cached while learning the Mealy machine in the Oracle Table $T$. For each pair $(a, c) \in T$ of abstract and concrete traces, we identify what path of the extended Mealy machine is traversed by the abstract trace $a$ and use the concrete trace $c$ to generate the constraints needed to identify the missing terms of each transition in the path—i.e., the terms that make the extended machine consistent with the concrete trace $c$. If needed, the algorithm can solicit more example traces and add them to $T$. The constraints are then solved using an SMT solver and the solution is used to generate the needed terms.

Let us illustrate how we would synthesize the extended machine in Figure 4(right) from the the sketch in Figure 4(left). We consider the following concrete trace appearing in $T$: [(ACK(0, 3, 0)/NIL), (SYN(2, 5, 0)/ACK(4, 5, 0))]. This time, we want to model that each symbol in the concrete trace also carries the synchronization number (sn) and the acknowledgement number (an)—e.g., for the first input these values are 0 and 3, respectively. Given this trace, our algorithm generates constraints containing a number of variables used to denote what we are trying to synthesize and what it means for the solution to be correct with respect to the input trace. For each unknown term, we have a finite list of possible terms we can instantiate it with. For example, the unknown $\mathbf{u}_1$ can be instantiated with one of the 8 terms in the list: $[r, r+1, pr, pr+1, pi, pi+1, \text{sn}, \text{an}]$. In our constraints, we use an integer variable $E_{\mathbf{u}_1}$ to indicate the possible choices (indices start at 0). For example, $E_{\mathbf{u}_1} = 1$ indicates that the term $r + 1$ will be the solution for the unknown $\mathbf{u}_1$.

As register values will change for each trace, our constraints need to model how the values of the register are updated throughout the execution. To do so, we introduce variables that track the values of each register after reading the $i$-th input in the trace. For example, $r[i]$ indicates the value stored in register $r$ after reading the first input packet. When generating constraints for multiple traces, we will have a variable $r_\pi[i]$ for each trace $\pi$ and index $i$.

The following simplified set of constraints capture the synthesis problem we are trying to solve:

// Constraints for $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4, \mathbf{u}_5, \mathbf{u}_6$
$0 \le E_{\mathbf{u}_1} \le 7$ // The value of $E_{\mathbf{u}_1}$ encodes the 8 possible terms for $\mathbf{u}_1$
$E_{\mathbf{u}_1} = 0 \implies r[1] = r[0]$ $\quad E_{\mathbf{u}_1} = 1 \implies r[1] = r[0] + 1$
$E_{\mathbf{u}_1} = 2 \implies r[1] = pr[0]$ $\quad E_{\mathbf{u}_1} = 3 \implies r[1] = pr[0] + 1$
$E_{\mathbf{u}_1} = 4 \implies r[1] = pi[0]$ $\quad E_{\mathbf{u}_1} = 5 \implies r[1] = pi[0] + 1$
$E_{\mathbf{u}_1} = 6 \implies r[1] = 0$ $\quad E_{\mathbf{u}_1} = 7 \implies r[1] = 3$
...
// Constraints for $\mathbf{o}_1, \mathbf{o}_2$
$0 \le E_{\mathbf{o}_2} \le 5$ // The value of $E_{\mathbf{o}_2}$ encodes the 6 possible terms for $\mathbf{o}_2$
$E_{\mathbf{o}_2} = 0 \implies r[2] = 5$ $\quad E_{\mathbf{o}_2} = 1 \implies r[2] + 1 = 5$
$E_{\mathbf{o}_2} = 2 \implies pr[2] = 5$ $\quad E_{\mathbf{o}_2} = 3 \implies pr[2] + 1 = 5$
$E_{\mathbf{o}_2} = 4 \implies pi[2] = 5$ $\quad E_{\mathbf{o}_2} = 5 \implies pi[2] + 1 = 5$

Note that setting $E_{\mathbf{u}_1} = 7$ corresponds to selecting the term an (i.e., the input ACK number) as a solution to the unknown $\mathbf{u}_1$. In this case, the constraints model that the value of the register $r$ after reading the *first* packet (i.e., $r[1]$) should be equal to the ACK number of the *first* input packet (i.e. 3). Similarly, setting $E_{\mathbf{o}_2}$ to the value 4 corresponds to selecting the term $pi$ as a solution to the unknown $\mathbf{o}_2$. In this case, the constraints model that the value of the register $r$ after reading the *second* packet (i.e., $r[2]$) should be equal to the ACK number of the *second* output packet (i.e., 5).

