Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents

Maria Leonor Pacheco*, Max von Hippel†, Ben Weintraub†, Dan Goldwasser*, Cristina Nita-Rotaru†
*Purdue University, West Lafeyette, IN, USA, {pachecog,dgoldwas}@purdue.edu
†Northeastern University, Boston, MA, USA, {vonhippel.m,weintraub.b,c.nitarotaru}@northeastern.edu

***Abstract*—Automated attack discovery techniques, such as attacker synthesis or model-based fuzzing, provide powerful ways to ensure network protocols operate correctly and securely. Such techniques, in general, require a formal representation of the protocol, often in the form of a finite state machine (FSM). Unfortunately, many protocols are only described in English prose, and implementing even a simple network protocol as an FSM is time-consuming and prone to subtle logical errors. Automatically extracting protocol FSMs from documentation can significantly contribute to increased use of these techniques and result in more robust and secure protocol implementations.**

In this work we focus on attacker synthesis as a representative technique for protocol security, and on RFCs as a representative format for protocol prose description. Unlike other works that rely on rule-based approaches or use off-the-shelf NLP tools directly, we suggest a data-driven approach for extracting FSMs from RFC documents. Specifically, we use a hybrid approach consisting of three key steps: (1) large-scale word-representation learning for technical language, (2) focused zero-shot learning for mapping protocol text to a protocol-independent information language, and (3) rule-based mapping from protocol-independent information to a specific protocol FSM. We show the generalizability of our FSM extraction by using the RFCs for six different protocols: BGPv4, DCCP, LTP, PPTP, SCTP and TCP. We demonstrate how automated extraction of an FSM from an RFC can be applied to the synthesis of attacks, with TCP and DCCP as case-studies. Our approach shows that it is possible to automate attacker synthesis against protocols by using textual specifications such as RFCs.

I. Introduction

Automated attack discovery techniques, such as attacker synthesis or model-based fuzzing, provide powerful ways to ensure network protocols operate correctly and securely. Such techniques, in general, require a formal representation of the protocol, often in the form of a finite state machine (FSM). Unfortunately, many protocols are only described in English prose, and implementing even a simple network protocol as an FSM is time-consuming and prone to subtle logical errors. Automated attack discovery techniques are therefore infrequently employed in the real world because of the significant effort required to implement a protocol FSM. Automatically extracting protocol FSMs from documentation can significantly contribute to increased use of these techniques and result in more robust and secure protocol implementations.

We observe that for network protocols there is an untapped *resource of information* available in the form of RFCs. With the recent interest in using data to automatically solve problems in several fields, we ask the question: "Can we leverage formal prose descriptions of protocols to improve protocol security?"

Given the inherent ambiguity of natural language text, extracting protocol information is not a straightforward task. The writers of protocol specifications often rely on human readers' understanding of context and intent, making it difficult to specify a set of rules to extract information. This is by no means unique to the computer networks domain, and as a result, the natural language community shifted its focus over the last decade to statistical methods that can help deal with such ambiguity. At the same time, one can not just apply "off-the-shelf" implementations of NLP tools combined in an ad-hoc way, as training such tools on poorly selected datasets will result in reduced performance and cause the resulting applications to be brittle.

Unlike other software, network protocols follow a specific pattern: they are described by messages and FSMs, they must meet temporal safety and liveness properties, and they follow a structured language. Thus, NLP tools trained on such aspects of a protocol are likely to generalize on protocols with a similar structure. Unlike other NLP tasks where high precision is needed, protocol validation is more robust to noisy NLP results because the ultimate result comes from protocol execution.

NLP techniques have been applied selectively in related problems. WHYPER [1] and DASE [2] apply NLP techniques to identify sentences that describe the need for a given permission in a mobile application description and extract command-line input constraints from manual pages, respectively. The work in [3] used documentation and source code to create an ontology allowing the cross-linking of software artifacts represented in code and natural language on a semantic level.

Several other works looked at inferring protocol specification – based on network traces [4], [5], [6], [7], [8], using program analysis [9], [10], [11], [12], or through model checking [13], [14]. Comparetti *et al.* [4] infer protocol state machines from observed network traces by clustering messages based on the similarity of message contents and their reaction to the execution. Caballero *et al.* [6] extracts the protocol message format, given a trace of protocol messages. Cho *et al.* [7] extracts the protocol state machines from network traces with the help of a set of end-user provided abstraction functions to generate an abstract alphabet out of trace messages. The approach relies intensively on human expert input.

Our contribution. In this work we focus on attacker synthesis as a representative technique for protocol security, and on RFCs as a representative format for protocol description. Our goal is to close the automation gap between automated protocol specification and security validation by extracting protocol FSMs from the corresponding RFCs. Unlike other works that rely on rule-based approaches or use off-the-shelf NLP tools directly, we suggest a *data-driven approach* for extracting information from RFC documents. Off-the-shelf NLP tools are typically trained over news documents, and when applied to technical documents that include many out-of-vocabulary words (i.e. technical terms), their performance degrades substantially. Rule-based systems, on the other hand, are developed to support information extraction based on the specific format of the textual input. Unfortunately, different RFC documents define variables, constraints, and temporal behaviors totally differently. Moreover, RFCs follow no common document structure. Machine-learning systems can deal with these challenges, however training such systems from scratch requires significant human effort annotating data with the relevant labels, which could be different for different protocols. We confront these challenges with a hybrid approach consisting of three key steps: (1) large-scale word-representation learning for technical language, (2) focused zero-shot learning for mapping protocol text to a protocol-independent information language, and (3) rule-based mapping from protocol-independent information to a specific protocol FSM.

While RFCs are the main form for textual specification for protocols, they do not necessarily contain the complete specification, referred to as canonical FSM. There does not exist a one-to-one mapping between the textual specification and a canonical formal specification of the state machine, as canonical FSMs are created based not only on information contained in RFCs, but on input from experts with domain knowledge. This is a limitation for any statistical NLP approach. In this paper, we propose an alternative intermediary representation (i.e. a grammar) that can be used to recover partial state machines.

Our approach exploits the large number of technical documents found in online technical forums to train a deep learning model, capturing the properties of and interactions between technical terms. This process *does not require direct annotation*, and does not add to the human effort involved in building the model. Our zero-shot information extraction approach builds on that representation. Since each protocol consists of its own set of predicates and variables, we suggest a zero-shot approach in which we separate between protocols observed during training and testing. The model learns to identify and connect concepts relevant for the training protocols and at test time it is evaluated on extracting a set of symbols which were not observed at training. The output of that step creates an *intermediate representation* of conditions, operations and transitions, extracted from protocol text. The final step transpiles the intermediate representation into an FSM written in PROMELA code [15]. We make the following contributions:

- We propose an embedding that allows us to learn network technical terms without the need to annotate data. To learn this embedding, we collected a set of 8,858 *unlabeled* RFCs from `ietf.org` and `rfc-editor.org` covering aspects of computer networking, including protocols, procedures, programs, concepts, meeting notes and opinions. These documents contain a total of 475M words.
- We suggest and evaluate an NLP framework for the task of recovering FSMs from the RFCs, designed to adapt to previously unobserved protocols. We show the generalizability of our FSM extraction by using the RFCs for six different protocols: BGPv4, DCCP, LTP, PPTP, SCTP, and TCP. As part of the NLP framework we propose a general-purpose abstraction for annotating the segments of text in RFC specification documents that describe the FSM for each of six network protocols. For example, of the 20 transitions in the TCP FSM, our NLP pipeline can extract 17, either correctly or partially so.
- We demonstrate how automated extraction of an FSM from an RFC can be applied to the synthesis of attacks, with TCP and DCCP as case-studies. We find that even when the extracted FSM has errors, we can generate attacks that are confirmed on a canonical hand-written model of the same protocol. However, the quality of the extracted FSM impacts the accuracy of the attack synthesis. For example, in the case of TCP, we can find attacks against only one property using our NLP pipeline, as opposed to against all four when using the canonical FSM.

The code is available at https://github.com/RFCNLP.

The rest of the paper is organized as follows. We discuss attack synthesis and NLP techniques in Section II. Our grammar is described in Section III, technical language embedding in Section IV, parsing in Section V, and FSM extraction in Section VI. We present the TCP and DCCP attack synthesis case studies in Section VII. We evaluate NLP components, FSM extraction, and automated attack synthesis in Section VIII. We present related work in Section IX. We discuss limitations and improvements in Section X.

II. MOTIVATION AND OUR APPROACH

In this section, we summarize the motivation of our work, the main challenges related to the extraction of FSMs from specification documents, and the way our approach is designed to circumvent these challenges.

A. Need for Automated FSM Extraction

Automated attack discovery methods typically model the system under attack abstractly — either implicitly, e.g. with a statistical representation, or explicitly, e.g. with a finite state machine [16], [17], [18]. An FSM represents a program as a graph, where the nodes are program *states* and the edges are *transitions* (i.e. changes in state). Recent work in the theory of security use FSMs to define what it means to attack [19], [18]. Conversely, various attack discovery methods leverage FSMs to compute attacks [16], [20], [21], [22], [18].

Current attack finding [16], [20], [22] and attack synthesis techniques [18] rely on a manually defined FSM specified by an expert. Anecdotally, there are reports where such FSMs

were derived from code directly because specifications lacked such a description [22].

B. Challenges in FSM Extraction from RFCs

One common way of specifying protocols is with RFCs. While RFCs provide some structure that can be exploited for automated information extraction, it is not a straightforward task. An RFC describes in natural language, which is inherently ambiguous, the protocol's variables, states, and conditions for state transitions. Even for humans, creating a formal model from the text requires considerable domain expertise. From an NLP perspective, this is a specialized information extraction task, called *semantic parsing* [23], mapping the protocol text into structured information: the FSM. The mapping consists of multiple inter-dependent predictions, each extracting individual elements from the document, which together should capture the conditions and transitions of the FSM. Unlike traditional semantic parsing domains that operate over short texts, such as mapping a request in natural language to a command for a personal assistant (e.g., *"set timer to 30 minutes"*), extracting an FSM requires processing multiple interconnected sentences to capture the transitions from just a single state.

Recently, very promising results were obtained by NLP researchers using deep learning methods for information extraction and semantic parsing tasks [24], [25], [26], [27]. However, most of the recent successes in these areas depend on large amounts of annotated data. When dealing with technical domains, high-quality annotated resources are scarce, and the fact that a deep understanding of the protocols is needed to annotate these documents makes generating enough data to support machine learning systems a costly and difficult process.

Furthermore, dealing with specialized domains also reduces the amount of non-labeled data which can potentially be used. Unlike traditional NLP domains, such as newswire text, in which a vast amount of data is available for training NLP models, when learning to extract FSMs from network protocol RFCs, we are limited by the number of existing protocols. Given the data scarcity problem, it is difficult to build an NLP model that will reliably generalize to new protocols that were not observed during the training phase, as there is no common document structure to RFCs and the different functionality described by the protocols results in a different set of symbols and behaviors used by each protocol. In the next section we describe our approach for dealing with these challenges.

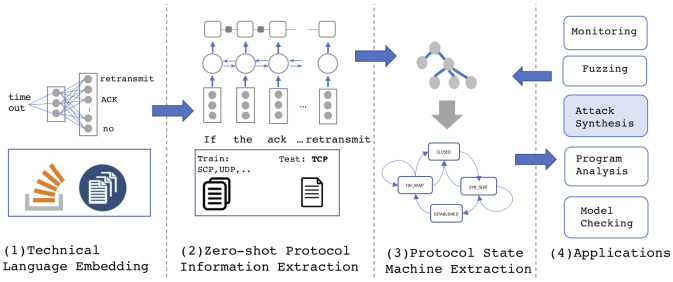


Fig. 1: Overview of our approach.

