ARTIFACT EVALUATED USENIX AVAILABLE

ARTIFACT EVALUATED USENIX FUNCTIONAL

ARTIFACT EVALUATED USENIX REPRODUCED

EYWA: Automating Model Based Testing using LLMs

Rajdeep Mondal[1*] Rathin Singha[1*] Todd Millstein[1] George Varghese[1]
Ryan Beckett[2] Siva Kesava Reddy Kakarla[2]

[1]*UCLA* [2]*Microsoft Research*

Abstract

Model-based testing (MBT), whereby a model of the system under test is analyzed to generate high-coverage test cases, has been used to test protocol implementations. A key barrier to the use of MBT is the need for users to understand protocol RFCs in detail to create a compliant model.

Our new approach to MBT uses LLMs to automatically build rich models of intended protocol behavior from knowledge embedded in Request for Comments documents (RFCs), blogs, and other natural language sources. Our approach addresses key challenges with using LLMs, including hallucinations and their inability to monolithically generate complex protocol models. We realize our approach through a novel protocol testing framework EYWA, and demonstrate its effectiveness through extensive case studies of DNS and BGP, and a smaller study of SMTP. Despite minimal user effort, applying EYWA enabled the discovery of 33 unique bugs across widely used DNS, BGP, and SMTP implementations, 16 of which were previously undiscovered despite extensive prior testing with manually crafted models.

1 Introduction

Networked systems depend on the correct and performant operation of hundreds of protocols from the physical to the application layer. Despite standardization, protocol hardware and software implementations still frequently suffer from bugs due to coding mistakes, misinterpretations of specifications, unsound optimizations, unforeseen corner cases, and poor data structure choices. Bugs in protocol implementations have led to security vulnerabilities [9, 21–23, 49, 103, 106], outages [4, 28, 29, 42, 92, 109, 127], and performance problems [44]. For example, in 2022, a bug in Akamai's DNS software caused a global outage for nearly 30,000 sites [92]. Recently, mishandling of a corrupted BGP attribute in router software resulted in the protocol spreading invalid routes and ultimately caused BGP sessions in remote networks to shut down [29].

[*]These authors contributed equally.

Model-Based Testing for Protocols: To test the correctness of protocol implementations, model-based testing (MBT) has emerged as a highly effective family of techniques. Rather than writing test cases manually, MBT requires the user to formulate a simplified model of a protocol – such as a logical specification, abstract state machine, or a reference implementation – and then uses existing program analysis techniques to generate tests from the model. Researchers have used MBT to find numerous bugs in implementations of QUIC [90, 91], BGP [119], DNS [49, 89], and various LTE [45] protocols, including proprietary systems where source code is unavailable but standards and documentation exist.

Our point of departure is a recent form of MBT that demonstrates how carefully constructed protocol models can drive high-coverage testing of real implementations [49, 119]. In this approach, the user implements an executable model of the protocol's RFC semantics. The approach then uses symbolic execution [10, 38] to systematically enumerate test cases that cover distinct execution paths through the model, in order to cover a wide variety of protocol behaviors. Since these models are typically orders of magnitude simpler than full protocol implementations, symbolic execution scales more effectively on them, mitigating the infamous path explosion problem [10] that frequently limits direct analysis of the source code. Additionally, instead of comparing implementation behavior against a single reference model, this approach performs differential testing — it tests multiple independent protocol implementations and flags any behavioral differences as potential bugs.

Challenges in Applying Model-Based Testing:. While this approach to MBT has successfully found critical protocol implementation bugs, it has several challenges:

C1 - Manual model creation: Users must first build a detailed model of the protocol under test by studying multiple RFCs to understand intended behavior.

C2 - Complex input requirements: Moreover, protocol inputs such as DNS queries and BGP routes often have complex requirements and data dependencies that must be satisfied. The user must understand these requirements and make them explicit in the model, or else symbolic execution will produce many invalid tests.

C3 - Statefulness: A test for a stateful protocol involves not only a specific input but also the current state of the protocol. To execute such a test, the implementation must first be driven into the corresponding state. For example, while DNS nameservers are stateless, BGP's behavior depends on earlier route announcements, and protocols like SMTP and TCP use state machines that demand specific input sequences to reach deeper states.

Leveraging LLMs for Protocol Model Construction. To mitigate these challenges, we ask: *"Can large language models (LLMs) reduce the cost of model-based testing for protocols?"* We demonstrate that they can, by leveraging recent advances in LLMs and their extensive embedded knowledge. Through training, LLMs have absorbed protocol expertise from RFCs, standards, networking forums, blogs, and other technical documents. Yet, applying LLMs to network protocol testing in practice raises two additional challenges:

C4 - Large models: Many protocols are highly complex and comprise multiple components. Asking an LLM to generate an end-to-end protocol model in a single shot is impractical. In our experience, doing so will lead the LLM to gloss over important implementation details, hence limiting the ability to generate interesting tests.

C5 - LLM errors: LLMs make mistakes and can easily hallucinate to produce flawed protocol models.

Our Approach. In this paper, we describe an approach that uses LLMs to significantly reduce the user burden for MBT while addressing each of the above challenges:

S1 - Modularized code synthesis: We introduce a modular approach to building executable protocol models for MBT. Specifically, the user has to (1) decide which protocol components they wish to test, (2) provide a short description of each component (or "module"), along with their input and output types, and (3) provide a graph that specifies how the different components should be composed together. Our approach then uses LLM invocations to both build implementations of the individual modules and stitch the modules together to produce the end-to-end protocol model, based on the provided graph. As we demonstrate, this approach makes LLM-based model generation feasible even for complex protocols, thereby addressing challenges C1 and C4. Modularization also naturally accommodates validity requirements on protocol inputs: the user simply specifies a module that should enforce these constraints, so that only valid inputs flow to the rest of the components of the model. This addresses challenge *C2*.

Our modular approach also allows the user to retain precise control over the structure of the resulting model, despite outsourcing its creation to the LLM. For example, the user can decide to test only a few components of a protocol that are of particular interest, and then later easily add additional ones. For each component, the ability to specify the input-output types also enables the user to easily focus on specific parts of the component or to elide lower-level details such as the bit-level data encoding.

S2 - State graph to represent state: For stateful protocols, it is not enough to create the model and generate tests, as the test cases are state-input pairs. In order to run these tests, we also need to drive the protocol implementation to the state required by each test case. To that end, we use a separate invocation of the LLM to generate a *state graph* of the protocol, which defines the protocol's states and transitions. For each test case, we then use this graph to search for an input sequence that drives the protocol implementation to the required starting state for testing. This addresses challenge *C3*.

S3 - Hallucination as a feature, not a bug: One approach to reducing hallucinations is to fine-tune LLMs with domain-specific networking knowledge. However, hallucinations are an inherent aspect of probabilistic generation and cannot be fully eliminated. Instead of resisting this property, we design our approach to embrace it. Prompting LLMs to generate multiple protocol models naturally introduces variety, and symbolic execution over these models yields a wide range of protocol tests that explore a broader behavioral space. Because we employ differential testing to identify inconsistencies across protocol implementations, our approach does not use the model as an oracle — we use the model to generate tests but do not care what the model returns for those tests. In this way, our approach remains robust even when models produce erroneous outputs. This addresses challenge *C5*.

EYWA: Architecture and Workflow. We have instantiated our approach in a tool called EYWA, which is implemented as a Python library. The user provides the per-component information along with a dependency graph. EYWA then modularly builds the model as executable C code using multiple invocations of an LLM. EYWA simultaneously compiles a *symbolic test harness* to integrate this model with the Klee [10] symbolic execution engine. The tool then invokes Klee to systematically enumerate test cases for the protocol, which the user can then run against real protocol implementations. For stateful protocols, EYWA also augments each test case with the input sequence required to drive the protocol to the necessary starting state before evaluation.

We evaluate the effectiveness of EYWA through an extensive case study of the DNS and BGP protocols, and a smaller study of SMTP as a canonical stateful protocol. Compared to prior model-based testing approaches for DNS and BGP that required months of effort to craft models that accurately encode the RFC intent [49, 119], EYWA constructed similar models in minutes with minimal user input. Using EYWA, we revealed **33** unique bugs across popular DNS, BGP, and SMTP implementations. Of these bugs, **16** were previously unknown despite the extensive testing by prior work.

Contributions: To summarize, our contributions are:

- A new approach to protocol testing that combines LLMs with model-based testing and differential testing.
- EYWA, a protocol testing library based on our approach

that provides novel abstractions for defining and composing protocol modules.

- Techniques to compile EYWA specifications by generating specialized LLM prompts and combining the resulting models with a *symbolic test harness*.
- Case studies that show EYWA's utility – they found 32 unique bugs across popular DNS, BGP, and SMTP implementations, including 15 previously undiscovered bugs.
- A small case study of SMTP showing how to handle stateful protocols by using the LLM to generate a state machine graph of the protocol, and traversing it to automatically drive an implementation to desired states.

Ethics: This work raises no ethical concerns.

2 Motivation and Overview

To motivate EYWA, we demonstrate its use to automatically generate tests for Domain Name System (DNS) nameservers. We selected DNS because its analysis has been the target of prior model-based testers. SCALE [49] was the first to apply MBT to the DNS protocol and found dozens of new correctness, performance, and security bugs in widely used DNS implementations such as BIND, Knot, PowerDNS, and CoreDNS. However, SCALE required extensive manual mathematical formulation of the DNS RFCs [48] and later over 3 months of effort to manually construct an executable model of DNS [49]. We give an overview of EYWA in the context of DNS testing. Our later experiments show that EYWA achieves results similar to SCALE, including finding 15 new DNS bugs that SCALE missed, all with minimal user effort.

2.1 Testing the DNS with EYWA

To test a protocol, users must describe the parts of that protocol that they wish to model using EYWA's Python library abstractions. Specifically, users define protocol-specific objects (*e.g.*, state) and formats (*e.g.*, headers, inputs) as well as "functions" over these objects. For instance, Figure 1 shows an example model defined in the EYWA library to test the part of the lookup logic in DNS that determines if a DNS query matches a resource record defined in a zone configuration.