[1] Existing extensions of the learning algorithm that handle automata with counters [25] do not meet the needs described in this paper as they do not support complex comparison operations and updates.

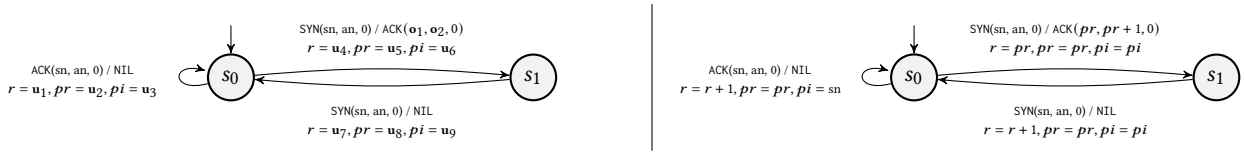


**Figure 4: Extended machine with unknown terms (left) and corresponding synthesized machine (right).**

Repeating this process on more examples, e.g $[(\texttt{SYN}(2,3,0)/\texttt{ACK}(4,5,0)), (\texttt{SYN}(2,3,0)/\texttt{NIL})]$, yields the automaton in Figure 4(right). This automaton corresponds to the solution that selects $E_{\mathbf{u}_1} = 1$ and $E_{\mathbf{o}_2} = 3$.

While this section shows a particular type of model, our framework is general and allows to implement other synthesis and learning algorithms for more complex models (e.g., allowing more complex constraints and updates) due to its interaction with the SUL.

Sometimes the synthesized models can contain incorrect register patterns if the pattern is not completely covered by the Oracle Table traces. These are detected through random equivalence testing, and trigger new queries in the synthesis algorithm—i.e., the synthesizer will restart with a larger set $T$ of traces to learn for as well as negative example—i.e., traces that the model should not contain. The constraints discussed in this section can be easily adapted to handle negative examples.

## 5 ANALYSIS MODULE

The last module of Prognosis enables analysis techniques using the outcomes of the learning modules to help the user infer behaviors of the SUL. Different abstraction levels allow us to expose different types of anomalies, however the analysis module is limited to uncovering logic errors captured in the model, which is restricted to observable events captured in the learning process. Some more nuanced quantity specific bugs can be analysed through the synthesized model, however these are limited to linear patterns.

We focus on analysing models for bug finding and knowledge acquisition rather than providing verified guarantees due to not all abstraction levels capturing enough information to fully verify the specification. In this section, we present some of these analyses and how they can be used to identify undesired behaviors.

*Nondeterminism Check.* As explained in Remark 3.1, the learning module typically expects deterministic behavior (i.e., the same input trace should trigger the same output trace). In general, nondeterminism can arise in our system for two reasons: (1) the abstract alphabet is so simplified that concrete traces corresponding to different behaviors are collapsed into the same input trace, or (2) the implementation is producing nondeterministic outputs in cases where it should not. For every Learner query, Prognosis expects a deterministic answer, as it is attempting to build a deterministic model. However, due to the nature of active-learning, environmental events such as latency and packet loss could cause non-determinism to be observed. To avoid this Prognosis will execute the query a specified minimum number of times, and if these are not all the same, it will continue to do so until a specified percentage of certainty is achieved. If after a limit number of queries nondeterministic is discovered, the learning process pauses, and the Adapter verifies the cause of the nondeterminism. If reason (1) is the cause, the user will see that the abstraction is too coarse and can provide a richer abstraction. While it is possible to use more complex learning algorithms that can handle nondeterminism in the traces, we argue that, because of reason (2), detecting sources of nondeterminism is a powerful analysis technique that can shed light on some undesired behaviors. As we show in Section 6.2.4, Prognosis was able to unveil a complex bug in a QUIC implementation thanks to the nondeterminism check.

*Learned Model Analysis.* If learning succeeds, Prognosis can produce visualizations of the models which can help a user understand whether an implementation works as expected. In addition, Prognosis also exposes various algorithms and decision procedures to check properties of the learned models. For example, Prognosis can check equivalence of models of two different implementations of the same protocol (whether they accept the same input/output traces). For Mealy machines, there are algorithms for performing this equivalence check efficiently [24]. Prognosis can produce concrete example traces that show the difference between the behaviors of the two implementations. We show in Section 6.2.5 how Prognosis is able to unveil a complex bug in a QUIC implementation thanks to these equivalence checks, and the visualizations of the differences were instrumental in communicating the problem to the developers.