C. Our Approach

As we discussed in the previous section, the lack of training resources prevents us from taking an end-to-end learning approach, in which a complex neural model is trained to predict the complete FSM directly from the text. Instead, we break the process into several parts, allowing us to exploit several different forms of supervision and human expertise. Figure 1 describes our approach consisting of the following steps:

Technical Language Embedding. While there are only a few protocol RFCs, there are large amounts of technical documents discussing them and other related networking concepts that provide the background for understanding the RFC. These documents include technical forums, blogs, research papers, and specification documents. We exploit these documents to learn a distributed word representation, also known as an *embedding* model, for technical language. The main advantage of this step is that it is an unsupervised process, and we do not require any annotations. Learning these representations will allow us to carry over information from the networking domain to our next step. Section IV describes this step in detail.

Zero-Shot Protocol Extraction. Once we have this representation, we turn our attention to learning a model to extract information regarding the FSM from the RFCs. To do this, we define a grammar that describes a higher-level abstraction of the structure of a general FSM for network protocols. While general, this abstraction will allow us to leverage different protocols to learn to extract this information, even when the underlying structure of their documents, the way the FSM is described and the specific names of variables, events and states vary between protocols. We explain this grammar in Section III. We annotate a set of six protocols, and use a zero-shot learning approach, in which the document for the predicted protocol is *completely unobserved during training*. The output of this step is a generic representation referred to as the *intermediate representation.* Section V describes this step in detail.

Protocol State Machine Extraction. The extracted information structured according to our general protocol grammar must be converted into an actual FSM implementing the described protocol. We use a set of heuristics to extract an FSM from the intermediate representation, as detailed in Section VI.

III. Finite State Machine Grammar

We define a general grammar to represent the state machine for the pertinent network protocols in their corresponding RFC specification documents. We use this grammar to annotate the segment of texts that describe the states, variables, and events that are relevant to the state machine, as well as the actions and the logical flow describing their behavior. Annotations are done using XML. We consider four types of annotation tags: *definition tags*, *reference tags*, *state machine tags*, and *control flow tags*, which are formalized below. Finally, we formulate the grammar in Backus-Naur Form in Figure 2.

A. Definition Tags

Definition tags are used to annotate the names of states, events, and variables that are relevant to each protocol. These

are text segments that are referenced throughout the document, and stake a role in defining the state machine. For example, a message may be tagged as an event if the receipt of such a message leads to a state transition.

State definition. When the name of a state is introduced in the text, it is annotated as such. Specifically, the `<def_state>` surrounds the first usage of the state name that is part of some discernible pattern. Often this pattern is in the form of a newline-delimited list or bullet points, but can also appear as a comma-delimited list. The tag also includes an identifier that is unique among the states. For example `<def_state id="##">IDLE</def_state>`, where ## is replaced by the identifier. We assign state identifiers (SID) as monotonically increasing integers in order of first appearance. Punctuation trailing the state name is not included in the tag. These tags and SIDs will be referenced by *reference tags*.

Event definition. Events are also annotated to be referenced throughout the text. Events follow the same annotation conventions as states and use the annotation form: `<def_event id="###">`. We will refer to unique event identifiers as EIDs.

Variable definition. Variables are defined in a similar way to events and states, however they do not include an analog to SIDs or EIDs, because they are not explicitly referenced by annotation in the rest of the text.

B. Reference Tags

When an event or state occurs in the text, it must be linked to an event or state which was tagged. They need to be explicitly defined because sometimes the proper name of a state or event will not be used. For example, an RFC may formally refer to one event as "ACK", but throughout the text these ACKs may also be referred to as "acknowledgments". These are really the same event, and the reference tags are used to clarify that.

State reference. States are referenced by surrounding the state's name throughout the text with the `<ref_state id="##">` tag, where ## corresponds to the appropriate SID that was included with the state's `<def_state>` tag. Punctuation trailing the state name is not included in the tag. An example might look like the following: `enter <ref_state id="2">SYN-SENT</ref_state> state`.

Event reference. Events follow the same convention as state references. The event reference must also include the *type* of event, where the three possible types are: send, receive, and compute. Type tags are included as XML attributes, and will appear as in the following example: `a <ref_event type="send" id="10">SYN</ref_event> segment`.

C. State Machine Tags

We define a set of five tags to represent the state machine logic. These are the crux of the annotation.

Transition. Denotes a state change that happens in the given context. We use argument tags `<arg_source>`, `<arg_target>` and `<arg_intermediate>` to specify the segment in the text playing that role. For example, `<transition>The server moves from the <arg_source>OPEN state</arg_source>, possibly through the <arg_inter>CLOSEREQ state</arg_inter>, to <arg_target>CLOSED</arg_target></transition>`. Note that in this case, the mentions to "OPEN", "CLOSEREQ" and "CLOSED" would also be enclosed in a `<ref_state>` tag. In cases where the text is not explicitly annotated with argument tags, the states mentioned are assumed to be the ending states of the transition.

Variable. Certain variables may be tracked as part of the state machine. This tag should be used to surround any logic that indicates that any of these variables are altered or set to a new value. For example, `<variable>SND.UP <- SND.NXT-1</variable>`

Timer. This tag is used if a timer value is changed or set. For example, `<timer>start the time-wait timer</timer>`.

Error. If a context results in an error or warning being thrown, the error message is then surrounded by this tag. For example, `<error>signal the user error: connection aborted due to user timeout</error>`.

Action. If a given context demands that some action be taken, we use this tag. We specifically mark three types of actions: *send*, *receive* and *issue*. Type tags are included as XML attributes. We use an argument tag `<arg>` to specify the argument in the text being sent, received or computed. For example: `<action type="send">Send <arg>a SYN segment</arg></action>`. Note that in this case, the mention to "SYN" would also be enclosed in a `<ref_event>` tag: `<ref_event id="10">SYN</ref_event>`. Additionally, there are certain events that are ambiguous in terms of how they relate to the state machine, in those cases, this tag can be used without further specifications.

D. Flow Control Tags

A `<control>` tag is introduced to indicate that some flow control or conditional logic is about to follow. The flow control logic should contain a `<trigger>` tag, which captures the event that triggers some action in the state machine, followed by one or more of the state machine tags. A single block of control tags may contain multiple state machine tags. These state machine tags should be in the form of a list. In this case, the implication is that the state machine tags should all be executed if the initial trigger condition is true. Figure 7 in Appendix A shows an example of a list of events within one control block from the TCP RFC (a.k.a. RFC 793) [28].

E. Grammar

Let `engl` denote any valid string in the English language. Then, the grammar for the state machine annotation can be described in Backus-Naur Form as observed in Figure 2. Here, `relevant=true` indicates that the corresponding annotation is relevant to the protocol state-machine.

IV. Technical Language Embedding

In this section we describe our approach to learn distributed word representations for technical language. We start by providing some background about the techniques used to learn these representations, then we describe our embedding in detail.

```
bool      ::= true | false
type      ::= send | receive | issue
def-tag   ::= def_state | def_var | def_event
ref-state ::= ref_state id="##"
ref-event ::= ref_event id="##" type="type"
ref-tag   ::= ref-event | ref-state
def-atom  ::= <def-tag>engl</def-tag>
sm-atom   ::= <ref-tag>engl</ref-tag> | engl
sm-tag    ::= trigger | variable | error | timer
act-atom  ::= <arg>sm-atom</arg> | sm-atom
act-struct::= act-struct | act-struct act-atom
trn-arg   ::= arg_source | arg_target | arg_inter
trn-atom  ::= <trn-arg>sm-atom<trn-arg> | sm-atom
trn-struct::= trn-struct | trn-struct trn-atom
ctl-atom  ::= <sm-tag>sm-atom</sm-tag>
            | <action type="type">act-struct</action>
            | <transition>trn-struct</transition>
            | sm-atom
ctl-struct::= ctl-atom | ctl-struct ctl-atom
ctl-rel   ::= relevant=bool
control   ::= <control ctl-rel>ctl-struct</control>
e         ::= control | ctl-atom | def-atom
            |  e_0 e_1
```

Fig. 2: BNF grammar for RFC annotation.

A. Background

Distributed representations of words aim to capture meaning in a numerical vector. Unlike symbolic representations of words, that use binary values to signal if the words are present or not, word embeddings have the ability to generalize by pushing semantically similar words closer to each other in the embedding space. When using binary representations of words, we can only consider features that we have seen during training. Consider a scenario in which during training, we only have access to DCCP. If we were to test our learned model on TCP, we could not represent words that were not observed during training.

Several models have been suggested to learn distributed word representations. Some approaches rely on matrix factorization of a general word co-occurrence matrix [29], while other approaches use neural networks trained to predict the context surrounding a word, and in the process, learn efficient word embedding representations in their inner layers [30], [31]. In this paper we focus on contextualized word representations. Unlike static word representations that learn a single vector for each word form, contextualized representations allow the same word form to take different meanings in different contexts. For example, in the sentence "The connection is in error and should be reset with Reset Code 5", the word "reset" has two different meanings. Contextualized representations compute different vectors for each mention.

State-of-the-art pre-trained language models provide a way to derive contextualized representations of text, while allowing practitioners to fine-tune these representations for any given classification task. One example of such models is BERT (Bidirectional Encoder Representations from Transformers) [32]. BERT is built using a Transformer, a neural architecture that learns contextual relations between words in a word sequence. A Transformer network includes two mechanisms, an encoder that reads the input sequence, and a decoder that predicts an output sequence. Unlike directional models that read the input sequentially, Transformer encoders read the whole sequence at once, and allow the representation of a given word to be informed by *all* of its surroundings, left and right. Details regarding the Transformer architecture can be found in the original paper [33].

To learn representations, BERT uses two learning strategies, masked language modeling and next sentence prediction. The first strategy masks 15% of the words in each sentence, and attempts to predict them. The second strategy uses pairs of sentences as input, and learns to predict whether the second sentence is the subsequent sentence in the original document. Figure 3 illustrates this process. BERT models were pre-trained on the BooksCorpus (800M words) and English Wikipedia (2,500M words) and are publicly available[1].

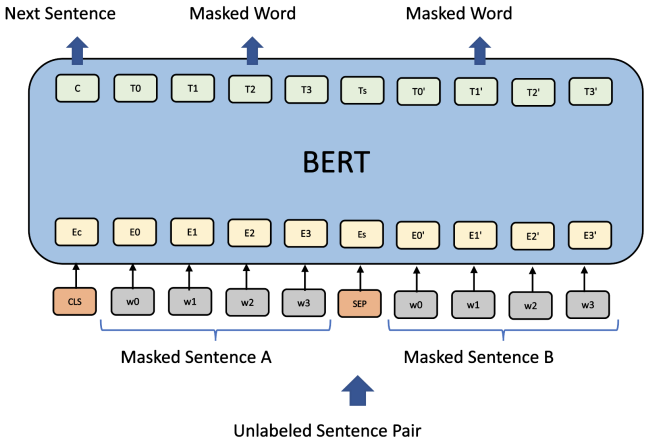


Fig. 3: BERT pre-training.

B. Our Embedding

While we could use pre-trained language models directly for predicting FSM tags, we note that these models were trained on general document repositories. To obtain a model that better represents the domain vocabulary, we further pre-train BERT using the masked language model and the next sentence prediction objective using networking data. We collected the full set of RFC documents publicly available in `ietf.org` and `rfc-editor.org`. These documents cover different aspects of computer networking, including protocols, procedures, programs, concepts, meeting notes and opinions. The resulting dataset consists of 8,858 documents and approximately 475M words. Note that we do not need any supervision for this step.