EYWA follows a modular approach in defining, synthesizing, and composing modules to build a model. Dependencies are specified via the library's built-in graph API. In the Python code in Figure 1(a), the first line defines a DNS domain name type (`domain_name`) as a string and limits its size to `5` characters for testing. It also defines DNS-specific types such as a record type (`"RecordType"`) as an `Enum` and a resource record (`"RR"`) as a `struct` containing a record type (`rtyp`), domain name string (`name`), and data value (`rdat`).

The main functionality is split into three modules. The first module `valid_query` takes a DNS query (`"query"`) input (simplified for the example by removing the DNS query type) and ensures that its format is valid. It is defined using a `RegexModule`, which is a pre-defined module type built into EYWA. The second module `ra` (`"record_applies"`) is the main functionality to test. It is defined as a function `FuncModule` that takes two arguments, a DNS query (`query`) and a DNS resource record (`record`), and returns a single boolean output (`result`). The function determines if the record is a match for the query. Finally, the third module `da` (`"dname_applies"`) is a helper module to assist in the construction of `ra`. It defines the matching logic specifically for records that have the `DNAME` record type, which has arguably the most complicated lookup logic among DNS record types.

The user then creates an Eywa `DependencyGraph` to indicate the interrelations among these modules. Here we use `Pipe` to pipe the output of the `RegexModule` module that produces a valid DNS query to the first input of the `ra` `FuncModule` (since it is the first `Pipe` added to `ra`). Next, we add a `CallEdge` to indicate to EYWA that the implementation of `record_applies` can invoke the `dname_applies` function.

Finally, the last two lines of the figure use the EYWA API to synthesize the end-to-end model for the given graph and then generate test cases, each of which is a list of arguments and the expected result, for example:

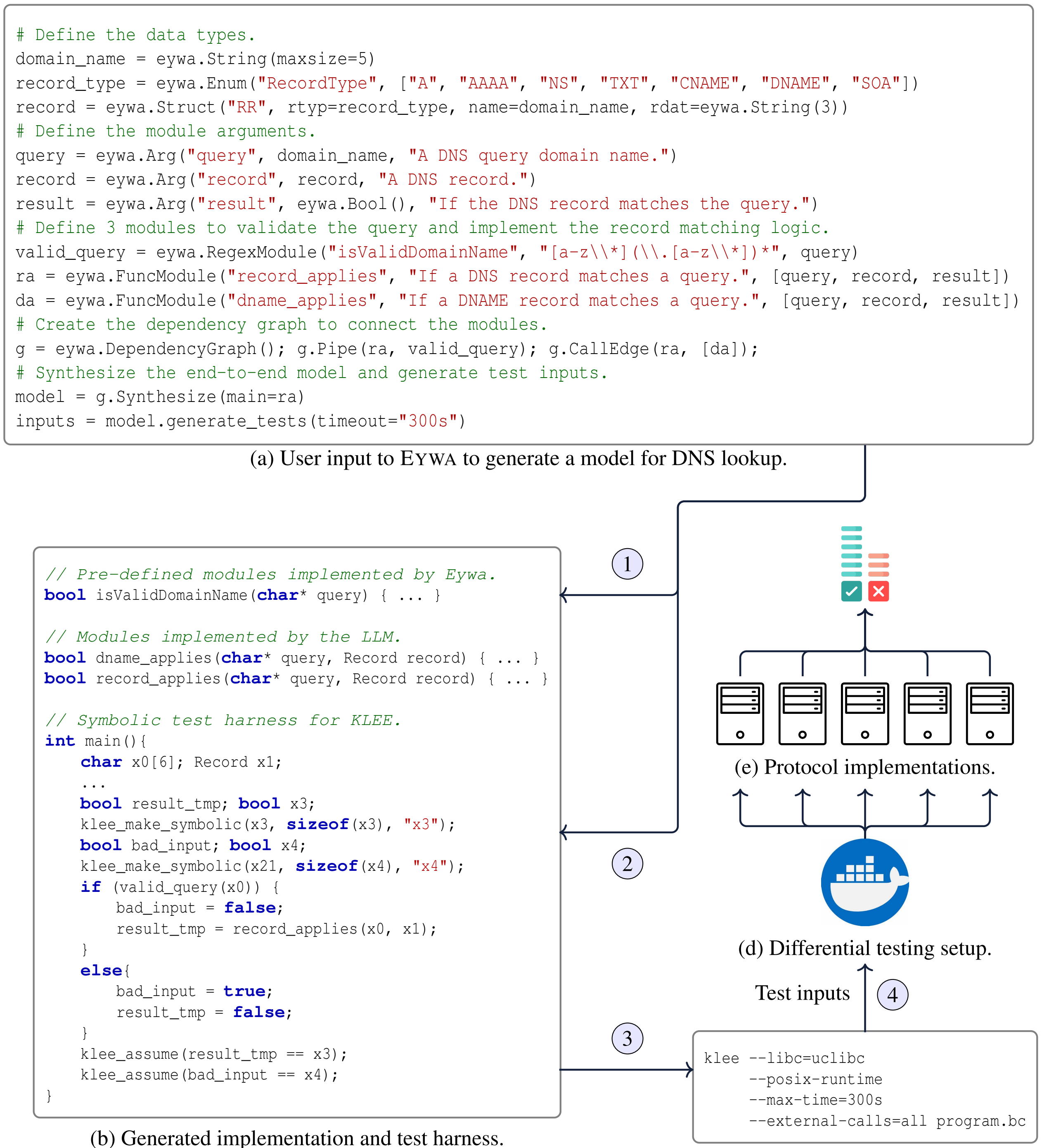
```
['a.*',{'rtyp':'DNAME','name':'*','rdat':'a.a'},
False]
```

How it works. EYWA first prompts the LLM to implement each `FuncModule` in C code (① in Figure 1). It takes into account the input and output argument type definitions and descriptions. Each `RegexModule` is translated into a separate C function that implements the constraint-checking logic using a pre-defined regex-matching implementation in C. EYWA also creates a `main` function by declaring symbolic inputs for the model that the LLM produces ②. It then invokes Klee to symbolically execute the final C code ③ to obtain a suite of test cases. The user can then use these tests for differential testing on actual protocol implementations ④. If, for a given test case, the implementations do not produce the same output, we manually check the relevant documentation to determine whether the discrepancy is expected. Otherwise, it is flagged as a potential bug.

2.2 Dealing with imperfect models

LLM models may be imperfect and generate invalid tests. For our DNS example, the implementation returned by the LLM (slightly simplified for clarity) is shown in Figure 2. The highlighted region of code for the helper function `dname_applies` is actually incorrect – a DNAME record can only apply to a DNS query if it is shorter than the query.

We address these kinds of issues in two ways. First, we ask the LLM to generate k models rather than just a single one, and we generate tests from all of the models using Klee. In our experiments for DNS described later, for example, we set k to 10.



```
# Define the data types.
domain_name = eywa.String(maxsize=5)
record_type = eywa.Enum("RecordType", ["A", "AAAA", "NS", "TXT", "CNAME", "DNAME", "SOA"])
record = eywa.Struct("RR", rtyp=record_type, name=domain_name, rdat=eywa.String(3))
# Define the module arguments.
query = eywa.Arg("query", domain_name, "A DNS query domain name.")
record = eywa.Arg("record", record, "A DNS record.")
result = eywa.Arg("result", eywa.Bool(), "If the DNS record matches the query.")
# Define 3 modules to validate the query and implement the record matching logic.
valid_query = eywa.RegexModule("isValidDomainName", "[a-z\\*](\\.[a-z\\*])*", query)
ra = eywa.FuncModule("record_applies", "If a DNS record matches a query.", [query, record, result])
da = eywa.FuncModule("dname_applies", "If a DNAME record matches a query.", [query, record, result])
# Create the dependency graph to connect the modules.
g = eywa.DependencyGraph(); g.Pipe(ra, valid_query); g.CallEdge(ra, [da]);
# Synthesize the end-to-end model and generate test inputs.
model = g.Synthesize(main=ra)
inputs = model.generate_tests(timeout="300s")
```

(a) User input to EYWA to generate a model for DNS lookup.

```
// Pre-defined modules implemented by Eywa.
bool isValidDomainName(char* query) { ... }

// Modules implemented by the LLM.
bool dname_applies(char* query, Record record) { ... }
bool record_applies(char* query, Record record) { ... }

// Symbolic test harness for KLEE.
int main(){
    char x0[6]; Record x1;
    ...
    bool result_tmp; bool x3;
    klee_make_symbolic(x3, sizeof(x3), "x3");
    bool bad_input; bool x4;
    klee_make_symbolic(x21, sizeof(x4), "x4");
    if (valid_query(x0)) {
        bad_input = false;
        result_tmp = record_applies(x0, x1);
    }
    else{
        bad_input = true;
        result_tmp = false;
    }
    klee_assume(result_tmp == x3);
    klee_assume(bad_input == x4);
}
```

(b) Generated implementation and test harness.

```
klee --libc=uclibc
     --posix-runtime
     --max-time=300s
     --external-calls=all program.bc
```

(c) Klee invocation.

1
2
3
4
Test inputs
(d) Differential testing setup.
(e) Protocol implementations.

Figure 1: Example of how EYWA generates tests. It uses the LLM to implement the module logic. It then compiles a symbolic test harness for Klee and invokes Klee to perform symbolic execution and generate test cases. Tests generated from the EYWA runtime are then executed for every implementation.

Second, as described earlier, we employ differential testing to find bugs, comparing the results of multiple implementations on each test case. This implies that *even if the expected output of an LLM-generated model is incorrect, it can be safely ignored, as we only need them to produce diverse test inputs, while differential testing determines which results from real implementations are inconsistent.* As a result, the model itself can also be simpler than would otherwise be required. In

```
bool dname_applies(char* query, Record record) {
    int l1 = strlen(query);
    int l2 = strlen(record.name);
    // If the DNAME domain name is longer than
    // the domain name, it cannot be a match.
    if (l2 > l1) {return false;}
    // Compare the domain names in reverse order.
    for (int i = 1; i <= l2; i++) {
        if (query[l1 - i] != record.name[l2 - i]) {
            return false;
        }
    }
    // If the DNAME domain name is equal to the
    // domain name, it is a match.
    if (l2 == l1) {
        return true;
    }
                                        Model bug!
    // If the character before the DNAME
    // domain name is a dot, it is a match.
    if (query[l1 - l2 - 1] == '.') {return true;}
    return false;
}
```

Figure 2: LLM model for the example in Figure 1.

our example, we only model the logic that determines whether a resource record applies to a given query, but we need not model what should be returned upon a successful match.