Prognosis also allows the user to specify temporal properties in logics such as LTL or CTL, e.g., *Packet numbers are always increasing*, and check whether the models satisfy such properties. For Mealy machines, this procedure simply boils down to checking that the traces of the model are a subset of those allowed by the property, a decidable problem [24]. For extended machines with counters, this problem is in general undecidable and we rely on randomised testing to make the analysis practical.

## 6 RESULTS

We evaluate Prognosis using two case studies. First, we show that Prognosis can replicate prior works on learning models of TCP implementations [22] (Section 6.1). Second, we use Prognosis to learn models of different implementations of the IETF QUIC protocol, which is undergoing standardization after successful initial development at Google (Section 6.2). Overall, we evaluate Prognosis' effectiveness in identifying unintended behavior in real protocol implementations, as well as its modularity and reusability.

Prognosis was implemented using several programming languages. The abstract learner is implemented in 3,500 lines of Java and uses the automata learning library LearnLib [25]. The extended Mealy machine synthesizer is implemented in 300 lines of Python and uses the Z3 SMT Solver[16].

## 6.1 Learning a TCP Implementation

In this case study, we demonstrate how Prognosis can reproduce results from previous work on learning models of TCP implementations [22]. We learn the *Ubuntu 20.04.1 LTS* TCP stack, particularly kernel version `5.8.0-40-generic`.

We provide Prognosis with the same abstract alphabet $\widehat{\Sigma}$ and translation pair $(\alpha, \gamma)$ used in prior work [22]. Concretely, the abstract alphabet models the TCP flags for the packet, leave the sequence and acknowledgement numbers unspecified, and provide the payload length.

$$\widehat{\Sigma} = \{\texttt{SYN(?,?,0)}, \texttt{SYN+ACK(?,?,0)}, \texttt{ACK(?,?,0)}, \texttt{ACK+PSH(?,?,1)}, \texttt{FIN+ACK(?,?,0)}, \texttt{RST(?,?,0)}, \texttt{ACK+RST(?,?,0)}\}$$

To implement the abstraction function $\alpha$ and the concretization function $\gamma$ we need a reference implementation. To illustrate the flexibility of Prognosis, we use the mapper implementing the TCP logic designed in [22] as reference and instrument it. Note that we could have used any other TCP implementation for this task. Modifying the reference implementation required only 300 lines of code to integrate with *Scapy* [8] instead of the >3000 lines of code required to implement a concretization function in [22].

The model learned by Prognosis has 6 states and 42 transitions and took 4,726 membership queries to learn. This model is slightly different to the one learned by Fiterău-Broştean et al. [22] due to major differences made to the implementations over time. Unfortunately, despite multiple attempts, we were not able to run the prior technique by Fiterău-Broştean et al. [22] on the new TCP implementation and we cannot see what model it will learn now. This case study and the difficulty of adapting prior work to new implementations showed that we could reproduce the level of detail obtained in prior work with very minor effort thanks to the modular components of Prognosis.

While we do not perform any analysis in this case study, we show that our synthesis procedure can help us derive richer models than those considered in prior work. In particular, using the technique described in Section 4.3, we could synthesize a model describing the behavior of extra parameters such as the `dataOffset`. Remarkably, applying this richer analysis only required changing one line of code in the implementation—simply selecting which concrete values of the TCP implementation we wish to synthesize over.

## 6.2 Learning QUIC Implementations

In this section, we analyze 4 implementations of the QUIC protocol. We devise the abstract alphabet $\widehat{\Sigma}$, provide a reference implementation for the translation pair $(\alpha, \gamma)$, and a set of properties from the QUIC specification to analyze. We show how Prognosis detected behavior that led to a change in the QUIC specification, as well as identified bugs in real implementations of the QUIC protocol.