Previous findings suggest that further pre-training large language models on the domain of the target task consistently improves performance [34]. Our experiments in Section VIII support this hypothesis.

[1]https://github.com/google-research/bert

V. Zero-shot Protocol Information Extraction

In this section we describe our design for a protocol information extraction system. Our main goal is to have a system that can adapt to new, unobserved protocols without re-training the system. To support this, we build on the general grammar introduced in Section III that focuses on concepts relevant to a wider set of protocols and takes advantage of the technical language embedding described in Section IV.

A. Sequence-to-Sequence Model

To parse specification documents, we designed a sequence-to-sequence model that receives text blocks as input, and outputs a sequence of tags corresponding to the grammar described in Section III. To tag the text, we use BIO (Beginning, Inside, Outside) tag labels. Text blocks correspond to paragraphs in the RFC document. Initially, we segment paragraphs into smaller units (e.g. individual words, chunks or phrases). Then, we map each unit to a particular tag. To illustrate this process, consider the parsed statement in Figure 4, mapping chunks to BIO-tags.

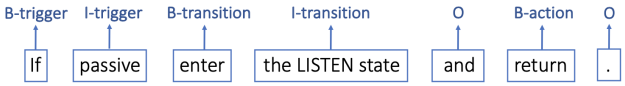


Fig. 4: BIO example.

We consider two models to learn the sequence to sequence mapping: a linear model we refer to as LinearCRF, and a neural model based on the BERT embedding, which we refer to as NeuralCRF.

Linear-Chain Conditional Random Fields (LinearCRF) works on a set of extracted features over each chunk. Conditional Random Fields model the prediction as a probabilistic graphical model; Chain Conditional Random Fields specifically consider sequential dependencies in the predictions [35].

Let $\boldsymbol{y}$ be a tag sequence and $\boldsymbol{x}$ an input sequence of textual units. We want to maximize the conditional probability:

$$
\begin{aligned}
p(\boldsymbol{y}|\boldsymbol{x}) &= \frac{p(\boldsymbol{y}, \boldsymbol{x})}{\sum_{\boldsymbol{y'}} p(\boldsymbol{y'}, \boldsymbol{x})} \\
p(\boldsymbol{x}, \boldsymbol{y}) &= \prod_{t=1}^{T} \exp(f(y_t, y_{t-1}, \boldsymbol{x_t}; \boldsymbol{\theta}))
\end{aligned}
\tag{1}
$$

Where f is a linear scoring function learned with parameter vector $\boldsymbol{\theta}$ over a feature vector $\boldsymbol{x_t}$. To learn $\boldsymbol{\theta}$, we minimize the negative log-likelihood $-\log p(\boldsymbol{y}, \boldsymbol{x})$. Learning is made tractable by using the forward-backward algorithm to calculate the partition function $Z(\boldsymbol{x}) = \sum_{\boldsymbol{y'}} p(\boldsymbol{y'}, \boldsymbol{x})$.

The second model considered is a BERT encoder enhanced with a Bidirectional LSTM CRF layer (NeuralCRF). LSTMs are recurrent neural networks, a class of neural network where connections between nodes form a directed graph along a sequence [36]. We outline this model in Figure 5. The BERT encoder is used to create chunk-level representations from word sequences. The resulting sequence of chunks is then processed

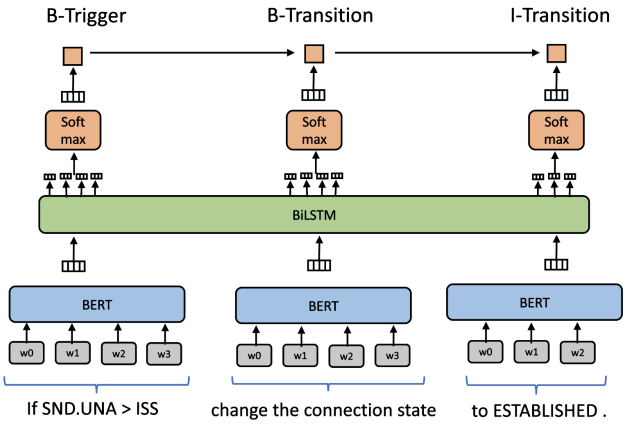


Fig. 5: NeuralCRF.

using a BiLSTM. A softmax activation is used to obtain scores for the labels. Finally, we add a CRF layer on top. This way, we are able to leverage the sequential dependencies both in the representation and in the output space [37], [38]. Note that BERT enforces a limit of 512 tokens per sequence, which is not enough to represent some of our control sequences. For this reason, we leverage a BiLSTM instead of using the CRF layer directly over BERT.

To formalize the NeuralCRF model, we first consider a textual unit containing n words $(\boldsymbol{w_0}, \boldsymbol{w_1}, ..., \boldsymbol{w_{n-1}})$. A BERT encoder is used to obtain a single representation $\boldsymbol{u}$ for the full textual unit, resulting in a d-dimensional vector.

Then, a BiLSTM computes a representation over the sequence of embedded textual units $(\boldsymbol{u}_0, \boldsymbol{u}_1, ..., \boldsymbol{u}_{m-1})$ to obtain representations $\boldsymbol{h_t} = [\overrightarrow{\boldsymbol{h}_t}; \overleftarrow{\boldsymbol{h}_t}]$ for every textual unit t. Here, $\overrightarrow{\boldsymbol{h}_t}$ represents the left context of the sequence, and $\overleftarrow{\boldsymbol{h}_t}$ represents the right context, at every unit t.

Finally, we add a CRF layer over these representations by replacing the function f in Eq. 1 with:

$$
f(y_t, y_{t-1}, \boldsymbol{x}_t) = \boldsymbol{h_t} + \boldsymbol{P}_{y_t, y_{t-1}} \tag{2}
$$

Where $\boldsymbol{x}_t$ represents the input for that textual unit, $\boldsymbol{h_t}$ is the representation of the textual unit computed with our model and $\boldsymbol{P}$ is a learned parameter matrix representing the transitions between labels. Like in the linear CRF case, we minimize the negative log likelihood, $-\log p(\boldsymbol{y}, \boldsymbol{x})$, to jointly learn the parameters of the BERT encoder, the BiLSTM layer, and the transition matrix $\boldsymbol{P}$.

Predictions for both models are done using the Viterbi algorithm. Viterbi is a dynamic programming algorithm for finding the most likely sequence of states. Viterbi takes into account both emission ($\boldsymbol{h_t^2}$), and transition ($\boldsymbol{P}_{y_t, y_{t-1}}$) scores at each unit t in the sequence.

B. Features

For each textual unit in the input, we extract a set of features to capture properties about the input and help us make a correct

classification.

Vocabulary. We extract bag-of-word features for all stemmed forms of the words in the training data. Stemming is the process of producing morphological variants of a root word. This way we can reduce redundancy, as word stems and their inflected or derived words usually have the same meaning.

Capitalization Patterns. We use features to indicate the different capitalization patterns of the original words (before stemming). We consider a feature for each of the following patterns: all letters are in lower case, all letters are capitalized, the first letter is capitalized, the word is in camel case, the word has only symbols, the word has only numeric characters, or the word has any other form of alpha-numeric capitalization.

Logical and Mathematical Expression Patterns. We identify different patterns corresponding to logical and mathematical expressions. These include assignments ($x := a$, x $x \leftarrow a$, $x = a$), comparisons ($a < b$, $a > b$, $a \leq b$, $a \geq b$, $a == b$), and arithmetic and algebraic expressions.

Dictionary Features. We include indicator features for a held-out dictionary of reserved state and variable names.

Part-of-Speech Tags. We include part-of-speech (POS) tags for all observed words (e.g. noun, verb, adjective). For extracting POS tags, we use an off-the-shelf tagger.

Position Features. We use position and relative position indicators for each word in a chunk.

All of the features used are standard in general NLP pipelines. For the LINEARCRF, this collection of features represents the input $\boldsymbol{x}_t$ for each textual unit t. For the NEURALCRF, we concatenate the feature vector to the resulting vector $\boldsymbol{u}_t$ from the BERT encoder, before being inputted to the BiLSTM layer.

C. Post-Processing

We experiment with a set of *rules* to correct some easy cases that the prediction models fail to identify. The rules are applied on top of the classification output, by flipping labels in the relevant cases. First, we look for textual units with mentions to states. If the unit mentions a state, and there is a transition verb (e.g. move, enter) or a directional preposition (e.g. to, from), we label the unit as a *transition* span. Then, we look for textual units with mentions to events. If the unit mentions an event, and there is an action verb (e.g. send, receive), we label the unit as an *action* span. Then, we label any remaining unlabeled span with mentions to states or events as a *trigger*. Finally, we label any remaining unlabeled spans with mentions to variable names, "error" or "timer" as *variable*, *error* and *timer*, respectively. We refer to the models that use these rules as LINEARCRF+R and NEURALCRF+R.

Once the triggers, transitions, actions, variables, and errors are identified, we use an off-the-shelf Semantic Role Labeler (SRL) [39] to identify predicted actions as either send, receive, or issue, depending on the verb used, as well as to identify the segment in the text being sent, received, or issued. Semantic Role Labeling consists of detecting semantic arguments associated with the predicate or verb in a sentence, and their classification into their specific roles. For example, given a sentence like "Send a SYN segment", an SRL model would identify the verb "to send" as the predicate, and "SYN segment" as the argument. Identified arguments are then tagged using the `<arg>` tag introduced in Section III. We also use the SRL output to identify transitions verbs such as enter and leave, and identify the segment in the text being explicitly mentioned as the source or target for this transition. For example, in the sentence "client and server sockets enter this state from PARTOPEN", the SRL model identifies the verb "to enter" as the predicate, the segment "this state" as the argument and "from PARTOPEN" as the directional modifier. Arguments and directional modifiers are then tagged as as `<arg_source>`, `<arg_target>` or `<arg_intermediate>`, depending on the prepositions used.

In addition, we use exact lexical matching to identify explicit mentions to states and events in the predicted sequences. We keep track of the indentation in the original documents to infer the scope of `<control>` statements. The resulting tagged text constitutes the intermediary representation that will be used for extracting the FSM.

VI. FSM EXTRACTION

The intermediary representation obtained using our LINEARCRF or NEURALCRF model is not an FSM, thus we need a procedure to extract an FSM from the intermediary representation. The FSM is expressed as $P = \langle S, I, O, s_0, T \rangle$ with finite *states* S, finite *inputs* I, finite *outputs* O disjoint from I, *initial state* $s_0 \in S$, and finite *transitions* $T \subseteq S \times (\{\epsilon, \textit{timeout}\} \cup (I \cup O)^*) \times S$.

We extract the states S by scanning the intermediary representation for `def-states`. If one of the `def-states` contains "initial" or "begin" in its body, we set the corresponding state as the initial state s_0; otherwise we just choose whichever is the first `def-state` in the document. We extract the inputs I and outputs O by scanning for `def-events` where the `type` is `receive` or `send`, respectively.

Although the intermediary representation contains `transition` blocks, these blocks do not exactly contain actual FSM transitions. Rather, they contain pointers for *where to look* in the intermediary representation in order to guess the source and target states, and labels, for the FSM transitions. A `transition` block might describe no transitions at all, or multiple transitions at once. It might describe only part of a transition, for example the label and the target state, while the rest is described somewhere else in its context. Such cases can occur even with a perfect intermediary representation, because of complex syntax and formatting used in the RFC text. To obtain the transition set T we proceed in two steps: first we extract potential transitions from the `transition` blocks; then we heuristically prune transitions that look like noise.