Further, we have observed that some model mistakes can even be helpful. In the case of the logical error above, this will result in Klee generating an extra test for the case where the lengths of the query and DNAME record domain names are equal, which is a useful corner case. While the LLM-generated model produces the wrong result for this case, the test can still be used to find bugs in real implementations by comparing their results to one another (*e.g.*, see §5.2).

2.3 Finding implementation bugs

EYWA is effective at finding implementation bugs. For example, EYWA found a previously unknown bug in the Knot authoritative nameserver implementation [24] with the simple LLM-generated model for DNAME records shown in §2.1. For DNS, we craft valid DNS zone files and queries from the test inputs via a postprocessing step that adds necessary resource records (*e.g.*, SOA and NS) and modifies the test's domain names to have the same suffix. Following this approach, EYWA created the following zone file from a test input (we add the `.test.` suffix in this example):

```
  test.   SOA     ...
  test.   NS      ns1.outside.edu.
*.test.   DNAME   a.a.test.
```

The test DNS query EYWA generated was ⟨a.∗.test., CNAME⟩. In DNS, a CNAME record creates an alias from one domain name to another, while a DNAME record creates an alias for all names under a domain subtree.

The DNAME record above matches the query, so nameserver implementations should return the DNAME record along with a synthesized CNAME record:

```
  *.test.   DNAME   a.a.test.
a.*.test.   CNAME   a.a.a.test.
```

Knot synthesized the CNAME correctly, but it generated a new DNAME record, as follows:

```
a.*.test.   DNAME   a.a.test.
a.*.test.   CNAME   a.a.a.test.
```

When a DNS resolver receives this response from the Knot DNS nameserver, the resolver will incorrectly determine that the DNAME record does not apply to the query and hence produce incorrect behavior.

After discovering the bug, we filed the issue on the Knot Gitlab source code, and the developers responded positively and fixed the issue within a week [62].

3 The EYWA Library and Runtime

3.1 EYWA Architecture

EYWA's architecture consists of several interacting components, illustrated in Figure 3. On the far left of the figure, the user employs the EYWA library to define protocol modules (functions) with arguments and return values, and composes them into a dependency graph, as shown in Figure 1. From these inputs, EYWA invokes its *Prompt Generator* and *Symbolic Compiler* to construct the model.

The *Prompt Generator* creates an LLM prompt for each `FuncModule` declared by the user and queries the LLM to generate a corresponding C function. The *Symbolic Compiler* uses the dependency graph to build the *Symbolic Harness*—the `main` function representing the full model. It declares function inputs symbolic through the Klee API, enabling symbolic execution. The *Symbolic Compiler* also generates C functions for predefined modules such as `RegexModule`. The complete model combines outputs from both the *Prompt Generator* and the *Symbolic Compiler*. This synthesis process is repeated k times to produce k distinct models. Finally, the *Test Generator* runs Klee on these models, extracts the resulting test cases, and translates them back into Python data structures.

3.2 EYWA Library

The Eywa library facilitates building models as typed functions that accept a set of arguments and return a result. This

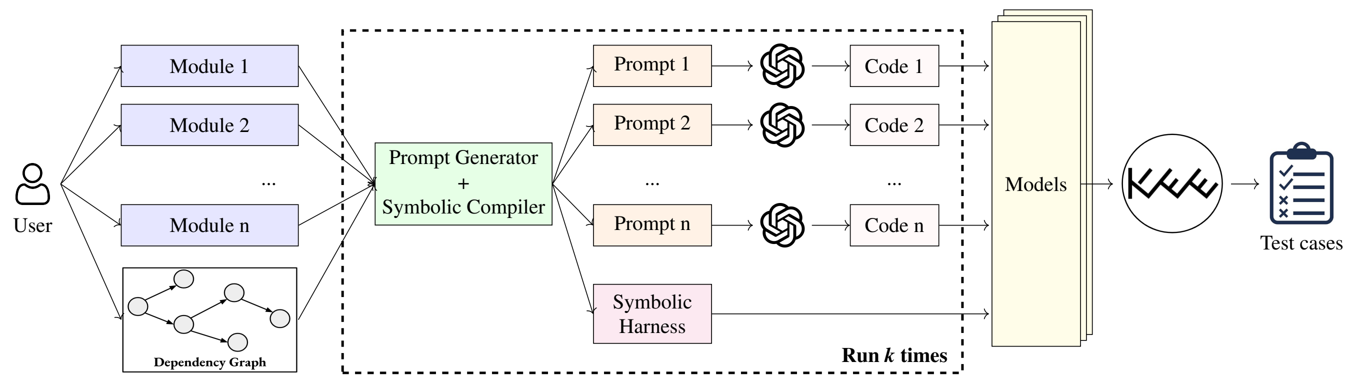


Figure 3: Architecture of EYWA. It takes modules and dependencies and produces *k* different protocol models.

Example Feature	Description
`eywa.Bool()`	Boolean value
`eywa.Char()`	Character value
`eywa.String(maxsize=5)`	String of size
`eywa.Int(bits=5)`	5-bit unsigned int
`eywa.Enum("name",["A", "NS"])`	Enum value
`eywa.Array(Bool(), 3)`	Array of 3 bools
`eywa.Struct("name",dst=Int())`	Struct with 1 field
`eywa.Alias("result",Bool())`	Type alias
`eywa.Arg("name",Int(),"desc")`	Function arg
`eywa.FuncModule("name","desc",args)`	Function

Figure 4: Summary of EYWA types and abstractions.

design is general and is applicable to many kinds of protocols. For instance, to model the logic of a stateful network protocol like SMTP, one could create a function that takes (1) an SMTP message representing a received email or command, and (2) the current state of the SMTP protocol for the host handling the message, and returns (3) the appropriate SMTP response or action to take and (4) the updated state.

A summary of the main modeling abstractions is shown in Figure 4. The library supports function arguments with standard types such as booleans, characters, strings, fixed bit-width integers, enums, arrays, and structs. It also provides type aliases, which let users assign custom names to types (e.g., to help the LLM understand a type's meaning). For types with potentially unbounded size (e.g., `eywa.String()`), users must provide a size bound to limit the number of test cases generated by EYWA. Functions are created by specifying a name, a natural language description of their purpose, and a list of arguments. Each argument includes a name, description, and associated type.

3.3 EYWA Support for Modularity

As discussed in §2.1, there are two kinds of modules available in EYWA. A `FuncModule` is initialized by providing a natural language description of its content. The code is then written by an LLM through a prompt generated by EYWA. EYWA also includes a built-in `RegexModule` since regex filtering is a common need for protocols, but users can provide their own modules as well for specialized functionality for which they want full control.

Dependency graphs determine how modules are connected to form a full model. EYWA's dependency graphs support two kinds of edges. A `Pipe` is used for *sequential composition*, where the output of one module is the input of another. A `CallEdge` indicates that the implementation of one module may invoke another one and thereby supports a form of *decomposition*: a complex module can be created through multiple LLM prompts. If a module m_1 is declared to call another module m_2, then the C function prototype for m_2 is provided when the LLM is asked to synthesize m_1. A separate LLM invocation will produce the implementation of m_2. See Appendices A and C for further discussion.

3.4 EYWA Symbolic Compiler

The *Symbolic Compiler* generates the *Symbolic Harness*, which is the `main` function used for symbolic execution. We use the `klee_make_symbolic` command to build symbolic inputs for every base type (**`bool`**, **`char`**, **`int`**, **`enum`** *etc.*) and then construct values of more complex types from these base types. For example, a struct will be initialized with symbolic values for each of its fields.

3.5 EYWA Prompt Generator

Given a set of modules and the dependency graph for a model, EYWA's *Prompt Generator* creates LLM prompts, then LLM implements each module, and then they are combined into one model. For each module, EYWA builds two prompts: a user prompt that frames the implementation task as a completion problem, and a system prompt (details given in Appendix D) to guide the behavior of the LLM. Consider again the example model defined using the EYWA library in Figure 1.

```c
#include <stdint.h>
...
typedef enum {
    A, AAAA, NS, TXT, CNAME, DNAME, SOA
} RecordType;

typedef struct {
    RecordType record_type;
    String name;
    String rdata;
} Record;

// If a DNAME record matches a query.
// ...
bool dname_applies(char* query, Record record);

// If a DNS record matches a query.
// Parameters:
//     query: A DNS query domain name.
//     dname: A DNS record.
// Return Value:
//     If the DNS record matches the query.
bool record_applies(char* query, Record record) {
```

Figure 5: LLM prompt for the example in Figure 1.

User Prompt. To create the user prompt, EYWA translates each of the user-defined types into C data types, creates the C function signatures using these types, and adds a function documentation string from the user descriptions. We show the resulting prompt that is generated for `record_applies` of Figure 1 in Figure 5. From this prompt, the LLM will naturally predict (complete) the rest of the function.

System Prompt. EYWA also generates a system prompt (for GPT4). It provides additional guidance to help ensure the LLM generates valid code, including the following:

- Describing the task to implement the C function provided by the user prompt.
- Requiring that the LLM only add import statements and not remove existing imports.
- Requiring that the LLM not delete or modify any of the user-defined type definitions.
- Requiring that the LLM not add its own `main()` function and instead just implement the function provided.
- Requiring that the LLM not use certain C functions that are not amenable to symbolic execution.
- Giving a valid input-output example. For instance, showing a blank template of a function that adds 1 to an integer as the input, and the fully implemented function as the output.

3.6 EYWA Test Generator

The final component to EYWA is the *Test Generator*, which is responsible for running Klee and translating the results back into Python values for the user. The *Test Generator* assembles the outputs from the *Prompt Generator* and *Symbolic Compiler* into a single program and then runs the `clang` compiler to build LLVM bytecode. It then executes Klee on the result with the user-provided timeout (if any). For isolation, both of these tasks are executed in a separate docker container, and any errors, including compile errors, are reported to the user as feedback. Users can use this feedback to update their model to assist the LLM or provide additional context.