*6.2.1 Background.* The QUIC protocol [31] combines specific features of independent protocols that are commonly used together, reducing the overhead that arises from having to coordinate these features. Specifically, QUIC is optimized for the Web, and as such, collapses the services provided by HTTP, TLS, and TCP into a single *super protocol*. Therefore, QUIC is extremely complex, and its design has been carefully thought out to ensure these services that were previously isolated can be communicated efficiently. QUIC achieves this goal using encapsulation. In TCP, a single packet is enough to communicate everything needed, from specific signals like `SYN` or `ACK` to the payload of the application layer itself. In QUIC, both signaling, and data transmission happen in *frames*. A packet serves merely as a mean of safely transporting different types of frames. In total, QUIC provides 7 packet types and 20 frame types, each responsible for signaling a specific aspect of the protocol. We will not be diving into the details here, due to their complex nature, and instead, we will introduce specific frames and packets as needed to explain key properties of the protocol and defer the reader to the QUIC specification [31] for further details. We will consider the following QUIC implementations:

**Quiche** is Cloudflare's QUIC implementation [1] that allows QUIC connections to any website protected by Cloudflare network. Cloudflare's CDN supports > 25 million websites.

**Google's QUIC** implementation [3] runs on Google servers and on Chromium browsers. Since October 7 of 2020, 25% of Google Chrome users had QUIC support enabled by default, with that proportion increasing over the followed weeks.

**Facebook's mvfst** QUIC implementation [2] is used mainly in the Proxygen server implementation. This server implementation is responsible for powering most public facing connections to facebook.com, as well as API connections used by the Facebook and Instagram mobile apps.

**QUIC-Tracker** is an implementation [28] designed to run testing scenarios over other implementations, with the goal of testing what technologies different implementations support.

Because these first three implementations account for a large portion of the web traffic, identifying erroneous behaviors in any of them is of critical importance.

*6.2.2 Learning Models of QUIC Implementations.* We use Prognosis to learn models of the first three QUIC implementations described above and analyze a subset of the properties from IETF's Draft 29 [31]—e.g., *The sequence number on each newly-issued connection id must increase by 1* and *An endpoint must not send data on a stream at or beyond the final size*. Prognosis requires 3 things from the user: An abstract input alphabet, a translation pair $(\alpha, \gamma)$ to be able to convert between abstract and concrete alphabets, and optionally, a set of properties $\Phi$ that we would like to test. We use an abstract input alphabet $\widehat{\Sigma}$ with 7 symbols:

$$\widehat{\Sigma} = \{\texttt{INITIAL(?,?)[CRYPTO]}, \texttt{INITIAL(?,?)[ACK,HANDSHAKE\_DONE]}, \texttt{HANDSHAKE(?,?)[ACK,CRYPTO]}, \texttt{HANDSHAKE(?,?)[ACK,HANDSHAKE\_DONE]}, \texttt{SHORT(?,?)[ACK,MAX\_DATA,MAX\_STREAM\_DATA]}, \texttt{SHORT(?,?)[ACK,STREAM]}, \texttt{SHORT(?,?)[ACK,HANDSHAKE\_DONE]}\}$$

The first four symbols are used to open the connection, perform the handshake, and transmit data, and the last three symbols allow us to model properties related to flow control. We focus on this alphabet, as due to QUIC's numerous packet and frame types, choosing to learn the behavior of, for example, all packet types containing up to 3 frames, would already give us an alphabet with over 30,000 symbols. Although, in theory, we could learn over such large alphabets, the learning process would take an infeasible amount of time. As we will show in the rest of the section, these seven symbols are enough to capture the main connection establishment, handshake, data transmission, and flow control behavior.

To build the translation pair $(\alpha, \gamma)$ in the Adapter, we use *QUIC-Tracker* [28] as a reference implementation and instrument it to convert between abstract and concrete symbols. Specifically, we have implemented the format we picked for our abstract alphabet, and whenever our Adapter receives an abstract request from the learner, we use QUIC-Tracker's state to determine if there is a valid concrete packet matching the abstract packet that should be sent from the current state. If so, we send this queued packet, if not, we request that QUIC-Tracker sends a new packet matching our abstract requests, filling in the concrete values required to make it a valid packet under the current state. QUIC-tracker then encodes and sends the created packet to the Implementation we are learning a model for and waits for a response. Once this response is detected, it is abstracted and sent back to the learner as a response.