Potential Transition Extraction. We define an initially empty set of *possible transitions* T_{pos}. For each `transition` block T in the intermediary representation xml, we compute potential transitions described in T using the Algorithm EXTRACTTRAN. Briefly, EXTRACTTRAN searches *lower* in the intermediary representation to find target states, and *higher* to find source states. It handles sentences like "starting at any state other

than CLOSED" using the set complement. It also handles explicitly labeled intermediate states, so that sentences like "the machine goes to CLOSED, then REQUEST, then PARTOPEN" are interpreted as CLOSED → REQUEST → PARTOPEN rather than CLOSED → PARTOPEN, REQUEST → PARTOPEN. It uses the helper function EXTRACTTRANLBL to guess the transition label ℓ, recursing upward in the ancestry of T at most six times until the result is well-formed. Pseudocode for EXTRACTTRAN is given in the Appendix.

Heuristic Transition Pruning. After adding the potential transitions extracted from each `transition` block T to a set T_{pos}, we filter T_{pos} using three heuristics. First, we remove any possible transition $t \in T_{pos}$ that does not type-check, that is, for which $t \notin S \times (\{\epsilon, \mathit{timeout}\} \cup (I \cup O)^*) \times S$. Second, we apply a "call and response" heuristic, where if T_{pos} contains some transitions $s \xrightarrow{x?y!} s'$, $s \xrightarrow{x?} s'$, and $s \xrightarrow{y!} s'$, then the latter two are discarded because they are likely noise generated by the first one. Third, we apply a "redundant epsilons" heuristic, where if T_{pos} contains some transitions $s \xrightarrow{\epsilon} s'$ and $s \xrightarrow{\ell} s'$, where $\ell \neq \epsilon$, then the ϵ-transition is discarded because it is likely noise generated by the ℓ-transition. The transitions T is the remaining filtered set T_{pos}.

VII. Task: Attacker Synthesis

In this section we use attacker synthesis as an exemplifying application for FSM extraction.

A. Attacker Synthesis

LTL program synthesis, also known as the *LTL implementability problem*, is to deduce for an LTL property ϕ if there exists some program P that makes ϕ true. For example, ϕ could be the homework assignment to implement multi-PAXOS, and the program synthesis problem would be to automatically compute a satisfying code submission. The problem is known to be doubly exponential in the size of the property [40].

LTL attacker synthesis is slightly different. In this work we consider a centralized attacker synthesis problem for protocols, where the attacker has just one component. Other variations on the problem are formulated in [18]. Suppose $P \parallel Q$ is a system consisting of some programs P and Q, and ϕ is an LTL correctness property which is made true by the system; that is $P \parallel Q \models \phi$. Consider the threat model where Q is the vulnerable part of the system. The attacker synthesis problem is to replace Q with some new *attacker* A having the same inputs and outputs as Q, such that the augmented system behaves incorrectly, that is, $P \parallel A$ violates ϕ. We only consider attackers which succeed under the assumption that (a) the attack eventually terminates, and (b) when the attack terminates, the vulnerable program Q is run. The *program synthesizer* must compute a program that satisfies ϕ in all of its (non-empty set of) executions, but the *attacker synthesizer* only needs to compute a program that violates ϕ in one execution.

B. Attacker Synthesis with KORG

KORG is an open-source attacker synthesis tool for protocols. It requires three inputs: (1) a PROMELA program P representing the invulnerable part of the system; (2) a PROMELA program Q representing the vulnerable part of the system, as well as its interface (inputs and outputs) in YAML format; and (3) a PROMELA LTL property ϕ representing what it means for the system to behave correctly. KORG computes $\exists$-attackers (attackers for which there exists a winning execution) by reducing the attacker synthesis problem to a model-checking problem over the system $P \parallel \textsc{Daisy}(Q)$, where the vulnerable program Q is replaced with a nondeterministic search automaton (called a Daisy Gadget) having the same interface as Q. The model-checker then computes an execution that violates the correctness property, and KORG projects the component of the execution representing the gadget's actions into a new PROMELA program, which is the synthesized attacker [18].

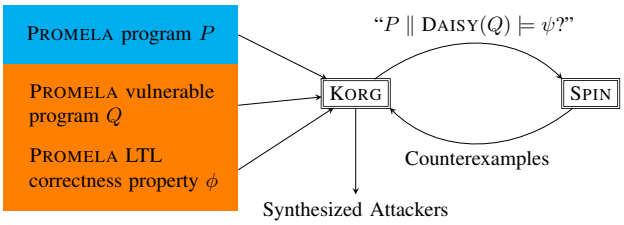


Fig. 6: KORG work-flow. With our NLP pipeline, the user need only supply the orange inputs and the system RFC. The property ψ is automatically computed from ϕ to ensure the attacker eventually terminates, at which point the original code Q is run. The DAISY gadget is defined in [18].

C. TCP and DCCP Attacker Synthesis with KORG

We focus on the TCP and DCCP connection establishment and tear-down routines as representative protocols for attacker synthesis. The TCP connection routine was previously studied using the attacker synthesis tool KORG (Fig. 6); now we conduct a similar analysis for both TCP and DCCP using the same tool, but we automatically extract FSMs using NLP. We want to show that the FSMs extracted from our NLP pipeline can be used directly for attacker synthesis, alleviating the considerable engineering effort required to hand-model the system under attack. We show the effectiveness of the FSM extraction on this task in Section VIII-B.

Our NLP pipeline and FSM extraction produce an FSM. In order to use the extracted FSM for attacker synthesis, we transpile it to PROMELA. For example, if we begin with the TCP RFC, then the result will be a PROMELA program describing the TCP connection routine.

For each of TCP and DCCP, we hand-write four LTL properties in PROMELA based on a close reading of the corresponding RFC. Our TCP properties are given in Equation 3, and our DCCP properties are given in Equation 4. We define the vulnerable PROMELA program Q to be a generic message channel between peers. For each of the four ϕ_i, we feed the inputs P, Q, and ϕ_i to KORG and generate attackers. But how do we know if these attackers are legitimate, since they were generated with a potentially incorrect program P? We solve this by testing the attackers against a Canonical PROMELA

program. For TCP we adapt the Canonical program from [18]. For DCCP, no such program was available and we created our own hand-written Canonical PROMELA program.

$$\begin{aligned}
\phi_1 &= \text{“No half-open connections.”} \\
\phi_2 &= \text{“Passive/active establishment eventually succeeds.”} \\
\phi_3 &= \text{“Peers don't get stuck.”} \\
\phi_4 &= \text{“}\texttt{SYN_RECEIVED}\text{ is eventually followed by }\texttt{ESTABLISHED}\text{, }\texttt{FIN_WAIT_1}\text{, or }\texttt{CLOSED}\text{.”}
\end{aligned} \tag{3}$$

$$\begin{aligned}
\theta_1 &= \text{“The peers don't both loop into being stuck or infinitely looping.”} \\
\theta_2 &= \text{“The peers are never both in }\texttt{TIME_WAIT}\text{.”} \\
\theta_3 &= \text{“The first peer doesn't loop into being stuck or infinitely looping.”} \\
\theta_4 &= \text{“The peers are never both in }\texttt{CLOSE_REQ}\text{.”}
\end{aligned} \tag{4}$$

Note that KORG expects that all its inputs (P, Q, and ϕ) are correct. However, since we test on an automatically extracted PROMELA program P, which may have some incorrect transitions when compared to the corresponding Canonical program, this assumption is broken. We therefore adapted KORG to work on incomplete or imperfect programs, while preserving the formal guarantees from the original paper (except for soundness, which depends on how different the extracted program is from the Canonical one).

VIII. EVALUATION

In this section we present an evaluation of both NLP tasks and attacker synthesis.

We use “Gold intermediary representation” to refer to the manual text annotations obtained using our protocol grammar presented in Section III. We use “Canonical FSM” to refer to the FSM which was derived from expert domain knowledge, the protocol RFC, and FSM diagrams in textbooks and literature.

A. Information Extraction Evaluation

We evaluate how much of the intermediary representation specified in Section III we can recover.

1) Methodology: We evaluate the output of the specification document parser in six different protocols: BGPv4, DCCP, LTP, PPTP, SCTP and TCP. We use a leave-one-out setup, by training on five protocols and testing on the remaining one. This means that no portions of a test protocol are observed during training. To artificially introduce more training sequences, we split recursive control statements into multiple statements at training time. At test time, we evaluate on each example once.

We evaluate predictions at the token-level and at the span-level. For tokens, we have 19 labels: beginning and inside tags for trigger, action, error, timer, transition and variable, as well as the outside label. We use standard classification metrics to measure the token-level prediction performance. We infer the control spans based on the indentation in the original documents. For identifying event and state references, we do direct lexical matching using a dictionary built on the definition tags described in Section III-A.

TABLE I: Average Results for Different Models

Model	Token-level			Span-level	
	Acc	Weighted F1	Macro F1	Strict	Exact
Rule-based	31.08	25.94	29.37	41.58	41.78
BERT-base	58.93	56.72	51.33	60.77	84.18
BERT-technical	62.38	60.31	52.50	62.84	83.81
LINEARCRF	58.95	56.61	49.58	63.98	85.65
LINEARCRF+R	58.60	56.79	50.62	63.52	85.18
NEURALCRF	**64.42**	**64.18**	**54.95**	**68.81**	**86.83**
NEURALCRF+R	62.79	62.50	53.64	66.22	86.10

To evaluate spans, we use the metrics introduced for the International Workshop on Semantic Evaluation (SemEval) 2013 task on named entity extraction [41]. We use the SemEval evaluation script on our data. In this case, we have six span types, plus the outside tag. The metrics are outlined below.

1) **Strict** matching, with exact boundary and type.
2) **Exact** boundary matching, regardless of the type.
3) **Partial** boundary matching, regardless of the type.
4) **Type** overlap between the tagged span and the Gold span.

We use the LINEARCRF provided by the pystruct library [42], which uses a structured SVM solver using Block-coordinate Frank-Wolfe [43], and use the default parameters during training. We implemented the NEURALCRF model using the transformers library [44] and PyTorch [45], and learn the model using the adaptive gradient algorithm Adam, with decoupled weight decay [46]. We initialize the BERT encoder with the parameters resulting from our pre-training stage described in Section IV, which further pre-trains BERT on technical documents. We use a learning rate of 2e-5 and 50 hidden units for the BiLSTM layer. For BERT, we use the standard parameter settings, and a maximum sequence length of 512. We randomly sample 10 percent of the training data to set aside as a development set, which we use to perform early stopping during training, using a patience of 5 epochs.

2) Segmentation strategies: We evaluate different segmentation strategies to create the base textual unit in our sequence-to-sequence models: segmenting by token, chunk, and phrase. For segmenting chunks, we use an off-the-shelf chunker [47]. For segmenting phrases, we split the text on periods, colons, semi-colons and newline markers, as well as on a set of reserved words corresponding to conditional statements (e.g. if, then, when, while). We find that segmenting by chunks yields the best token-level results (Weighted F1 of 61.25), but segmenting by phrases gives us better span-level results (Strict matching of 63.98, and Exact boundary matching of 85.56). Detailed results for these models are in the Appendix, in Table VI. Moving on, all evaluations are done using the phrase segmentation strategy.

3) Extraction models: We evaluate the two models proposed in Section V, and obtain a significant improvement with respect to a rule-based baseline that applies the rules outlined in Section V-C directly, without any learning. In addition, we test a BERT model by removing the BiLSTM CRF Layer, both with and without the pre-training strategy introduced in Section IV. Average results can be observed in Table I. When pre-training on technical documentation is not done, we use the BERT model trained on BookCorpus and Wikipedia. Here, we

can appreciate both the advantage of the technical embeddings, as well as the advantage of the BiLSTM CRF layer. We find that leveraging expressive neural representations for sequence-to-sequence models is advantageous for this task. Note that both the NEURALCRF and the LINEARCRF models make use of the full set of features introduced in Section V-B. Finally, we find that applying rules on top of our models to correct predictions does not improve their general performance.