The result of running Klee is a set of test inputs that assign each base symbolic C variable to a value. The *Test Generator* then serializes these values back to EYWA, which walks over the base values and reconstructs any richer types (*e.g.*, struct, array) from these values. EYWA also captures the result value of each test case and includes that information.

4 Implementation

The EYWA library is implemented in around 2K lines of Python code with an additional 200 lines of C code. For its language model, it currently uses GPT 4 hosted on the Azure OpenAI service [95]. When generating models, EYWA allows the user to specify the LLM temperature value τ between 0.0 and 1.0 as well as the number of C implementations k to generate for each model. To improve test coverage, we generate k implementations and then aggregate the resulting test cases. For our experiments, we use $k = 10$ and temperature $\tau = 0.6$ (see Appendix B). After assembling each model from the LLM and symbolic test harness, we attempt to compile it and invoke Klee in a Docker container, and skip the implementation in the event of a compilation error.

5 Evaluation

5.1 Methodology and Setup

Alongside DNS, we also experimented with BGP and SMTP, each with multiple implementations that we use for differential testing, summarized in Table 1. For each protocol, we build several models with EYWA in order to test various aspects of the protocol. Table 2 presents all the models we created with EYWA for the purpose of testing. The table shows the number of lines of Python code that were needed to define each model in EYWA (similar to Figure 1). The table also gives the range (both the minimum and maximum) of lines of code in C that EYWA generates across the k model implementations. Finally, we show the total number of unique test cases for each model returned by EYWA after running Klee on the generated C code. This includes the union of all unique test cases across the k different implementations.

5.1.1 Overview of models

We constructed eight DNS models using EYWA (shown in Table 2) to test different aspects of authoritative nameservers. To evaluate when EYWA performs well and when it does not, we designed a diverse set of models: (1) simple models that

Protocol	Tested Implementations
DNS	BIND [20], COREDNS [14], GDNSD [5], NSD [86] HICKORY [34], KNOT [24], POWERDNS [19] TECHNITIUM [133], YADIFA [30], TWISTED [87]
BGP	FRR [17], GOBGP [18], BATFISH [32]
SMTP	AIOSMTPD [2], SMTPD [26], OPENSMTPD [1]

Table 1: Protocol implementations tested by EYWA.

Protocol	Model	LOC (Python)	LOC (C)	Tests
DNS	CNAME	21	222 / 246	435
DNS	DNAME	23	209 / 230	269
DNS	WILDCARD	23	210 / 238	470
DNS	IPV4	21	209 / 229	515
DNS	FULLLOOKUP	26	487 / 510	12,281
DNS	RCODE	26	487 / 510	26,617
DNS	AUTH	26	477 / 504	31,411
DNS	LOOP	26	474 / 489	31,453
BGP	CONFED	22	189 / 202	957
BGP	RR	16	59 / 76	36
BGP	RMAP-PL	48	150 / 162	400
BGP	RR-RMAP	48	341 / 366	7147
SMTP	SERVER	26	245 / 252	80

Table 2: Models, lines of code (min / max), number of tests generated for DNS, BGP and SMTP with EYWA.

test record matching for individual DNS types, (2) end-to-end models of the full DNS lookup process, and (3) specialized models that target unusual or corner-case behaviors.

The first four models (CNAME, DNAME, WILDCARD, IPV4) determine whether a DNS query matches a single record type in a zone. The FULLLOOKUP model implements the complete authoritative lookup procedure for a query and zone file. Two variants, AUTHORITATIVE and RCODE, return only part of the response—specifically, the authoritative flag and the return code—rather than the full DNS response. Finally, the LOOP model targets a corner case by counting how many times a DNS query is rewritten for a given zone file. This forces EYWA to generate inputs that trigger repeated or even infinite lookups. Because both LOOP and FULLLOOKUP reach the Klee timeout, the simpler structure of LOOP allows it to generate more test cases.

For the Border Gateway Protocol (BGP), we modeled several features. These include confederations (CONFED), route-reflector behavior with clients, non-clients, and external ASes (RR), the use of route-maps and prefix lists in processing route advertisements (RMAP-PL), and a combined model of route reflectors with route-maps (RR-RMAP).

Finally, for the Simple Mail Transfer Protocol (SMTP), we created an SMTP server model that responds to requests based on server state, including returning appropriate error messages for invalid inputs. It is implemented as a function

```c
#include <stdint.h>
#include <stdbool.h>
...
typedef enum { INITIAL, HELO_SENT,
               EHLO_SENT, MAIL_FROM_RECEIVED,
               RCPT_TO_RECEIVED, DATA_RECEIVED,
               QUITTED } State;
// A function that takes the current state of
// the SMTP server, the input string, updates
// the state and returns the output response.
//
// Parameters:
//     state: Current state of the SMTP server
//     input: Input string
// Return Value:
//     Output string
char* smtp_server_resp(State state, char* input)
{...
```

Figure 6: First prompt for the SMTP server model.

that takes two parameters, *state* and *input*, and returns the corresponding server response. Figure 6 and Appendix E show the EYWA-generated prompt and the resulting code from the LLM, respectively.

5.1.2 Overview of testing setup

Seven of the ten DNS implementations we evaluated had also been tested by SCALE [49]. For these seven, we examined both historical versions (before bug fixes) and current versions. This approach enabled us to measure overlap between bugs found by EYWA and SCALE, while also identifying previously undiscovered issues. To test each implementation, we applied differential testing as described in Figure 1. We built Docker [94] containers for each implementation and version, with each container serving a single zone file as an authoritative zone. Using the `dnspython` [15] library, we constructed DNS queries, sent them to the containers, and compared the responses to detect discrepancies (*e.g.*, in the answer, authoritative section, flags, additional section, or return code).

Because multiple tests can expose the same underlying bug, we aimed to isolate unique root causes. For each implementation whose response deviated from the majority group, we classified the cause of the discrepancy as a tuple abstracting the differing components. For example, if an implementation returned `NXDOMAIN` instead of `NOERROR`, the tuple would be (COREDNS, `rcode`, `NXDOMAIN`, `NOERROR`). Each unique tuple was then manually inspected, and any unexpected results were reported to the maintainers.

For the BGP implementations, we set up a Docker network containing three nodes R1, R2 and R3 connected in series. We ran the BGP implementation under test on R2 and R3 but ran an ExaBGP [16] instance on R1 so that we could inject routes into R2. For BGP confederations specifically, we built a lightweight reference implementation to enable differential testing against FRR, as confederation logic is not

~~~
User_Prompt
Create a python dictionary that maps the state
transitions: (state,input) --> state
as per the following C code snippet:
smtp_server_response(State state, char* input) {
    <SMTP server code generated by LLM>
}
Output_Format
A python dictionary like
{(state1, input1): state2,
(state3, input2): state4, ...}
GPT_Response
Here is the Python dictionary that maps the
state transitions:

```python
state_transitions = {
    (INITIAL, "HELO"): HELO_SENT,
    (INITIAL, "EHLO"): EHLO_SENT,
    (HELO_SENT, "MAIL FROM:"): MAIL_FROM_RECEIVED,
    (EHLO_SENT, "MAIL FROM:"): MAIL_FROM_RECEIVED,
    (MAIL_FROM_RECEIVED, "RCPT TO:"):
    RCPT_TO_RECEIVED,
    (RCPT_TO_RECEIVED, "DATA"): DATA_RECEIVED,
    (HELO_SENT, "QUIT"): QUITTED,
    (EHLO_SENT, "QUIT"): QUITTED,
    (MAIL_FROM_RECEIVED, "QUIT"): QUITTED,
    (RCPT_TO_RECEIVED, "QUIT"): QUITTED,
    (DATA_RECEIVED, "QUIT"): QUITTED,
}
```
~~~

Figure 7: Second LLM prompt and response to generate the SMTP state graph as a Python dictionary.

fully supported in Batfish or GoBGP. We then checked the BGP routing tables on both R2 and R3. For Batfish, which is a BGP simulator rather than an implementation, we set up a simulated network of three nodes. We wrote test translators for all three implementations and then performed differential testing on the three implementations. Error triaging is done in the same way as for the DNS experiments.

Finally, for SMTP, each test case is a (state, input) pair, so we must first drive the SMTP implementation into the desired state before running the test input. To do this, EYWA uses *another LLM call* to convert the generated SMTP server code into a state-transition graph (see Figure 7). For each test case, we run a breadth-first search (BFS) on that graph to find an input sequence that transitions the server from its initial state to the target state; we then prepend that sequence to the test's input. Server implementations listen on 127.0.0.1:8025, and a separate process on the same machine generates the input requests. After each test run, the server is reset to its initial state. We also demonstrate that this methodology can model more complex stateful protocols (e.g., TCP; see Appendix F); full TCP testing, however, is left to future work.

5.1.3 Research Questions

Our evaluation is guided by three questions:

- **RQ1:** Can EYWA generate tests quickly?
- **RQ2:** Are the models produced by EYWA high quality?
- **RQ3:** Does EYWA find real bugs in implementations?

5.2 Results

We answer the above questions through the results of our experiments with EYWA on DNS, BGP, and SMTP implementations, including quantitative and qualitative evidence of its practical utility and benefit over prior approaches.

RQ1 – Test generation speed. The running time for EYWA is dominated by the time to test the implementations. Each LLM query took under 20 seconds to complete and could be reduced by using faster LLMs. For test generation using Klee, the time taken varies based on the complexity of the case. For the initial four models of DNS and for the SMTP server model, Klee completes the process in approximately 5-10 seconds. However, for the other DNS models, which are more complex, Klee consistently hits the 5-minute timeout that we use. All of our BGP models are bounded in size, so Klee always terminated on them within 5-10 seconds.

RQ2 - Model quality. We manually inspected EYWA's generated implementations to assess quality. For each model listed in Table 2, EYWA produced implementations that captured the intended protocol semantics. For example, in the DNS FULL-LOOKUP task — the most challenging DNS task because it requires a full nameserver-lookup model — the LLM correctly implemented the behavior of each record type, including `DNAME` (suffix rewrite), `CNAME` (exact rewrite), and wildcard records (partial match). The LLM did not fully implement DNS's "closest encloser" semantics (implementing it precisely would require a complex data structure); instead it typically used a sequential, first-match search through zone records. Though technically incorrect, this approximation, combined with symbolic execution, produced effective tests.