The QUIC Adapter is comprised in total of over 10,000 lines of Go for its reference implementation, however the instrumentation requires only an additional 2,000 lines of code, which we believe is a relatively low cost for Prognosis to handle a protocol as complex as QUIC. Furthermore, the same instrumentation can be used to learn models of all the QUIC implementations. When ran on these three implementations Prognosis could learn models for two of the three implementations (see Section 6.2.4 for explanation). The two models had 12 and 8 states, and 84 and 56 transitions, respectively. Learning took 24,301 queries for one implementation and 12,301 queries for the other. Having these two models enables reasoning with much fewer traces: for the alphabet $\widehat{\Sigma}$ above there are 329,554,456 traces of length up to 10, however we only need to check 1210 and 715 of those traces, respectively, for the two implementations. This reduction is gained because the traces in the learned model are a subset of all traces that can be made with the alphabet.

In the rest of the section, we describe what issues were unveiled in the QUIC RFC and implementations using a number of formal and informal analysis of the models generated by Prognosis. As some of the problems we have identified introduce serious security vulnerabilities that the developers of the protocols have not yet addressed, we will not identify in which specific implementations the issues were found.

#### 6.2.3 Issue 1: RFC Imprecision.

**Analysis technique and outcome:** When comparing the models learned for different implementations, we discovered that these models, as reported in Section 6.2.2, had vastly different sizes. We manually explored the models to identify the key differences in structure and found that different implementations were not consistent on what to do if a client resets the *Packet Number Spaces* when retrying a connection. We reported this anomaly to the IETF QUIC Working Group who issued a fix [5] on the next version of the specification to make the expected behavior clearer.

**Underlying issue:** As QUIC is a protocol undergoing standardization, its RFC is a living document that is still being perfected. One of the guiding principles for this document is the Robustness Principle [29] that was already employed in the development of TCP. The principle states that a system should be strict about what it sends, and liberal on what it receives. As such, the QUIC RFC is perhaps a looser specification than those used for formal verification, and certain aspects of the specification are ambiguous.

Our reported inconsistency unveiled an ambiguity in the behavior of `RETRY` packets. `RETRY` packets are a special kind of packet used by a server to verify the source address of the received packet. It is called `RETRY` as the client should retry to open the connection with the `RETRY` token received from the server. Most importantly, our **report led to a discussion** on the topic by the RFC maintainers, and **a fix was issued** to clarify that a server *MAY* abort the connection when a client resets their Packet Number Spaces [5].

#### 6.2.4 Issue 2: Nondeterminism in Connection Closure.

**Analysis technique and outcome:** As described in Section 5, Prognosis ensures that the answer to every Learner query has a deterministic response via the nondeterminism check. During this check, we found that it was possible to have Facebook's mvfst close the connection and remain in a state where it will not always respond with `RESET`s to subsequent packets.

**Underlying Issue.** For backwards compatibility reasons, QUIC packets are carried over the UDP protocol. While TCP has been vastly optimized in devices ranging from internet middle-boxes such as switches and routers to endpoint devices such as servers and clients, UDP has not received the same level of attention, mostly due to TCP being the *de facto* protocol for the transport layer.

To understand this bug, we need to introduce a specific QUIC frame: `HANDSHAKE_DONE` is a signaling frame sent by the server (and only the server) to notify the client that the handshake is now complete, and it can proceed to transmit its data. We include this frame in the input alphabet used by the Learner, and as such the learned model will depict what happens in case the client sends this server-only frame at different states of the system. The ideal response is that, as this frame should never be sent by a client, the server would treat receiving this frame as a protocol violation error, and immediately close the connection, and any further packets sent on the same connection *may* be met with a `RESET` packet, a decision that is up to the developer to make. The `RESET` packet type is a last case resort that an endpoint can use to notify the other side that this connection no longer exists, even when data transmissions keys are no longer available.

Prognosis detected that if the client starts a connection as usual with `INITIAL(?,?)[CRYPTO]`, and instead of finishing the handshake, sends a `HANDSHAKE_DONE` frame to the server with the `HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE]` abstract symbol, this packet will be met with a `CONNECTION_CLOSE` frame, effectively closing the connection. However, after testing the same packet sequence repeatedly, it found that if the client keeps sending packets, it is only in 82% of the responses that following packets are met with a `RESET`.