In Table II, we show the individual performance for the six protocols and show that we obtain better performance using the NEURALCRF model for all protocols.

TABLE II: Results by Protocol for our Best Models

Protocol	LINEARCRF		NEURALCRF		# Control Statements
	Strict	Exact	Strict	Exact	
BGPv4	52.99	82.56	**57.34**	**86.86**	6
DCCP	69.74	92.73	**75.60**	**93.25**	150
LTP	67.25	**94.44**	**74.22**	94.41	65
PPTP	84.21	96.05	**87.34**	**98.73**	25
SCTP	52.21	65.49	**58.54**	**65.85**	19
TCP	57.46	**82.64**	**59.82**	81.90	31

4) FSM extraction: We compare both the NLP and the Gold extracted FSMs with the Canonical FSM in Table III, based on how many states and transitions are successfully extracted. Both the NLP and the Gold FSMs are extracted from the predicted/annotated intermediary representation introduced in Section III, by using the procedure outlined in Section VI. The results presented in Tables I and II correspond to how accurately we can recover this intermediary representation from the text, before we attempt to construct the FSM.

Note that even with Gold annotations, we are not able to extract all expected transitions because in some cases, the transitions are not explicit in the text or in other cases, our general grammar and extraction procedure are not able to capture the intended behavior. In all cases, we are able to recover all relevant states. Graphic visualizations for all resulting state machines can be found in the Appendix.

We manually analyzed all the partially correct, incorrect and missed transitions in Table III and found that, for the Gold FSM, they are caused by ambiguities in the RFC, or the information about some transition missing completely (67% for TCP and 96% for DCCP). The remaining errors are due to difficulties capturing complex logical flows using our method. The difference between the Gold FSMs and the predicted FSMs can always be attributed to errors in the text predictions.

For example, we notice that one incorrect behavior in the TCP Gold FSM is caused by ambiguity in the TCP RFC text. The only outgoing communication transition in the TCP Gold FSM from `SYN_SENT` sends `ACK` and goes to `SYN_RECEIVED`. The correct logic is to *receive* `SYN` first, before sending the `ACK` and transitioning. The TCP RFC does not textually mention the expected `SYN`. We only know to expect it because it is illustrated in Figure 6 of the RFC. We show more examples of FSM extraction errors in the Appendix.

5) Summary: In Tables I and II we evaluated how much of our intermediary representation we could extract from natural language, while in Table III we evaluated how much of the canonical FSM we recovered after running the extraction procedure in Section VI. There is not a one-to-one mapping between the intermediary representation extracted from the text and the resulting state machines for four reasons: 1) Not all Canonical FSM behaviors are clearly and unambiguously described in the text. 2) Some behaviors are mentioned more than once, giving us several opportunities to extract an expected transition. 3) We have annotated for a larger set of behaviors than needed to extract the communication transitions, we do this to be able to capture the language used to express FSM behaviors. 4) The metrics shown in Tables I and II are based on text span matching, however, we do not need to have a strict match in a text segment to successfully recover a behavior.

TABLE III: Transitions Extracted (Partially Correct means source and target state are correct, and at least one of the events on the edge is also correct).

TCP FSM	Canonical	Extracted	Correct	Partially Correct	Incorrect	Not Found
Gold	20	18	8	8	2	4
LINEARCRF		28	2	3	23	15
LINEARCRF+R		30	7	10	13	3
NEURALCRF		11	2	3	6	15
NEURALCRF+R		30	7	10	13	3
DCCP FSM	**Canonical**	**Extracted**	**Correct**	**Partially Correct**	**Incorrect**	**Not Found**
Gold	34	24	15	1	8	18
LINEARCRF		8	1	5	2	28
LINEARCRF+R		17	6	3	8	25
NEURALCRF		20	9	1	10	24
NEURALCRF+R		19	8	3	8	23

Our results show that learning technical word representations is useful for the task of extracting FSM information from protocol specifications. We demonstrate that we can recover a significant portion of the intermediary representation for the six evaluated protocols. Moreover, we show that we can recover partially correct FSMs by using the procedure outlined in Section VI. This analysis indicates that the grammar proposed in Section III can capture enough information to reconstruct a significant portion of the FSM, while being general enough to be applied to various protocols. Ambiguity and missed information in the RFCs result in transitions being partially/incorrectly recovered or missed. We show examples in the Appendix and discuss limitations in Section X.

B. Attacker Synthesis Evaluation

In this section we use KORG [18] to automatically synthesize attackers against the TCP and DCCP connection establishment and tear-down routines. Note we cannot extract Canonical FSMs like the ones manually derived and used by [18]. Our FSMs are partial, and we had to modify KORG to make it work with partial FSMs. We also had to modify KORG to support DCCP. We use our modified-KORG on all the models including the Canonical FSM and report these results below.

1) Methodology: We apply the same methodology to TCP and DCCP. We use the intermediary representations obtained with the models with best results for transition extraction (LINEARCRF+R and NEURALCRF+R), and Gold. We then extract FSMs and transpile them to PROMELA programs. All FSMs are presented in Appendix D.

We synthesize attackers that invalidate the properties from Eqn. 3 for TCP and Eqn. 4 for DCCP. Given a property and a PROMELA program, we can only use KORG if the program supports the property. We check what properties are supported by each program and present the results in Table IV.

We ask KORG to synthesize at most 100 attackers which we refer to as *candidate attackers* because they might not work against the protocol's Canonical PROMELA program. We check the *candidate attackers* against the corresponding Canonical PROMELA program; those that succeed are *confirmed attackers*. Unconfirmed attackers can be thought of as false positives.

TABLE IV: Properties Supported by Each PROMELA Program (checkmark/x-mark means property is supported/not supported).

TCP PROMELA program	$\models \phi_1$	$\models \phi_2$	$\models \phi_3$	$\models \phi_4$
Canonical	✓	✓	✓	✓
Gold	✓	✓	✗	✗
LINEARCRF+R	✓	✓	✗	✓
NEURALCRF+R	✓	✓	✗	✓
DCCP PROMELA program	$\models \theta_1$	$\models \theta_2$	$\models \theta_3$	$\models \theta_4$
Canonical	✓	✓	✓	✓
Gold	✓	✓	✓	✓
LINEARCRF+R	✓	✓	✓	✓
NEURALCRF+R	✓	✓	✓	✓

2) Supported Properties: Why do noisier models for TCP support a property the Gold model does not support? As shown in Table IV, the TCP Gold PROMELA program does not support property ϕ_4, while the TCP LINEARCRF+R and NEURALCRF+R PROMELA programs do. This might seem counterintuitive, as the Gold PROMELA program is derived from the Gold intermediary representation, which is theoretically less noisy than the LINEARCRF+R and NEURALCRF+R intermediary representations. Recall that ϕ_4 relates to connection tear-down from the TCP state `SYN_RECEIVED`. Upon investigation, we found that the TCP Gold PROMELA program violates ϕ_4 because of a single erroneous transition from `SYN_RECEIVED` to `CLOSE_WAIT`, and a missing `SYN?` event in the transition from `SYN_SENT` to `SYN_RECEIVED`. While the TCP LINEARCRF+R and NEURALCRF+R PROMELA programs contain similar erroneous transitions from `SYN_RECEIVED`, they nonetheless satisfy ϕ_4 because their erroneous transitions are never enabled. Basically, the same erroneous transition manifests in all three TCP PROMELA programs, but in the TCP Gold PROMELA program the code is reachable, while in the TCP LINEARCRF+R and NEURALCRF+R PROMELA programs it is unreachable.

Why do TCP and DCCP have such different support for properties intended to capture comparable behavior? In Table IV, we notice that the TCP Gold, LINEARCRF+R, and NEURALCRF+R PROMELA programs all violate ϕ_3, meaning they all have stuck states. For DCCP, all PROMELA programs support θ_1 and θ_3, meaning they never self-loop into a stuck state, or self-loop forever. Notably, either case would constitute a stuck state. It seems strange that the TCP PROMELA programs would be so susceptible to stuck states, while the DCCP PROMELA programs are apparently invulnerable to a closely related problem. Further investigation revealed that in contrast to TCP, DCCP does not support active/active establishment. Hence in order for a DCCP PROMELA program to support connection establishment, it requires both an active and a (matching) passive establishment routine. The DCCP Gold, LINEARCRF+R, and NEURALCRF+R PROMELA programs all capture the active establishment routine but not the passive one. Therefore, in all three PROMELA programs, none of the states containing self-loops are reachable, and so θ_1 and θ_3 are vacuously supported.

3) Examples of Attacks: Table V presents the *candidate attackers* generated for all programs and properties and false positives. We present some examples of *confirmed attackers*. Each example A is named following the convention $protocol.M.\alpha.N$, where *protocol* is TCP or DCCP, and A was the N^{th} PROMELA program output by KORG when given the *protocol* PROMELA program M and property α.

- *TCP.*NEURALCRF+R.ϕ_1.*32* injects a single `ACK` to Peer 2, causing a desynchronization between the peers which can eventually cause a half-open connection, violating ϕ_1.
- *DCCP.*LINEARCRF+R.θ_4.*32* injects and drops messages to and from each peer to first (unnecessarily) start and abort numerous connection routines, then guide both peers at once into `CLOSE_REQ`, violating θ_4.
- *DCCP.*NEURALCRF+R.θ_2.*96* is programmatically different from *DCCP.*LINEARCRF+R.θ_4.*32*, but violates θ_4 using basically the same approach.

TABLE V: Candidate and Unconfirmed Attacks Synthesized using each PROMELA Program P and Correctness Property φ. If P does not support φ, KORG cannot generate any attackers.

	Candidates Guided by φ.				Unconfirmed Candidates Guided by φ.			
TCP PROMELA program	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4
Canonical	1	9	36	17	0	0	0	0
Gold	2	0	0	0	0	0	0	0
LINEARCRF+R	1	0	0	0	0	0	0	0
NEURALCRF+R	1	0	0	0	0	0	0	0
DCCP PROMELA program	θ_1	θ_2	θ_3	θ_4	θ_1	θ_2	θ_3	θ_4
Canonical	0	12	0	1	0	0	0	0
Gold	0	1	0	1	0	0	0	0
LINEARCRF+R	8	2	13	1	2	0	13	0
NEURALCRF+R	5	2	9	1	2	0	9	0

4) Candidate Attackers: Why does property ϕ_2 not yield candidate attackers with TCP? In detail, ϕ_2 says: "if the two peers infinitely often revisit the configuration where the first is in `LISTEN` while the second is in `SYN_SENT`, then eventually the first peer will reach `ESTABLISHED`". In the TCP Gold, LINEARCRF+R, and NEURALCRF+R PROMELA programs, the tear-down routine is incomplete, so a connection cannot be fully closed. Moreover, the *timeout* transitions needed to abort a connection establishment are missing. Hence these PROMELA programs cannot capture the antecedent of ϕ_2, where two peers "infinitely often revisit the configuration where the first is in `LISTEN` while the second is in `SYN_SENT`". Since the PROMELA programs satisfy ϕ_2 only vacuously, they cannot be used by KORG to generate *candidate attackers* with ϕ_2.

Why does property ϕ_4 not yield candidate attackers with the TCP LINEARCRF+R *or* NEURALCRF+R PROMELA *programs?* In the TCP LINEARCRF+R and NEURALCRF+R PROMELA programs, `SYN_RECEIVED` is unreachable because of two missing transitions. Therefore, the TCP LINEARCRF+R and NEURALCRF+R PROMELA programs support ϕ_4 only vacuously and thus cannot be used with KORG to generate any candidate attackers using ϕ_4. Either of the missing transitions would fix the problem (the TCP Gold PROMELA program has one).