For BGP, EYWA's models were similarly accurate overall. With careful prompt engineering we could make the LLM cover missed corner cases; however, for BGP confederations, the LLM sometimes failed to update AS paths correctly despite multiple prompt variations. Because we use differential testing, the test cases generated by these imperfect models nonetheless proved useful. For the SMTP server, EYWA consistently produced a correct model.

In several cases, the LLM initially produced flawed C models due to misunderstandings about the side effects of the C function `strtok`. We fixed this by updating the system prompt to instruct the LLM to avoid `strtok`. Across all experiments, the LLM produced only a single C model that failed to compile; such models can be discarded. As LLMs improve, we expect issues we observed (for example, mis-

Protocol	**Implementation**	**Description**	**New Bug?**	**Acked?**
DNS	BIND	Sibling glue record not returned	○	●
DNS	BIND	Inconsistent loop unrolling	●	●
DNS	COREDNS	Wildcard CNAME and DNAME loop [57]	○	●
DNS	COREDNS	Sibling glue record not returned [58]	○	●
DNS	COREDNS	Returns SERVFAIL yet gives an answer [47]	●	○
DNS	COREDNS	Returns a non-existent out-of-zone record [46]	●	○
DNS	COREDNS	Wrong RCODE for synthesized record [61]	○	●
DNS	COREDNS	Wrong RCODE for empty non-terminal wildcard [60]	●	●
DNS	GDNSD	Sibling glue record not returned [50]	○	●
DNS	HICKORY	Wildcard CNAME and DNAME loop [51]	○	●
DNS	HICKORY	Incorrect handling of out-of-zone record [59]	●	●
DNS	HICKORY	Wildcard match only one label [54]	○	●
DNS	HICKORY	Wrong RCODE for empty non-terminal wildcard [53]	●	●
DNS	HICKORY	Wrong RCODE when '*' is in RDATA [52]	●	●
DNS	HICKORY	Glue records returned with authoritative flag [56]	○	●
DNS	HICKORY	Authoritative flag set for zone cut NS records [55]	○	●
DNS	KNOT	DNAME record name replaced by query [62]	●	●
DNS	KNOT	Wildcard DNAME leads to wrong answer [64]	●	●
DNS	KNOT	Error in DNAME-DNAME loop Knot test [65]	○	●
DNS	KNOT	DNAME not applied recursively [66]	○	●
DNS	KNOT	Incorrect record synthesis when '*' is in query [63]	○	●
DNS	NSD	DNAME not applied recursively [72]	○	●
DNS	NSD	Wrong RCODE when '*' is in RDATA [71]	○	●
DNS	POWERDNS	Sibling glue record not returned due to wildcard [70]	●	●
DNS	TECHNITIUM	Sibling glue record not returned [81]	○	●
DNS	TECHNITIUM	Synthesized wildcard instead of applying DNAME [80]	●	●
DNS	TECHNITIUM	Invalid wildcard match [78]	○	●
DNS	TECHNITIUM	Nested wildcards not handled correctly [79]	●	●
DNS	TECHNITIUM	Duplicate records in answer section [76]	○	●
DNS	TECHNITIUM	Wrong RCODE for empty nonterminal wildcard [77]	●	●
DNS	TWISTED	Empty answer section with wildcard records [68]	○	●
DNS	TWISTED	Missing authority flag and empty authority section [67]	○	●
DNS	TWISTED	Wrong RCODE for empty nonterminal wildcard [69]	●	●
DNS	TWISTED	Wrong RCODE when '*' is in RDATA [68]	○	●
DNS	YADIFA	CNAME chains are not followed [75]	○	●
DNS	YADIFA	Missing record for CNAME loop [73]	●	○
DNS	YADIFA	Wrong RCODE for CNAME target [74]	○	●
BGP	FRR	Prefix list matches mask greater than or equals [115]	○	●
BGP	FRR	Confederation sub AS equal to peer AS [116]	●	○
BGP	FRR	Replace-AS not working with confederations [100]	●	○
BGP	GOBGP	Prefix set match with zero masklength but nonzero range [96]	○	●
BGP	GOBGP	Confederation sub AS equal to peer AS [99]	●	○
BGP	BATFISH	Local preference not reset for EBGP neighbor [98]	●	●
BGP	BATFISH	Confederation sub AS same as peer AS [97]	●	●
SMTP	AIOSMTPD	Server accepting request without appropriate headers [117, 118]	●	●

Table 3: The bugs found in DNS, BGP, and SMTP implementations tested by EYWA, with status indicating whether they were acknowledged by developers. The BIND issues are no longer available on their GitLab.

use of `strtok`) to diminish, making EYWA's models more accurate and reducing the need for prompt engineering.

RQ3 – Bug-finding effectiveness. To evaluate the quality of tests generated by EYWA, we compared its results against prior work that relied on manually written models: SCALE [49] for DNS and MESSI [119] for BGP. Unlike SCALE and MESSI, which used carefully constructed models derived from RFCs, EYWA leverages LLMs and requires orders of magnitude less modeling effort. However, we initially expected the hand-crafted models to be more effective at uncovering bugs. Table 3 summarizes the results of our differential testing. The table reports each bug found in the tested protocol implementations, along with its description, type, and effect. It records whether the bug was detected by SCALE or MESSI, or whether it would have been detected had SCALE tests been applied to newly added implementations. It also contains information on which bugs were confirmed by the developers. § 2.3 discussed an example DNS bug found by EYWA. Here we show two more interesting bugs found by our tool, one in BGP and another in SMTP:

Bug #1. BGP confederations allow administrators to divide a large AS into smaller "sub-ASes" while still appearing as a single AS to external peers. For the CONFED model, EYWA generated an interesting test case where a router R within a confederation had the same sub-AS number as its neighbor N's AS number, where N is outside the confederation. The bug arose when R tried to establish an iBGP connection with N because it thought that N was in the same sub-AS as itself. On the other hand, N thought R was outside its AS, so N tried to initiate an eBGP connection. As a result, a successful BGP peering could not be established. Klee was probably able to find such a test case because it tends to assign similar values to symbolic variables of the same type unless strictly constrained. This bug is now fixed by the Batfish developers [97].

Bug #2. We found a behavioral discrepancy in OpenSMTPD when testing the `DATA_RECEIVED` state. After issuing the sequence `HELO`, `MAIL FROM`, `RCPT TO`, `DATA`, and terminating the message with `"."`, OpenSMTPD responded with `550 5.7.1 Delivery not authorized, message refused: Message is not RFC 2822 compliant`, whereas the same test executed against aiosmtpd returned `250 OK`. In SMTP, `HELO` establishes client identity, `MAIL FROM` specifies the envelope sender, `RCPT TO` specifies the recipient, `DATA` transitions the session into the message-content phase, and the terminating `"."` signals the end of the message body (i.e. the `DATA_RECEIVED` state). Investigation revealed that our test message contained only the SMTP envelope commands but lacked mandatory RFC 2822 headers (e.g., `Date` and `From`) in the message body. OpenSMTPD developers clarified that their implementation correctly enforces RFC 2822 Section 3.6, which requires specific header fields, while aiosmtpd does not [118].

Overall, for DNS, EYWA identified **37** bugs across ten implementations, of which **15** were previously undiscovered by SCALE. After accounting for duplicates across implementations, this corresponds to **27** unique bugs, of which **12** were not detected by SCALE. In comparison, SCALE revealed **22** unique bugs, of which **7** were missed by EYWA. *Thus,* EYWA *discovered more bugs than SCALE, despite requiring minimal modeling effort.* For BGP, EYWA replicated two bugs in prefix lists and route filtering previously reported by MESSI. More importantly, it modeled two complex BGP features—confederation and route reflection—that were not included in MESSI's manually written model. Through this extension, EYWA uncovered **5** unique bugs, including **3** that were new ones. Finally, in our SMTP experiments, EYWA demonstrated its ability to handle stateful protocols. Despite the simplicity of the server model, one generated test case revealed a previously unknown bug: in the `RCPT_TO_RECEIVED` state, providing `DATA` as input to the server triggered an error.

Additional Insights:. We manually examined the seven bugs that SCALE identified, but EYWA did not. In four of the seven cases, the corresponding test purposely used an invalid zone file to test how implementations handle them. The EYWA-generated DNS models ensure validity, but we could also use EYWA to test invalid zone files. In one case, the bug found by SCALE depends on a subtlety in the logic of handling `CNAME` records that the LLM-generated model did not include. Finally, in a few cases, we identified a bug that went beyond the bounds that we set for the EYWA-generated tests, for example, involving multiple records in a zone file.

Discussion:. For EYWA to be effective, the LLM must have a strong understanding of the protocol being modeled. For the protocols we tested (i.e., DNS, BGP, and SMTP), LLMs already encode substantial knowledge from publicly available documentation, and often only minimal function descriptions were needed; but for newer protocols user may need to provide additional documentation. Because EYWA relies on differential testing, it assumes some behavioral diversity across implementations: in principle, if all implementations agree yet jointly deviate from the specification, EYWA would not flag the issue (a false negative). In practice, however, as the number of independently developed implementations increases, the likelihood that all of them exhibit the same incorrect behavior decreases substantially. Even when multiple implementations are unavailable, independently generated models or agents could provide additional diversity.

6 Related Work

Automated testing. Automated testing is often categorized by visibility into source code. Black-box testers [13, 31, 35, 104, 111, 128] generate random tests without source access. Grey-box testers [7, 8, 12, 33, 36, 108, 124] use lightweight instrumentation and coverage feedback. White-box testers [10, 37–39] use symbolic methods to solve path constraints. Black and grey-box approaches may get stuck on complex conditions, resulting in low coverage, while white-box approaches suffer from path explosion and require non-trivial source code modification [10, 38, 39, 105].