This behavior discovered in our analysis is **erroneous**: a specific implementation may choose to use `RESET` packets, but it must be **consistent** in the decision. It cannot nondeterministically switch between sending and not sending packets. Furthermore, this RESET behaviour has no back-off mechanism, meaning that a client can exploit this bug to request new packets from the server on demand. The client could keep sending the exact same packet, with no computation needed, and the server would have to produce new `RESET` packets every time it receives this unexpected packet. This behavior, coupled with the fact that `RESET` packets are relatively

small, means that system-level UDP optimizations are not triggered, resulting in sending each RESET packet being an expensive operation. If exploited, this behavior can be used in a *Denial-of-Service* attack against the server. This critical bug has been acknowledged by the developers and will be fixed soon.

#### 6.2.5 Issue 3: Inconsistent Port on RETRY in QUIC-Tracker.

**Analysis technique and outcome:** After learning the model of one of the implementations we detected a discrepancy (in terms of traces) in how two different implementations handle the *Retry Mechanism*: if the server sent a RETRY packet the model would transition to a state where connection establishment was impossible. This greatly contrasted with behavior learned in another model.

One of the properties of a confirmed erroneous trace is that either the Implementation or the Adapter are behaving unexpectedly. In this case, it was the reference implementation, QUIC-Tracker, that had a bug in its retry mechanism, and only one of the target implementations considered it critical enough to not allow connection establishment to take place.

**Underlying Issue:** QUIC's Retry Mechanism, as introduced in Section 6.2.3, is a way for the server to validate that a packet is indeed being sent from a specific IP address and port, and not just an attacker replaying packets from a spoofed source. The client starts by sending a `ClientHello` as usual, but this time the server replies to it with a RETRY packet. This packet has a unique token which the client must take and send in a new `ClientHello` from the same IP address and port. The server then checks if the `Retry Token` matches the one previously sent to confirm these packets are being sent from the claimed source. Our reference implementation was correctly returning the token; however it was doing so with a new UDP socket using a random free port instead of the one we were using before. As such, the token would be sent from a different port, and address validation would fail, interrupting the handshake.

#### 6.2.6 Issue 4: Stream Data Blocked Bug in Google QUIC.

**Analysis technique and outcome:** Using the extended models of Section 4.3, we synthesized an extended Mealy Machine (shown in Appendix **??**) that describes how the `Maximum Stream Data` field in the STREAM_DATA_BLOCKED frame changes over the states. By inspecting this model we were able to detect that this field always has the value `0`, and is never updated, even when the stream gets blocked.

**Underlying Issue:** One of the key features of QUIC is flow control. For this, QUIC makes use of a series of frames dedicated to coordinating flow limits between the two endpoints. One such frame, the STREAM_DATA_BLOCKED frame, is responsible for alerting the other endpoint when we would like to send data on a stream, but the flow control limits imposed on us by the endpoint itself do not allow us to do so. This frame has two fields, a `Stream ID` field indicating which stream is blocked, and a `Maximum Stream Data` field, indicating at what offset of the data the block happened. We found that although one of the implementations used these frames and did not transmit data over the agreed limits, it did not set the `Maximum Stream Data` correctly, and instead used the constant `0`. The developers have confirmed that this section of the specification was incorrectly implemented. They explained that `0` was a placeholder set during initial development, which they had forgotten to update.

## 7 RELATED WORK

Model learning has been applied to a range of communication and security protocols [9, 10], including network protocols [12, 17, 18, 22]. Prognosis improves on previous work in several directions: first, the architecture of Prognosis is parametric with respect to reference implementation, an aspect that greatly decreases the amount of work and expertise needed to analyze one protocol implementation; second, Prognosis allows a user to swap different implementations of the same protocol seamlessly by requiring only a socket change; third, Prognosis uses synthesis to enrich the models with data and enables a more fine-grained control in the type of properties and details one can analyze. Prognosis is the first model-learning tool used to analyze several QUIC implementations.

Other approaches to analyze QUIC include recent work by McMillan and Zuck [27] who manually build a formal specification of the wire protocol in the Ivy language, and then use it for test generation and find a range of bugs in different implementations. In contrast, the approach used in Prognosis automates the building of a finite state model guided by an abstract alphabet. The models learned by Prognosis can also be used for test generation, and crucially one does not need to manually encode the protocol logic, including complex cryptographic components, to use Prognosis. McMillan and Zuck [27] specify the full protocol state including the security handshake, with unbounded data. The models we learned with Prognosis abstract away details not covered by the alphabet or the registers we chose to synthesize. Closely related to [27], but only for TCP, is the work of [15] using symbolic model checking.