Why does property θ_3 not yield confirmed attackers with DCCP? As shown in Table V none of the candidate DCCP attackers generated using property θ_3 are confirmed. We investigated and found that for the canonical model the attacker can not violate θ_3, unless it is allowed to loop forever, i.e. the attack is continuous, a different (and less realistic) attacker model than the one we consider.

5) Comparison to Canonical Attacker Synthesis: For each attack synthesized using the TCP Gold, LINEARCRF+R, or NEURALCRF+R FSM, a similar attack was also synthesized using the TCP Canonical FSM. However, attacks found using TCP Canonical FSM exhibited five overarching strategies, of which attacks found using TCP Gold, LINEARCRF+R, or NEURALCRF+R FSM, exhibited only one.

Using the DCCP Gold, LINEARCRF+R, or NEURALCRF+R FSM, we find numerous attacks all of which passively spoof both peers in order to guide the peers into `CLOSE_REQ`×`CLOSE_REQ`. We cannot find active-spoofing attacks using the DCCP Gold, LINEARCRF+R, or NEURALCRF+R FSM, because these FSMs lack a functional passive establishment routine for active-spoofing to interact with. In contrast, all of the DCCP Canonical attacks use active spoofing. DCCP Canonical has both active and passive establishment, but in this case the SPIN model-checker finds counter-examples where the peers do passive establishment first.

We show examples of attacks synthesized with the canonical FSM, but not with the NLP generated FSMs in the Appendix.

6) Summary: Our NLP pipeline and attacker synthesis task successfully generated several confirmed attackers against two representative protocols: TCP and DCCP. However, our method depends on the accuracy of the NLP extraction task, the correctness of the extracted FSM, the quality of the selected properties, and the power of the attack synthesis tool. We discuss limitations and improvement directions in Section X.

IX. RELATED WORK

Below we present related works across three categories.

Logical Information Extraction. Rule-based systems like WHYPER [1] and DASE [2] identify sentences describing mobile application permissions and extract command-line input constraints from manual pages, respectively. Witte *et al*. [3] use rules over documentation and source code to create an ontology allowing the cross-linking of software artifacts.

Other works combine NLP with techniques from traditional software engineering and security. Lin *et al*. [11] infer protocol formats by combining NLP with program analysis. NLP has also been used to gather *threat intelligence* by interacting with botnets [48], logically contrasting CVEs [49], or analyzing bug reports in the context of data collected with a honeypot [50].

Ding and Hu [51] used pre-trained word embeddings to identify physical channels in IoT from application descriptions. Tian *et al*. [52] used pre-trained word vectors and other standard NLP features to compare security policy descriptions written in text in the context of IoT application authorization. Both works relied on off-the-shelf NLP tools, and worked over keywords in isolation, or over short and simple sentences.

Recently, Jero *et al*. [53] proposed a system to extract protocol rules from textual specifications for grammar-based fuzzing. They also took a zero-shot learning approach to adapt to protocols that are unseen at training time. However, they focused on a limited set of properties, and did not explicitly model the behavior of the protocol. More closely related to our work, Chen *et al*. [21] explored the use of NLP to discover logical vulnerabilities in payment services. They extended the FSMs for evaluated payment services by using the dependency parse tree of sentences in a developer guide to extract the parties involved in the process, as well as the content transmitted between them. To identify relevant sentences, they used word embeddings trained on relevant documentation. In our work, we also leverage word representations trained on in-domain data. However, we aim to reconstruct the full FSM from the text, while they relied on a manually implemented FSM. While their language analysis was done at the sentence-level, we predict logical flow structures that span multiple sentences.

Full Correctness Specification. Zhai *et al*. [54] automatically extract formal software specifications from comments in the implementation code. Zhang *et al*. [55] use NLP to extract LTL correctness specifications from prose policies for IoT apps. In contrast to our work, they assume that the actual software code is known ahead of time. Other related works infer abstract protocol implementations using network traces [4], [5], [7], program analysis [10], or model checking [13], [14]. These approaches rely extensively on input from human experts and do not easily generalize to new software or protocols.

Implementation Extraction. Yen *et al*. [56] explored the use of NLP techniques to map RFCs to protocol implementations. To do this, they manually engineer an existing semantic parser to handle networking-specific vocabulary, and translate individual sentences to logical forms that can then be mapped to executable functions. They include the spec author in the loop to disambiguate cases where the functionality is underspecified. They do not perform any task-specific learning, and they work at the sentence-level.

X. LIMITATIONS

In this section we discuss some of the limitations of our approach and directions for improvement.

Why our NLP models could not extract Canonical FSMs from RFCs. Canonical FSMs are created based not only on RFCs but also on input from experts with exposure to protocol implementations, and often also rely on analyzing the code [22], [57], [58]. RFCs contain ambiguities, unspecified

behaviors that human experts solve in creating the Canonical FSM [16], [56], or simply missing information. Thus, unlike traditional NLP semantic parsing problems [59], [60], [24], which study methods for translating natural language into a complete formal representation, in our setting there is not a complete one-to-one translation between the text and the FSM. We address this challenge by defining an intermediary semantic representation that can be extracted unambiguously from the text, and then use this intermediary representation as the basis for the FSM extraction. The ground truth for these intermediary representations is what we refer to as Gold intermediary representations.

One avenue to extract better FSMs, possibly canonical ones, is to solve ambiguities existing in the text by leveraging human expertise. This can be done by using NLP methods that exploit unlabeled data and human knowledge. A potential direction for improvement is to design learning objectives that, in addition to exploiting domain-specific corpora, can augment the intermediary representations and constraint the predictions using structured domain knowledge.

Limitations of attacker synthesis with partial FSMs. The partial FSMs produced by the NLP pipeline combined with the FSM extraction algorithm exhibit numerous errors, which impacted our ability to use these FSMs for attacker synthesis. Some attacks which could be found using the Canonical FSMs were not found using the partial FSMs, and, some of the attacks found using the partial FSMs were not confirmed on the Canonical FSMs. There are two causes for these mistakes: missed transitions and incorrect transitions.

One direction to address these limitations is by leveraging protocol completion [61], where given an incomplete protocol FSM and some properties, the goal is to strategically add transitions so that the completed FSM supports all the properties. Their solution relied on counterexample-guided inductive synthesis (CEGIS) [62]. Our problem is a little more difficult, because in addition to missing transitions, we also need to worry about incorrect transitions, so the approach used in [61] would need to be modified such that the solver is also allowed to delete or edit transitions. Another approach would be to leverage prior work in automatic program repair [63].

Selecting properties. The attackers we find are driven by the selection of properties that the Canonical and extracted FSMs support. For attacker synthesis, the most useful properties describe critical functionality of a protocol, for example, that it must reliably open and close connections, or that it must not deadlock. We also prefer properties that are not too implementation-specific, because there are multiple ways to implement a protocol while still achieving the intended functionality, as illustrated for Alternating Bit Protocol in [61].

Protocol correctness properties should be provided by protocol designers. Unfortunately, protocols are often implemented and deployed before textual specifications are published. This is the case with QUIC, which was deployed without detailed public specification or analysis. The authors of a 2015 QUIC security analysis [57] mention that they relied on code and discussion with protocol developers to derive a protocol description as the available documentation was insufficient.

Extracting properties. While several NLP works looked at converting natural language statements into properties expressed in temporal logic, RFCs do not have a dedicated section detailing protocol correctness properties in an explicit and succinct way. Instead, humans identify these properties by observing the behaviors emerging from the specification and inferring the intent behind them, or by reading prose descriptions of the developer's intention. One promising approach is to study these inference processes and formulate them as NLP problems that take into account the functionality described by the protocol as part of the input. Rather than converting the explicit textual statement into properties, one can define an abductive process that infers relevant desired properties of the extracted model and rely on textual description of protocol tests for specifications that offer similar functionality.

Limitations of KORG. KORG was not designed for broken or partial FSMs (expressed as PROMELA programs), that might violate or vacuously satisfy the provided properties. In these cases it might generate no candidates whatsoever, or some candidates, none of which are confirmed. Also, KORG outputs many identical or similar candidates, but we would prefer a diversity of candidate attackers so that if some are not confirmed, perhaps others will be. The problem of determining when two candidate attackers are similar reduces to defining an equivalence relation on counterexamples, as studied in [64]. Perhaps such work could be leveraged to quotient KORG's search-space by the equivalence class of the candidates it already found, resulting in a diversity of attackers.

Generalizability to other RFCs. While we consider a set of 6 different protocols, including TCP (one of the most well-known and used protocols), there are further aspects we did not consider in this work. One such aspect is considering changes in RFCs. We believe that one promising direction for investigating changes in RFCs and impact on FSMs is investigating congestion control protocols that share a common approach in detecting congestion, where newer refinements were proposed enhancing the original protocol. We expect that while we can use the same technical domain knowledge we might need to update our grammar to handle changes.

We did not consider secure protocols in this work. Note that QUIC was just recently standardized in May 2021, as RFC 9000 [65]. Here we can focus on RFC drafts changes for QUIC and TLS 1.3, particularly the key exchange aspects. Secure protocols will most likely require us to refine both the grammar and the domain knowledge we built for this work.

ACKNOWLEDGEMENTS

This work was supported by NSF grants CNS-1814105, CNS-815219, and GRFP-1938052. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We thank our reviewers and shepherd for their constructive feedback.

APPENDIX

Alg. EXTRACTTRAN(xml, T)

Inputs:

- xml intermediary representation
- `transition` block T, contained with xml.

Outputs:

- A set T_{T} containing potential transitions $s \xrightarrow{\ell} s'$ described in and around the block T.

---

1. *from* := EXTRACTSOURCESTATE(T, xml)
2. *to* := EXTRACTTARGETSTATE(T, xml)
3. *int* := EXTRACTINTERMEDIARYSTATES(T, xml)
4. C := CLOSESTCONTROLCONTAINING(T, xml)
5. *outer* := [ ]
6. If (*to* = *null* and *from* = *null*):
   1. *to* := SCANCHILDRENFORTARGETSTATE(T)
7. If (*to* = *null* or *from* = *null*):
   1. *outer* := SCANCONTEXTFORSTATES(C, T)
8. $\ell := \epsilon$
9. $i := 1$
10. While (not SEARCHEDENOUGH(ℓ, *outer*, i, or, C)):
    1. If $\ell = \epsilon$:
       /* ℓ is the transition label, brk indicates if the source states are given outside C, and or indicates if ℓ is of the form "ℓ_0 or ℓ_1 or ... or ℓ_k". */
       1. (ℓ, brk, or) := EXTRACTTRANLBL(T, C).
    2. If *outer* = [ ] and (*from* = *null* or *to* = *null*):
       1. *outer* := SCANCONTEXTFORSTATES(C, T)
    3. C := CLOSESTCONTROLCONTAINING(C, xml)
    4. $i++$
11. (*fromS*, *to*) := FIXFROMTOSTATES(*from*, *to*, *outer*)
12. If *int* $\neq$ [ ]:
    1. $(\ell_0, ..., \ell_j)$:= PARTITIONLABELACROSS(ℓ, *int*)
    2. Let $S_0 := \{s_0 \xrightarrow{\ell_0} s_1 \mid s_0 \in \mathit{fromS}\}$
    3. Let $S_1 := \{s_1 \xrightarrow{\ell_1} s_2, ..., s_j \xrightarrow{\ell_j} \mathit{to}\}$
    4. Return $S_0 \cup S_1$
13. If brk = *true*:
    a) C := CLOSESTCONTROLCONTAINING(T, xml)
    b) C′ := CLOSESTCONTROLCONTAINING(C, xml)
    c) fromS := SCANCONTEXTFORSTATES(C′, C)
14. Return $\{s_0 \xrightarrow{\ell} \mathit{to} \mid s_0 \in \mathit{fromS}\}$.