Another approach is model-based or specification-based testing [27, 49, 83, 90, 91, 122], which applies to black-box implementations by using a reference model or specification to generate tests. Compared to black and grey-box fuzzing, MBT offers stronger semantic guidance and more principled coverage. But it still requires users to manually craft a model, which is a non-trivial undertaking. In contrast, EYWA retains the systematic exploration benefits of traditional MBT, yet reduces effort by leveraging an LLM's prior knowledge for model construction. Our work is similar in spirit to XM [25], which emphasizes lightweight, disposable models for rapid feedback. However, EYWA facilitates automation by shifting the burden of building the models from the human to an LLM.

Protocol testing. Some testing work has directly targeted protocol implementations due to their importance and complexity [3, 6, 35, 40, 49, 84, 102, 105, 107, 119]. Our approach builds on MBT approaches for DNS [49] and BGP [119], which combine symbolic execution on the model with differential testing of implementations. However, EYWA offloads most of the modeling task to LLMs through an API for modular and declarative model construction, significantly reducing the human burden and making EYWA useful for testing many protocols, while each prior tool targets a single protocol.

Other approaches attempt to marry software engineering testing techniques, such as grey-box fuzzing, with a learned protocol *state* (*e.g.*, the protocol state machine). For instance, Pulsar [35] and AFLNet [107] combine black-box and grey-box testing, respectively, with protocol state machine learning from execution traces. While they facilitate improved exploration over vanilla fuzzing techniques, they still rely on mutation-based input generation and "reverse engineer" protocol semantics from observed behavior. In contrast, EYWA uses LLMs to directly encode a protocol's state and logic.

N-version programming. N-version programming (NVP) [11, 85, 110] is a fault-tolerance technique in which multiple independently developed implementations of the same specification are executed in parallel and their outputs compared, often via majority voting. In contrast, EYWA focuses on *bug discovery*, leveraging LLMs to build models that are used to generate tests for existing implementations.

LLM-based testing. Recent breakthroughs with LLMs have led researchers to reconsider the possibility of automatically testing software [41, 82, 112, 114, 121, 123, 125, 129, 130, 132]. Most of these works directly ask LLMs to write unit tests for software or fuzz programs (*i.e.*, mutate strings). In the networking domain, LLMs have been used to derive test cases from structured information extracted from RFCs [126]. These are then converted to executable test code through multiple iterations using various retrieved examples and LLM/user feedback. EYWA instead uses LLMs to write simple implementations with well-typed inputs. It then uses off-the-shelf symbolic execution tools to generate exhaustive tests. Recently, ChatAFL [93] enhanced the stateful protocol fuzzer AFLNet [107] using LLMs. Like EYWA, ChatAFL uses LLM knowledge to improve testing, but it directly modifies the protocol messages used in tests. These two approaches are quite distinct and likely have different strengths.

NLP for networks. Some early work shared the insight that RFCs and other natural language sources could provide useful information for testing network protocols [131] or generating network configurations [43]. More recently [113], the authors used LLMs to extract protocol specifications from RFCs in the form of protocol automata. EYWA too extracts a specification, however it does so with human guidance and in the form of a C program that may be combined with symbolic execution.

7 Conclusion

Despite widespread enthusiasm, a major challenge in deploying large language models (LLMs) in safety-critical applications is their tendency to produce errors. For example, when LLMs generate code, we must test [88] or formally verify [101, 120] the outputs and iterate as necessary. Conversely, we argue in this paper that when LLMs are used to *test* software, errors in both the models and the tests they generate can be mitigated using differential testing.

Specifically, we introduce *model-based testing with LLMs*, a new approach for automatic, black-box protocol testing. We use LLMs for two complementary tasks: (1) constructing protocol models and (2) generating state graphs that identify efficient sequences to drive protocols into target states. For lesser-known protocols, the method can be augmented with retrieval-augmented generation (RAG) over available software documentation. We defer empirical testing and broader evaluation to future work. EYWA substantially reduces the burden of manually constructing protocol models, works with black-box implementations without source-code changes, and produces tests that uncover deep functional correctness bugs.

Our testing framework, EYWA, discovered 45 bugs across protocol implementations, including 21 that prior manually constructed model-based testing (MBT) tools had missed. We rapidly built models for BGP features such as confederations and route reflectors, which earlier work (MESSI [119]) had not tested, and we also developed an SMTP server model. The SMTP experiments further showed that LLMs can drive protocols to specified states for testing, suggesting substantial untapped potential beyond our initial exploration.

8 Acknowledgments

We thank our shepherd, Sanjay Rao, and the NSDI reviewers for their helpful feedback. We also thank the protocol implementation developers for their responses to our bug reports. The work of the first four authors was partially supported by the National Science Foundation under award CNS-2402958.

References

[1] Opensmtpd. https://www.opensmtpd.org/, 2024.

[2] aiosmtpd community. aiosmtpd - an asyncio based smtp server. https://aiosmtpd.aio-libs.org/en/latest/, 2024.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[4] Scott Berinato. All systems down. https://www.computerworld.com/article/2581420/all-systems-down.html, 2023. Accessed: 2023-9-29.

[5] Brandon L Black and Community. Gdnsd. https://gdnsd.org/, 2023.
Eywa commit: https://github.com/gdnsd/gdnsd/tree/877e15cf55593fa618d2009027e928d5f52da775.

[6] Gregor V Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, 1994.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[9] Chris Brook. Cisco fixes dos, authentication bypass vulnerabilities, ospf bug. https://threatpost.com/cisco-fixes-dos-authentication-bypass-vulnerabilities-ospf-bug/127185/, 2023. Accessed: 2023-9-29.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[11] Liming Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, pages 113–, 1995.

[12] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[13] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[14] CoreDNS community. Coredns. https://coredns.io/, 2023.
Eywa commit: https://github.com/coredns/coredns/tree/45923b6e12a2eabaf55d7380e6df4e7354a1207
SCALE commit: https://github.com/coredns/coredns/tree/6edc8fe7f6c2f57844c8ee7f7f5deef71085ebe8.

[15] Dnspython Community. Dnspython. https://dnspython.readthedocs.io/en/latest/index.html, 2023.

[16] ExaBGP community. Exabgp. https://github.com/Exa-Networks/exabgp, 2024.

[17] FRR community. The frrouting protocol suite. https://frrouting.org/, 2024.
Version: https://github.com/FRRouting/frr/releases/tag/frr-10.1.2.

[18] GoBGP community. Gobgp. https://github.com/osrg/gobgp, 2024.

[19] PowerDNS Community. Powerdns. https://www.powerdns.com/, 2023.
Eywa commit: https://github.com/PowerDNS/pdns/tree/8314f12e92a8b75e33438bc7c16c6430028fbef9
SCALE commit: https://github.com/PowerDNS/pdns/tree/a03aaad7554483ee6efe72a81eda00a9d1a94fe5.

[20] Internet Systems Consortium. Bind 9. https://www.isc.org/bind/, 2023.
Eywa commit: https://gitlab.isc.org/isc-projects/bind9/-/tree/85ee12f60edb6b79535f6f226250ac471d68fbab
SCALE commit: https://gitlab.isc.org/isc-projects/bind9/-/tree/dbcf683c1a57f49876e329fca183cb39d20ca3a4.

[21] curl. Ftp server response buffer overflow. https://curl.se/docs/CVE-2000-0973.html, 2023. Accessed: 2023-9-29.

[22] curl. Ftp shutdown response buffer overflow. https://curl.se/docs/CVE-2018-1000300.html, 2023. Accessed: 2023-9-29.

[23] cyberstanc. Pinging our way to remote code execution: The new icmp vulnerability you need to know about! https://cyberstanc.com/blog/pinging-our-way-to-remote-code-execution-the-new-icmp-vulnerability-you-need-to-know-about/, 2023. Accessed: 2023-9-29.

[24] CZ.NIC. Knot.
https://www.knot-dns.cz/, 2023.
Eywa commit: https://gitlab.nic.cz/knot/knot-dns/-/tree/c08e5738b6eed43b052a127d56db6451106386fa
SCALE commit: https://gitlab.nic.cz/knot/knot-dns/-/tree/89aaeb729a0856fefaed111c114ebb8a5a3f4ed2.

[25] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. Extreme modelling in practice. *Proc. VLDB Endow.*, 13(9):1346–1358, May 2020.

[26] Python developer community. Smtpd python library. https://docs.python.org/3.10/library/smtpd.html, 2024.

[27] Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, John Kastner, Anwar Mamat, Matt McCutchen, Neha Rungta, Bhakti Shah, Emina Torlak, and Andrew Wells. How we built cedar: A verification-guided approach. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 351–357, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Jim Duffy. Bgp bug bites juniper software. https://www.networkworld.com/article/2289950/bgp-bug-bites-juniper-software.html, 2023. Accessed: 2023-9-29.

[29] Tushar Subhra Dutta. Bgp error handling flaw leads to prolonged network outage. https://cybersecuritynews.com/bgp-error-handling-flaw/, 2023. Accessed: 2023-9-29.

[30] EURid.eu. Yadifa.
https://www.yadifa.eu/, 2023.
Eywa commit: https://github.com/yadifa/yadifa/tree/9bb6facead9e7ba222962b2980f85fa6ba02e465
SCALE commit: https://github.com/yadifa/yadifa/tree/dc5bed2fb8ec204af9b65eeb91934c2c85098cbb.

[31] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 337–350, 2021.

[32] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 469–483, USA, 2015. USENIX Association.

[33] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[34] Benjamin Fry and Community. Hickory-dns.
https://github.com/hickory-dns/hickory-dns, 2023.
Eywa commit: https://github.com/hickory-dns/hickory-dns/tree/65c5327ef6b8dbda92654837b8b5cb31fa0000ad
SCALE commit: https://github.com/bluejekyll/trust-dns/tree/7d9b186121fb5cb331cf2ec6baa47846b83de8fc.

[35] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*, pages 330–347. Springer, 2015.

[36] github. google/afl. https://github.com/google/AFL, 2023. Accessed: 2023-9-29.

[37] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[38] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[39] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[40] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns*, 10(8):239, 2010.

[41] Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*, 2023.

[42] ipSpace. Oversized as paths: Cisco ios bug details. https://blog.ipspace.net/2009/02/oversized-as-paths-cisco-ios-bug.html, 2023. Accessed: 2023-9-29.