Other approaches to analyze protocol implementations include building a *correct-by-construction* reference implementation [14, 19] and then use it when testing a new implementation. This approach has the advantage of having every component of the protocol in the reference implementation formally verified. However, the amount of manual work to build such implementation is out of reach of most development teams and, moreover, it will be making specific choices in terms of the RFC specification.

Prognosis is closed-box—it does not assume access to the code of the implementation being analyzed. Open-box approaches to analyze QUIC also appeared in the literature [23, 30]. These works focus on testing protocol compliance using symbolic execution. However, subtle bugs related to ambiguities in the RFC or differences between implementations (like the ones detected by Prognosis and described in Section 6) would not be detected easily.

Finally, Prognosis complements differential testing [26]: the learned models as well as the Adapter can be used to create high-quality test cases that trigger complex behaviors of the protocols, something that is typically hard in a closed-box setting. Furthermore, Prognosis produces models that can be inspected by protocol designers, something that differential testing cannot help with.

In summary, our work focuses on developing a modular, reusable, and flexible framework, where different formal analysis techniques can easily be made accessible for different protocols and implementations thereof.

## 8 CONCLUSION

Prognosis is a modular framework to automatically learn models of network protocol implementations and analyze them. Prognosis has been successfully used to analyze TCP and QUIC implementations and has found several bugs in different mainstream QUIC implementations. One of the key contributions of Prognosis is the use of a reference implementation to remove the burden of having to implement a protocol logic from the user. This step was required in previous approaches based on model learning. Though this contribution means that less experienced users have easier access to our framework, Prognosis still requires some knowledge on how to instrument the reference implementation.

One direction for further work is investigating whether there are semi-automated ways of aiding the user of Prognosis in finding the key places where the code needs instrumentation. The use of *active* model learning is core to Prognosis as the precision of the learned model can be adjusted on demand. However, in cases where access to logs is possible, and to avoid resorting to so many expensive queries, the learning process could be speeded up using a combination of *passive* and *active* learning.

The models Prognosis can currently synthesize do not encompass any form of *environment quantities*, which are interesting in a networking context to capture e.g., congestion, latency, or memory usage properties. Extending Prognosis to richer quantitative models is perhaps the most challenging yet impactful direction for future work, as it will require significant advances on the design of learning algorithms. Recent developments in active learning of weighted automata [13, 33] provide a good starting point for this direction. Learning quantitative properties will also require an enriched Adapter that can answer quantitative queries.

## 9 ACKNOWLEDGMENTS

Ferreira and Silva were partially funded by ERC grant AutoProbe (101002697). D'Antoni was partially funded by Facebook research awards, a Microsoft Faculty Fellowship, and an Amazon Research Award. The authors would like to thank Hongqiang Liu for his help in preparing the final version of the paper.