A. Grammar Examples

Figure 7 shows an example of an annotated block from the TCP RFC. Here, we can observe a list of events within one control statement

B. Segmentation Results

In Table VI, we show the detailed performance of different segmentation strategies to create the base textual unit in our sequence-to-sequence models.

```
<control>
  <trigger>
    if active and the foreign socket is
    specified,
  </trigger>
  <action type="issue">
    issue <arg>a <ref_event id="10">SYN</ref_event>
        segment</arg>.
  </action>
  <variable>
    An initial send sequence number (ISS) is
    selected.
  </variable>
  <action type="send">
    A <arg><ref_event id="10">SYN</ref_event>
        segment of the form
    <SEQ=ISS><CTL=SYN></arg> is sent.
  </action>
  <variable>
    Set SND.UNA to ISS, SND.NXT to ISS+1,
  </variable>
  <transition>
    enter the <arg_target><ref_state
        id="2">SYN-SENT</ref_state>
        state<arg_target>
  </transition>
</control>
```

Fig. 7: Example of flow control annotations for TCP.

C. FSM Extraction Errors Examples

In Table VII, we show examples of FSM extraction errors.

D. Finite State Machine Figures

We present FSMs for TCP and DCCP in Figures 8,9 and 10. Note that in the DCCP diagrams we omit the states CHANGING, STABLE, and UNSTABLE, which are described in the RFC but are (a) unreachable dead code in all the extracted FSMs and (b) unrelated to the connection routine. We use $*$ as a wild-card, ! to mean *send*, ? to mean *receive*, == to denote variable-*reading*, and := to denote variable-*writing*.

E. Attack Synthesis Errors Examples

Below we show examples of attacks that are synthesized with the canonical FSM, but not found with the NLP models.

TCP.Canonical.3.9 spoofs both peers passively. When tested against ϕ_3, the attack causes the peers to end up in a deadlock in `SYN_RECEIVED×SYN_RECEIVED`. None of the TCP Gold, LINEARCRF+R, or NEURALCRF+R attacks do passive spoofing; nor do any of them cause the peers to deadlock in `SYN_RECEIVED×SYN_RECEIVED`.

DCCP.Canonical.2.18 spoofs both peers actively. When tested against θ_2, the attack causes the peers to navigate to `RESPOND×RESPOND`. On the way, they enter `TIME_WAIT×TIME_WAIT`, violating θ_2. None of the DCCP Gold, LINEARCRF+R, or NEURALCRF+R attacks do active spoofing; nor do any of them conclude in the state `RESPOND×RESPOND`.

TABLE VI: Average Results for Different Segmentation Strategies (LINEARCRF)

Segmentation	Token-level			Span-level			
	Acc	Weighted F1	Macro F1	Strict	Exact	Partial	Type
Token	60.37	59.58	44.76	31.36	36.14	59.78	58.81
Chunk	**62.02**	**61.25**	46.36	33.48	39.11	62.19	62.14
Phrase	58.95	56.61	**49.58**	**63.98**	**85.65**	**85.65**	**63.98**

TABLE VII: Examples of FSM Extraction Errors

FSM	Transition	Error Type	Reason	Text Excerpt
Gold TCP	`FIN_WAIT_1` $\xrightarrow{\texttt{FIN!}}$ `LAST_ACK`	Not Found	Target state not explicit	CLOSE-WAIT STATE: Since the remote side has already sent FIN, RECEIVEs must be satisfied by text already on hand, but not yet delivered to the user.
Gold DCCP	`PARTOPEN` $\xrightarrow{\texttt{DCCP-CLOSE?}}$ `OPEN`	Incorrect	Text is ambiguous	The client leaves the PARTOPEN state for OPEN when it receives a valid packet other than DCCP-Response, DCCP-Reset, or DCCP-Sync from the server.
LINEARCRF+R and NEURALCRF+R	`SYN_SENT` $\xrightarrow{\texttt{SYN!ACK!}}$ `SYN_RECEIVED`	Partially Recovered (expected `SYN?ACK!`)	Receive action is not explicit	If the state is SYN-SENT then enter SYN-RECEIVED, form a SYN,ACK segment and send it.

REFERENCES

[1] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX Security*, 2013.

[2] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: document-assisted symbolic execution for improving automated software testing," in *ACM/IEEE ICSE*, 2015.

[3] R. Witte, Q. Li, Y. Zhang, and J. Rilling, "Text Mining and Software Engineering: an Integrated Source Code and Document Analysis Approach," *IET Software*, 2008.

[4] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *IEEE SP*, 2009.

[5] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: a probabilistic approach," in *ACNS*, 2011.

[6] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering," in *ACM CCS*, 2009.

[7] C. Y. Cho, E. C. R. Shin, D. Song *et al.*, "Inference and analysis of formal models of botnet command and control protocols," in *ACM CCS*, 2010.

[8] W. Cui, J. Kannan, and H. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX Security*, 2007.

[9] N. Kothari, T. D. Millstein, and R. Govindan, "Deriving state machines from tinyos programs using symbolic execution," in *IPSN*, 2008.

[10] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *USENIX Security*, 2011.

[11] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, 2008.

[12] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *ACM CCS*, 2007.

[13] D. Lie, A. Chou, D. Engler, and D. L. Dill, "A simple method for extracting models from protocol code," in *IEEE ISCA*, 2001.

[14] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *ICSE*, 2000, pp. 439–448.

[15] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[16] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *IEEE/IFIP DSN*, 2015.

[17] Q. Kang, J. Xing, and A. Chen, "Automated attack discovery in data plane systems," in *12th USENIX Workshop on Cyber Security Experimentation and Test (USENIX 19)*, 2019.

[18] M. von Hippel, C. Vick, S. Tripakis, and C. Nita-Rotaru, "Automated attacker synthesis for distributed protocols," in *Computer Safety, Reliability, and Security*, 2020, pp. 133–149.

[19] T. F. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[20] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in TCP congestion control using a model-guided approach," in *NDSS*, 2018, best paper award.

[21] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang, K. Chen, and W. Zou, "Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis," in *USENIX Security*, 2019.

[22] A. Peterson, S. Jero, E. Hoque, D. Choffnes, and C. Nita-Rotaru, "aBBRate: Automating BBR attack exploration using a model-based approach," in *RAID*, 2020.

[23] R. J. Mooney, "Learning for semantic parsing," in *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 311–324.

[24] J. Cheng, S. Reddy, V. Saraswat, and M. Lapata, "Learning structured natural language representations for semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 44–55. [Online]. Available: https://www.aclweb.org/anthology/P17-1005

[25] M. Gardner, P. Dasigi, S. Iyer, A. Suhr, and L. Zettlemoyer, "Neural semantic parsing," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 17–18. [Online]. Available: https://www.aclweb.org/anthology/P18-5006

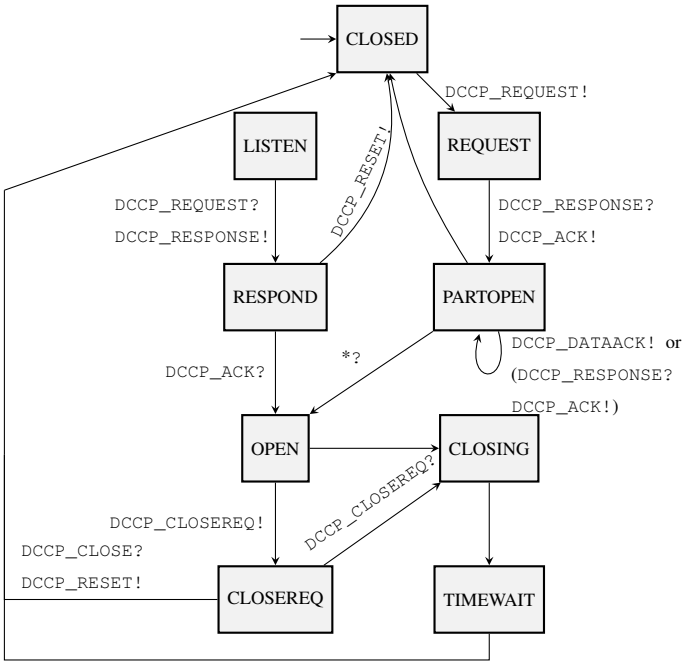
[26] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional LSTM-CNNs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 357–370, 2016.

[27] D. Wadden, U. Wennberg, Y. Luan, and H. Hajishirzi, "Entity, relation, and event extraction with contextualized span representations," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5784–5789.

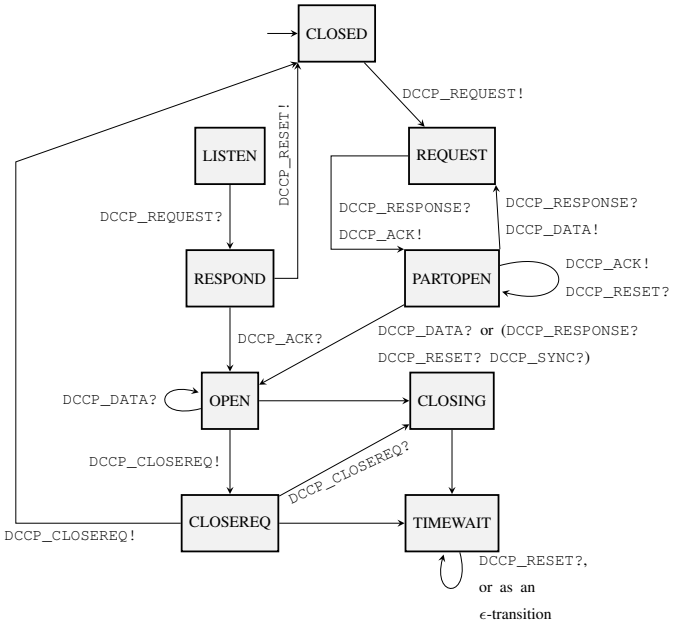
[28] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[29] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

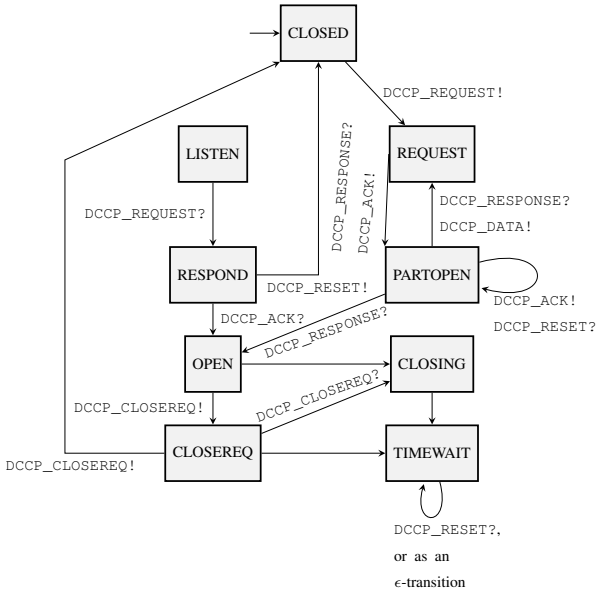
[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality,"



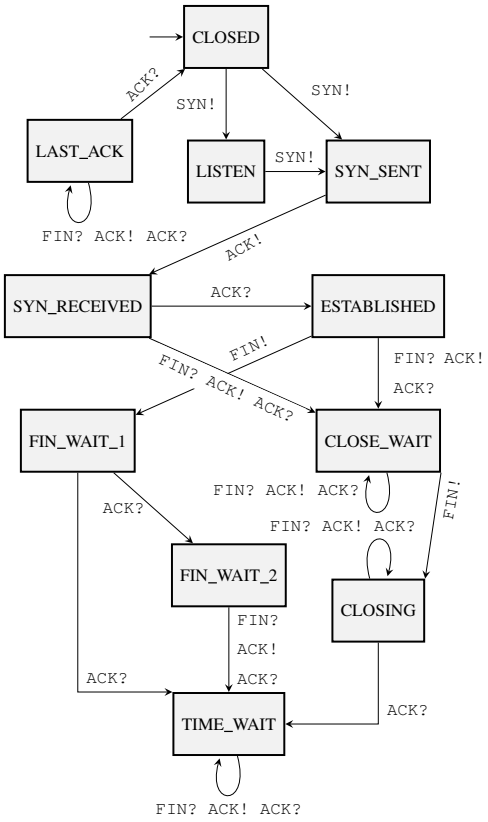
(a) **DCCP Gold FSM**.