[43] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. Hey, lumi! using natural language for {intent-based} network management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 625–639, 2021.

[44] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[45] William Johansson, Martin Svensson, Ulf E Larson, Magnus Almgren, and Vincenzo Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332. IEEE, 2014.

[46] Siva Kesava Reddy Kakarla. Incorrect interaction between Corefile and SOA record. https://github.com/coredns/coredns/issues/6420.

[47] Siva Kesava Reddy Kakarla. SERVFAIL but with records in the ANSWER section. https://github.com/coredns/coredns/issues/6419.

[48] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. SCALE: Automatically finding RFC compliance bugs in DNS nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 307–323, Renton, WA, April 2022. USENIX Association.

[50] Siva Kesava Reddy Kakarla and Brandon L Black. Sibling (In-bailiwick rule of RFC 8499) domain IP records can also be returned along with NS records. https://github.com/gdnsd/gdnsd/issues/239.

[51] Siva Kesava Reddy Kakarla and Benjamin Fry. CNAME loops throws off the server. https://github.com/hickory-dns/hickory-dns/issues/1283.

[52] Siva Kesava Reddy Kakarla and Benjamin Fry. Nonexistent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR rcode. https://github.com/hickory-dns/hickory-dns/issues/2099.

[53] Siva Kesava Reddy Kakarla and Benjamin Fry. Query matching empty wildcard node returned with NXDOMAIN instead of NOERROR. https://github.com/hickory-dns/hickory-dns/issues/1275.

[54] Siva Kesava Reddy Kakarla and Benjamin Fry. Wildcards match only one label. https://github.com/hickory-dns/hickory-dns/issues/1342.

[55] Siva Kesava Reddy Kakarla and Benjamin Fry. Zone cut NS RRs returned as authoritative records. https://github.com/hickory-dns/hickory-dns/issues/1273.

[56] Siva Kesava Reddy Kakarla, Benjamin Fry, and Jonas Bushart. Glue records returned as authoritative records by the server . https://github.com/hickory-dns/hickory-dns/issues/1272.

[57] Siva Kesava Reddy Kakarla and Miek Gieben. Handling wildcard CNAME loops. https://github.com/coredns/coredns/issues/4378.

[58] Siva Kesava Reddy Kakarla and Miek Gieben. Sibling (In-bailiwick rule of RFC 8499) domain IP records can also be returned along with NS records. https://github.com/coredns/coredns/issues/4377.

[59] Siva Kesava Reddy Kakarla and Dirkjan Ochtman. Incorrect interaction between Config.toml and SOA record. https://github.com/hickory-dns/hickory-dns/issues/2098.

[60] Siva Kesava Reddy Kakarla, Chris O'Haver, and Miek Gieben. Empty wildcard matching query returned with NXDOMAIN. https://github.com/coredns/coredns/issues/4256.

[61] Siva Kesava Reddy Kakarla, Chris O'Haver, and Kohei Yoshida. Return code for synthesized CNAME records (from wildcards and DNAMEs). https://github.com/coredns/coredns/issues/4341.

[62] Siva Kesava Reddy Kakarla, Libor Peltan, and Daniel Salzman. DNAME record returned with query domain name instead of actual name. https://gitlab.nic.cz/knot/knot-dns/-/issues/873.

[63] Siva Kesava Reddy Kakarla, Libor Peltan, and Daniel Salzman. Record incorrectly synthesized from wildcard record. https://gitlab.nic.cz/knot/knot-dns/-/issues/715.

[64] Siva Kesava Reddy Kakarla, Libor Peltan, and Daniel Salzman. Use of wildcard DNAMEs for synthesis? https://gitlab.nic.cz/knot/knot-dns/-/issues/905.

[65] Siva Kesava Reddy Kakarla, Libor Peltan, Daniel Salzman, and mscbg. DNAME-DNAME loop test case is not a loop. https://gitlab.nic.cz/knot/knot-dns/-/issues/703.

[66] Siva Kesava Reddy Kakarla, Libor Peltan, Daniel Salzman, and Vladimír Čunát. DNAME not applied more than once to resolve the query. https://gitlab.nic.cz/knot/knot-dns/-/issues/714.

[67] Siva Kesava Reddy Kakarla and Adi Roiban. Twisted Names does not set Authoritative flag in responses. https://github.com/twisted/twisted/issues/11990.

[68] Siva Kesava Reddy Kakarla and Adi Roiban. Twisted Names doesn't support wildcard records? https://github.com/twisted/twisted/issues/12043.

[69] Siva Kesava Reddy Kakarla and Twisted. Twisted-Names incorrect handling of empty non-terminals. https://github.com/twisted/twisted/issues/12042.

[70] Siva Kesava Reddy Kakarla and Peter van Dijk. Wildcard sibling glue records missing. https://github.com/PowerDNS/pdns/issues/13540.

[71] Siva Kesava Reddy Kakarla and Wouter Wijngaards. '*' in Rdata causes the return code to be NOERROR instead of NX. https://github.com/NLnetLabs/nsd/issues/152.

[72] Siva Kesava Reddy Kakarla and Wouter Wijngaards. DNAME not applied more than once to resolve the query. https://github.com/NLnetLabs/nsd/issues/151.

[73] Siva Kesava Reddy Kakarla and yadifa. Wildcard CNAME self-loop misses a record in the ANSWER section. https://github.com/yadifa/yadifa/issues/21.

[74] Siva Kesava Reddy Kakarla, yadifa, and edfeu. Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR. https://github.com/yadifa/yadifa/issues/11.

[75] Siva Kesava Reddy Kakarla, yadifa, and edfeu. Why are CNAME chains not followed? https://github.com/yadifa/yadifa/issues/10.

[76] Siva Kesava Reddy Kakarla and Shreyas Zare. Duplicate records in the ANSWER section. https://github.com/TechnitiumSoftware/DnsServer/issues/795.

[77] Siva Kesava Reddy Kakarla and Shreyas Zare. Improper handling of non-terminal wildcard. https://github.com/TechnitiumSoftware/DnsServer/issues/748.

[78] Siva Kesava Reddy Kakarla and Shreyas Zare. Improper use of wildcard record. https://github.com/TechnitiumSoftware/DnsServer/issues/792.

[79] Siva Kesava Reddy Kakarla and Shreyas Zare. Nested wildcards not handled properly. https://github.com/TechnitiumSoftware/DnsServer/issues/794.

[80] Siva Kesava Reddy Kakarla and Shreyas Zare. Synthesizing from wildcard DNAMEs over using DNAME power. https://github.com/TechnitiumSoftware/DnsServer/issues/791.

[81] Siva Kesava Reddy Kakarla and Shreyas Zare. Wildcard sibling glue records missing.
https://github.com/TechnitiumSoftware/DnsServer/issues/793.

[82] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03543*, 2023.

[83] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11:403–434, 2004.

[84] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*, pages 202–208. IEEE, 2010.

[85] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, January 1986.

[86] NLnet Labs. Nsd.
https://nlnetlabs.nl/projects/nsd/about/, 2023.
Eywa commit: https://github.com/NLnetLabs/nsd/tree/42208cc79ebd9c8594f15ef859c2fa426851ca9d
SCALE commit: https://github.com/NLnetLabs/nsd/tree/4043a5ab7be7abaec969011e48e4d0d60a0056a6.

[87] Twisted Matrix Labs. Twistednames.
https://twisted.org/, 2023.
Eywa commit: https://github.com/twisted/twisted/tree/157cd8e659705940e895d321339d467e76ae9d0a.

[88] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[89] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A formal framework for end-to-end dns resolution. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 932–949, New York, NY, USA, 2023. Association for Computing Machinery.

[90] Kenneth L McMillan and Lenore D Zuck. Compositional testing of internet protocols. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 161–174. IEEE, 2019.

[91] Kenneth L. McMillan and Lenore D. Zuck. Formal specification and testing of quic. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 227–240, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Shikhar Mehrotra. What led to internet outage that took down some major websites on july 22? https://www.republicworld.com/technology-news/other-tech-news/what-led-to-internet-outage-that-took-down-some-major-websites-on-july-22-check-out-why.html, 2023. Accessed: 2023-9-29.

[93] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. NDSS, 2024.

[94] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239):2, March 2014.

[95] Microsoft. Azure openai service.
https://azure.microsoft.com/en-us/products/ai-services/openai-service, 2023.

[96] Rajdeep Mondal. Gobgp: Prefix set match with zero masklength but nonzero range.
https://github.com/osrg/gobgp/issues/2690, 2023.

[97] Rajdeep Mondal. Batfish: Confederation sub as same as peer as.
https://github.com/batfish/batfish/issues/9263, 2024.

[98] Rajdeep Mondal. Batfish: Local preference not reset for ebgp neighbor.
https://github.com/batfish/batfish/issues/9262, 2024.

[99] Rajdeep Mondal. Gobgp: Confederation sub as equal to peer as.
https://github.com/osrg/gobgp/issues/2846, 2024.

[100] Rajdeep Mondal. Frr: Replace-as not working with confederations.
https://github.com/FRRouting/frr/issues/17887, 2025.

[101] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. What do llms need to

synthesize correct router configurations? In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 189–195, New York, NY, USA, 2023. Association for Computing Machinery.

[102] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):191, 2022.

[103] Lily Hay Newman. Decades-old code is putting millions of critical devices at risk. https://cybersecuritynews.com/bgp-error-handling-flaw/, 2023. Accessed: 2023-9-29.

[104] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[105] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. Analyzing protocol implementations for interoperability. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 485–498, 2015.

[106] Ken Pfeil. Buffer overflow in digital mapping system's pop3 server. https://www.itprotoday.com/email-and-calendaring/buffer-overflow-digital-mapping-systems-pop3-server, 2023. Accessed: 2023-9-29.

[107] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[108] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[109] Erik Romijn. Ripe ncc and duke university bgp experiment. https://labs.ripe.net/author/erik/ripe-ncc-and-duke-university-bgp-experiment/, 2023. Accessed: 2023-9-29.

[110] Javier Ron, Diogo Gaspar, Javier Cabrera-Arteaga, Benoit Baudry, and Martin Monperrus. Galápagos: Automated n-version programming with llms. *ACM Trans. Softw. Eng. Methodol.*, December 2025. Just Accepted.