## REFERENCES

[1] 2021. cloudflare/quiche. (Jan. 2021). https://github.com/cloudflare/quiche original-date: 2018-09-29T18:22:05Z.
[2] 2021. facebookincubator/mvfst. (Jan. 2021). https://github.com/facebookincubator/mvfst original-date: 2018-04-09T22:49:15Z.
[3] 2021. Google QUIC - The Chromium Projects. (Jan. 2021). https://www.chromium.org/quic
[4] 2021. Heartbleed - CVE-2014-0160. (Jan. 2021). https://nvd.nist.gov/vuln/detail/CVE-2014-0160
[5] 2021. Let server abort on post-Retry packet number reset by dtikhonov · Pull Request #3990 · quicwg/base-drafts. (Jan. 2021). https://github.com/quicwg/base-drafts/pull/3990 Library Catalog: github.com.
[6] 2021. NVD - CVE-2017-1000253. (Jan. 2021). https://nvd.nist.gov/vuln/detail/CVE-2017-1000253
[7] 2021. NVD - CVE-2018-5390. (Jan. 2021). https://nvd.nist.gov/vuln/detail/CVE-2018-5390
[8] 2021. Scapy. (Jan. 2021). https://scapy.net/
[9] F. Aarts, J. De Ruiter, and E. Poll. 2013. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 461–468. https://doi.org/10.1109/ICSTW.2013.60
[10] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. 2010. Inference and Abstraction of the Biometric Passport. In *Leveraging Applications of Formal Methods, Verification, and Validation (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Berlin, Heidelberg, 673–686. https://doi.org/10.1007/978-3-642-16558-0-54
[11] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6
[12] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. 2016. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-Box Differential Automata Learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1690?1701. https://doi.org/10.1145/2976749.2978383
[13] Borja Balle and Mehryar Mohri. 2015. Learning Weighted Automata. In *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings (Lecture Notes in Computer Science)*, Andreas Maletti (Ed.), Vol. 9270. Springer, 1–21. https://doi.org/10.1007/978-3-319-23021-4_1
[14] Karthikeyan Bhargavan et al. 2016. Everest: Towards a Verified, Drop-in Replacement of HTTPS. (2016), 11.
[15] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. 2019. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *J. ACM* 66, 1 (2019), 1:1–1:77. https://doi.org/10.1145/3243650
[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[17] Joeri de Ruiter. 2016. A Tale of the OpenSSL State Machine: A Large-Scale Black-Box Analysis. In *Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings (Lecture Notes in Computer Science)*, Billy Bob Brumley and Juha Röning (Eds.), Vol. 10014. 169–184. https://doi.org/10.1007/978-3-319-47560-8_11
[18] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 193–206.
[19] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. [n. d.]. A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer. ([n. d.]), 17.
[20] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. 2020. A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer. *IACR Cryptol. ePrint Arch.* 2020 (2020), 114. https://eprint.iacr.org/2020/114
[21] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of {DTLS} Implementations Using Protocol State Fuzzing. 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean
[22] Paul Fiterău-Broștean, Ramon Janssen, and Frits Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 454–471. https://doi.org/10.1007/978-3-319-41540-6-25
[23] Vidhi Goel, Rui Paulo, and Christoph Paasch. 2020. Testing QUIC with packetdrill. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, Virtual Event USA, 1–7. https://doi.org/10.1145/3405796.3405825
[24] Harry B. Hunt and Daniel J. Rosenkrantz. 1977. On Equivalence and Containment Problems for Formal Languages. *J. ACM* 24, 3 (July 1977), 387–396. https://doi.org/10.1145/322017.322020
[25] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib - A Framework for Active Automata Learning. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 487–495. https://doi.org/10.1007/978-3-319-21690-4_32
[26] William M McKeeman. 1998. Differential Testing for Software. 10, 1 (1998), 8.
[27] 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 (SIGCOMM '19)*. Association for Computing Machinery, Beijing, China, 227–240. https://doi.org/10.1145/3341302.3342087
[28] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. 2018. Observing the Evolution of QUIC Implementations. *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Dec. 2018), 8–14. https://doi.org/10.1145/3284850.3284852 arXiv: 1810.09134.
[29] J. Postel. 1980. DoD standard Transmission Control Protocol. (1980). https://www.rfc-editor.org/info/rfc0761 Number: RFC 761.
[30] 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 (EPIQ'18)*. Association for Computing Machinery, New York, NY, USA, 15?21. https://doi.

org/10.1145/3284850.3284853
[31] Martin Thomson and Jana Iyengar. [n. d.]. QUIC: A UDP-Based Multiplexed and Secure Transport. ([n. d.]). https://tools.ietf.org/html/draft-ietf-quic-transport-29 Library Catalog: tools.ietf.org.
[32] Frits Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95. https://doi.org/10.1145/2967606
[33] Gerco van Heerdt, Clemens Kupke, Jurriaan Rot, and Alexandra Silva. 2020. Learning Weighted Automata over Principal Ideal Domains. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020 (Lecture Notes in Computer Science)*, Jean Goubault-Larrecq and Barbara König (Eds.), Vol. 12077. Springer, 602–621. https://doi.org/10.1007/978-3-030-45231-5_31

*Appendices are supporting material that has not been peer-reviewed.*

# A ARTIFACT APPENDIX

## Abstract

We provide the source code and scripting to run Prognosis, as well as re-run and verify claimed properties and results.

## Scope

The artifact covers the full code and tooling used in both the development and execution of Prognosis, as well as the source code of its targets.

## Contents

The artifact bundles the learner, adapter, synthesizer, and target implementations required to run Prognosis, as well as custom scripting and documentation.

## Hosting

The artifact's code, scripts, and documentation are fully accessible at: https://doi.org/10.5281/zenodo.5040974

## Requirements

A device capable of running Docker and Docker Compose, as well as x86 based images, either natively or through virtualisation.

A minimum of 8 GB of memory, recommended 16 GB for running all experiments.