(b) **DCCP NEURALCRF+R FSM**.



(c) **DCCP LINEARCRF+R FSM**.



(d) **TCP Gold FSM**.

Fig. 8: DCCP Gold, NEURALCRF+R, and LINEARCRF+R FSMs; and TCP Gold FSM.

in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[31] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter*

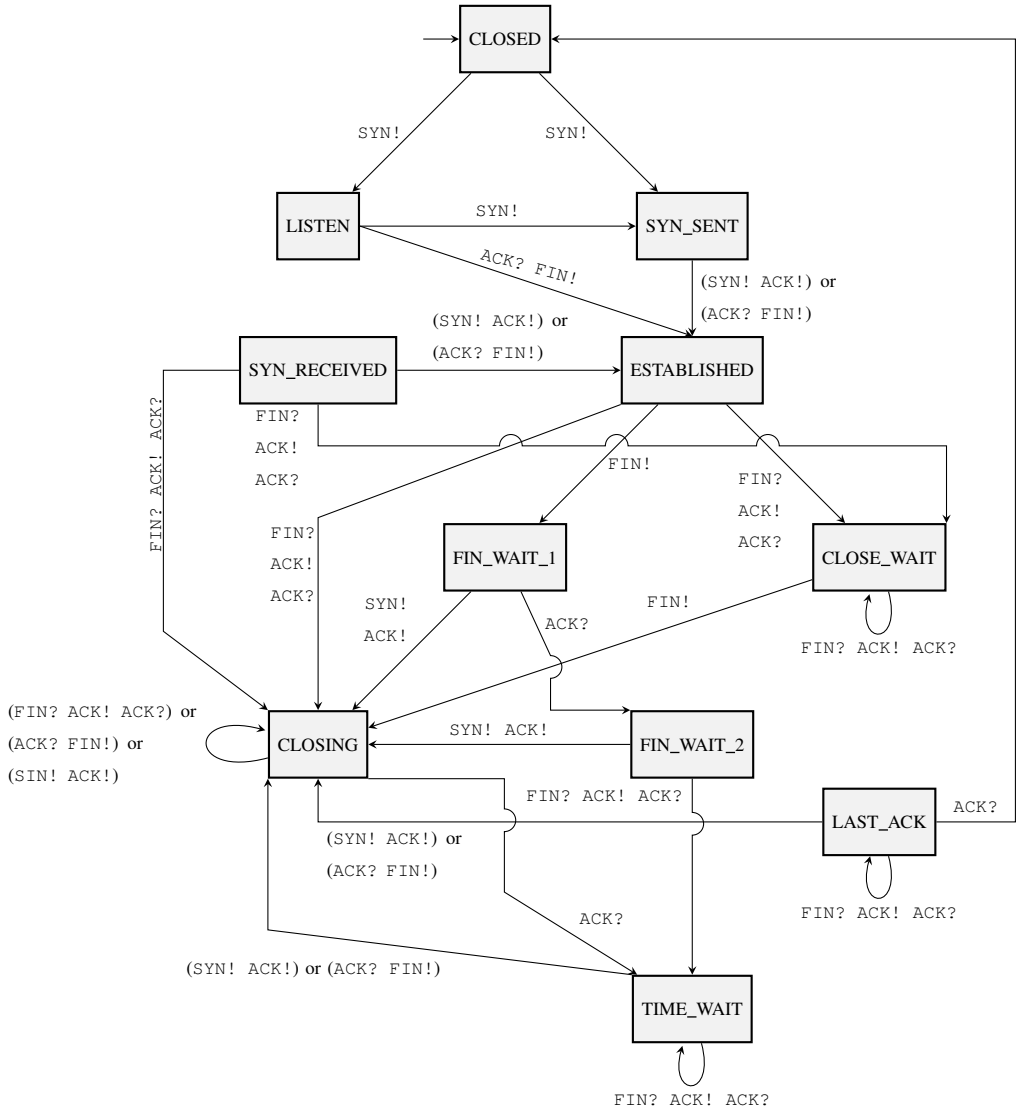


Fig. 9: **TCP NEURALCRF+R and LINEARCRF+R FSM**. (They are identical.)

of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[34] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Jul. 2020, pp. 8342–8360.

[35] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 282–289.

[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

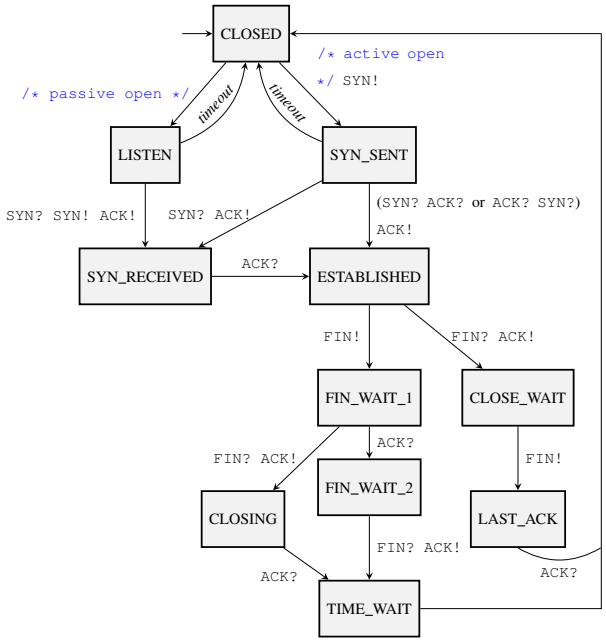
[37] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Jun. 2016, pp. 260–270.

[38] X. Ma and E. Hovy, "End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1064–1074.

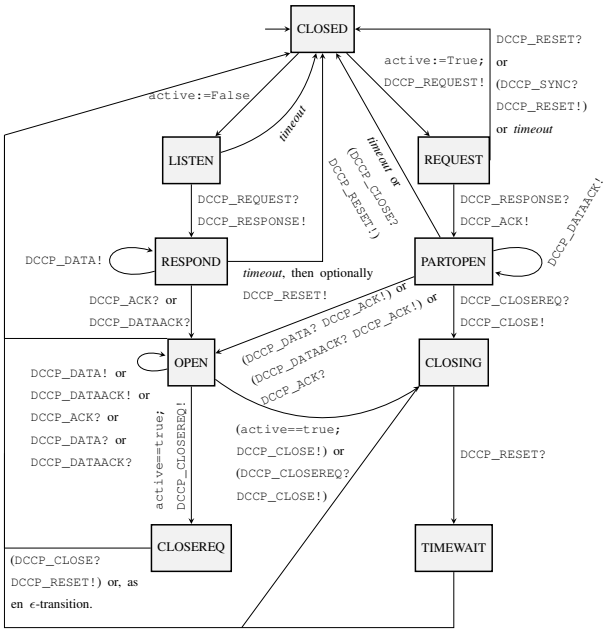
[39] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. Peters, M. Schmitz, and L. S. Zettlemoyer, "AllenNLP: A deep semantic natural language processing platform," in *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, 2017.

[40] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*, 1989.

[41] I. Segura-Bedmar, P. Martínez, and M. Herrero-Zazo, "SemEval-2013 task 9 : Extraction of drug-drug interactions from biomedical texts (DDIExtraction 2013)," in *Second Joint Conference on Lexical and*



(a) **TCP Canonical FSM**. User commands are shown in `/* blue */`; you can view these as comments in the code (having no bearing on its logic).



(b) **DCCP Canonical FSM**.

Fig. 10: TCP and DCCP Canonical FSMs.

*Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*. Atlanta, Georgia, USA: Association for Computational Linguistics, Jun. 2013, pp. 341–350.

[42] A. C. Müller and S. Behnke, "pystruct - learning structured prediction in python," *Journal of Machine Learning Research*, vol. 15, no. 59, pp. 2055–2060, 2014. [Online]. Available: http://jmlr.org/papers/v15/mueller14a.html

[43] M. Frank and P. Wolfe, "An algorithm for quadratic programming," *Naval Research Logistics Quarterly*, vol. 3, no. 1-2, pp. 95–110, 1956.

[44] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *EMNLP: System Demonstrations*, Oct. 2020.

[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.

[46] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *ICLR*, 2019.

[47] Apache Software Foundation, "openNLP Natural Language Processing Library," 2014, http://opennlp.apache.org/. [Online]. Available: http://opennlp.apache.org/

[48] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield, "To catch a predator: A natural language approach for eliciting malicious payloads." in *USENIX Security Symposium*, 2008, pp. 171–184.

[49] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 869–885.

[50] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and securing device vulnerabilities through automated bug report analysis," in *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[51] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *ACM CCS*. Association for Computing Machinery, 2018.

[52] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *USENIX Security*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 361–378.

[53] S. Jero, M. L. Pacheco, D. Goldwasser, and C. Nita-Rotaru, "Leveraging textual specifications for grammar-based fuzzing of network protocols," in *Innovative Applications of Artificial Intelligence (IAAI)*, 2019.

[54] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2s: translating natural language comments to formal program specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 25–37.

[55] S. Zhang, J. Zhai, L. Bu, M. Chen, L. Wang, and X. Li, "Automated generation of ltl specifications for smart home iot using natural language," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 622–625.

[56] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-automated protocol disambiguation and code generation," in *SIGCOMM*, 2021.

[57] A. Boldyreva, R. Lychev, S. Jero, and C. Nita-Rotaru, "How secure and quick is QUIC: Security and performance analyses," in *IEEE Symposium on Security and Privacy*, 2015, iETF/IRTF Applied Networking Research Prize 2016.

[58] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," in *SIGCOMM*, 2019, pp. 227–240.

[59] R. J. Kate, Y. W. Wong, and R. Mooney, "Learning to transform natural to formal languages," in *AAAI*, 2005.

[60] T. Kwiatkowksi, L. Zettlemoyer, S. Goldwater, and M. Steedman, "Inducing probabilistic ccg grammars from logical form with higher-order unification," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Oct. 2010, pp. 1223–1233.

[61] R. Alur and S. Tripakis, "Automatic synthesis of distributed protocols," *Acm Sigact News*, vol. 48, no. 1, pp. 55–90, 2017.

[62] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements," in *Haifa Verification Conference*. Springer, 2014, pp. 75–91.

[63] B. Bonakdarpour and S. S. Kulkarni, "Automated model repair for distributed programs," *ACM SIGACT News*, vol. 43, no. 2, pp. 85–107, 2012.

[64] C. Vick, E. Kang, and S. Tripakis, "Counterexample classification," *arXiv preprint arXiv:2108.00885*, 2021.

[65] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," *Internet Engineering Task Force, Internet-Draft draftietf-quic-transport-17*, 2021.