[111] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *Acm sigplan notices*, 44(2):37–48, 2008.

[112] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*, 2023.

[113] Prakhar Sharma and Vinod Yegneswaran. Prosper: Extracting protocol specifications using large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 41–47, 2023.

[114] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418*, 2023.

[115] Rathin Singha. Frr: Prefix list matches mask greater than or equals. https://github.com/FRRouting/frr/issues/14280, 2023.

[116] Rathin Singha. Frr: Confederation sub as equal to peer as. https://github.com/FRRouting/frr/issues/17125, 2024.

[117] Rathin Singha. Opensmtpd: Server accepting request without appropriate headers. https://github.com/OpenSMTPD/OpenSMTPD/issues/1273, 2025.

[118] Rathin Singha. Aiosmtpd: Server accepting request without appropriate headers. https://github.com/aio-libs/aiosmtpd/issues/565#issuecomment-3930821664, 2026.

[119] Rathin Singha, Rajdeep Mondal, Ryan Beckett, Siva Kesava Reddy Kakarla, Todd Millstein, and George Varghese. MESSI: Behavioral testing of BGP implementations. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1009–1023, Santa Clara, CA, April 2024. USENIX Association.

[120] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*, page 134–155, Berlin, Heidelberg, 2024. Springer-Verlag.

[121] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

[122] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[123] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346*, 2023.

[124] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[125] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221*, 2023.

[126] Yunze Wei, Kaiwen Chi, Shibo Du, Xiaohui Xie, Ziyu Geng, Yuwei Han, Zhen Li, Zhanyou Li, and Yong Cui. Large language model driven automated network protocol testing. In *Proceedings of the 2025 Applied Networking Research Workshop*, ANRW '25, page 32–38, New York, NY, USA, 2025. Association for Computing Machinery.

[127] Wikipedia. 2022 rogers communications outage. https://en.wikipedia.org/wiki/2022_Rogers_Communications_outage, 2023. Accessed: 2023-9-29.

[128] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[129] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748*, 2023.

[130] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*, pages 435–450, 2021.

[131] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 272–286, 2021.

[132] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[133] Shreyas Zare and Community. Technitium dns server. https://technitium.com/dns/, 2023.
Eywa commit: https://github.com/TechnitiumSoftware/DnsServer/tree/d4352680b3f14fa2884fc8b7fa9c7772379bbc61.

A Regex Implementation

We show EYWA's minimal regular expression implementation in Figure 8. The implementation supports regular expression range, union, sequence, and iteration (star) constructs, defined by simple `Regex` type. To preserve branches for symbolic execution, rather than build a finite automaton, the matching logic uses a notion of regular expression continuation `RegexCont`, which is a linked list of regular expressions that must be matched after the current regular expression. Initially, this list is empty but gets added to any time there is a regular expression sequence (`regex->op == SEQ`) case. Continuations allow for a simple recursive implementation, and work well with symbolic execution since the regular expression itself is always concrete. Path constraints come from matches against the symbolic text `*text` such as the comparison with `'0'` and the case for `regex->op == RANGE`. For the `match` function, we pass in a "null" continuation to begin with.

For each RegexModule, EYWA converts them to functions with C expressions that evaluate to boolean values. The translation is straightforward – for instance, a Python constraint such as `eywa.RegexModule("[a-z]*", a1)` will generate the C code:

```
Regex r1; r1.clo = 'a'; r1.chi = 'z';
Regex r2; r2.op = STAR; r2.left = r1;
klee_assume(match(&r2, &a1));
```

EYWA then calls a custom `match` function to check if the string matches the regular expression. The `match` function is a minimal regular expression matching implementation written by hand and is amenable to symbolic execution. Klee will encode the regex match condition and other conditions symbolically and solve them as part of its path exploration.

```
typedef enum { OR, SEQ, STAR, RANGE } RegexOp;
typedef struct Regex Regex;
struct Regex {
    RegexOp op; int clo; int chi;
    Regex* left;Regex* right;
};
typedef struct RegexCont RegexCont;
struct RegexCont {
    Regex* regex;
    RegexCont* next;
}
// Matching logic for a regular expression with a
// continuation.
static int match(Regex* regex, RegexCont* cont,
char *text) {
  if (regex == NULL) {return *text == '\0';}
  if (regex->op == OR) {
    return match(regex->left, cont, text) ||
    match(regex->right, cont, text);
  }
  if (regex->op == SEQ) {
    RegexCont c;
    c.next = cont; c.regex = regex->right;
    return match(regex->left, &c, text);
  }
  if (regex->op == STAR) {
    Regex r; r.op = SEQ; r.left = regex->left;
    r.right = regex;
    return match(cont->regex, cont->next, text) ||
    (*text != '\0' && match(&r, cont, text));
  }
  if (regex->op == RANGE) {
    char c = *text++;
    return c != '\0' && c >= regex->clo &&
    c <= regex->chi &&
    match(cont->regex, cont->next, text);
  }
  return 0;
}
// Matching logic for a regular expression.
static int match(Regex* regex, char *text) {
  RegexCont cont;
  cont.next = NULL; cont.regex = NULL;
  return match(regex, &cont, text);
}
```

Figure 8: EYWA's minimal regular expression implementation that is amenable to symbolic execution.

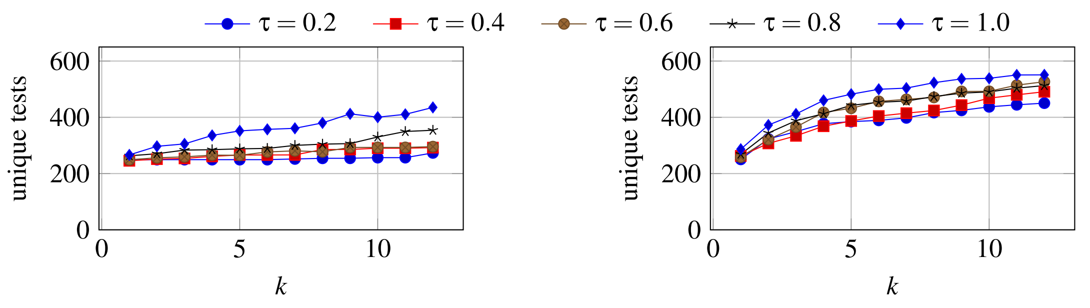
B Hyperparameter selection

To understand how the temperature τ and the number of attempts k affect test case generation, we took the CNAME DNS model and varied the temperature and counted the number of tests generated for each k averaged over 10 runs. We plot this count vs. k in Figure 9 for values of τ ranging from 0.2 to 1.0. Results for the other models were similar. We can see from the graph that there are greatly diminishing returns around $k = 10$, whereas τ appears to have less effect on the test generation for values above 0. For this reason, we selected $k = 10$ and $\tau = 0.6$ in our experiments, which we believe represents a reasonable trade-off between efficiency and test coverage.

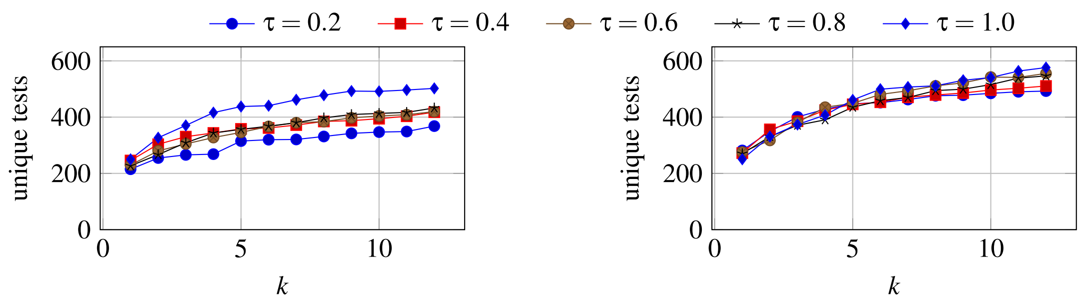
C Graph API

Figure 10 shows the construction of the dependency graph for the RMAP-PL BGP model from our experiments, which matches routes against a prefix list used in a route-map stanza. Each of the modules is first defined independently using the prompting technique shown in Figure 1a, allowing us to modularize the overall BGP semantics into smaller, reusable components. We then specify the inter-module dependencies explicitly using Eywa's built-in graph API. Concretely, we first initialize an empty graph and incrementally add all dependencies using the `CallEdge` and `Pipe` methods, thereby constructing a directed acyclic graph (DAG) that captures the execution structure of the model.

In this example, we use a `Pipe` between the functions `checkValidInputs` and `isMatchRouteMapStanza`, since the former enforces structural and semantic constraints on the inputs before they are processed by the latter. In con-



(a) DNAME (b) IPV4



(c) WILDCARD (d) CNAME

Figure 9: Hyperparameter analysis for DNS models. The plots show how the total number of unique tests vary as we increase the number of runs for different models.

```
# initialize the dependency graph
g = eywa.DependencyGraph()

# add the call edges to the graph
g.CallEdge(isValidPrefixList,
    [prefixLengthToSubnetMask])

g.CallEdge(isValidRoute,[prefixLengthToSubnetMask])

g.CallEdge(checkValidInputs,
    [isValidPrefixList,isValidRoute])

g.CallEdge(isMatchPrefixListEntry,
    [prefixLengthToSubnetMask])

g.CallEdge(isMatchRouteMapStanza,
    [isMatchPrefixListEntry])

# add the pipe
g.Pipe(isMatchRouteMapStanza,checkValidInputs)

# synthesize final model
final_model = g.Synthesize()
```

Figure 10: EYWA code for building a dependency graph for the BGP RMAP-PL model in Table 2

trast, the remaining edges are specified as `CallEdge`s, because in those cases the output of one function is directly consumed by another as part of the computation. Multiple `CallEdge`s are required due to hierarchical dependencies in the logic. For instance, `isMatchRouteMapStanza` depends on `isMatchPrefixListEntry`, which in turn invokes `prefixLengthToSubnetMask` to perform prefix-length computations. Additionally, unlike most other modules, `checkValidInputs` depends on two distinct validation routines—`isValidRoute` and `isValidPrefixList`. These must therefore be provided together as a list when invoking the `CallEdge` method.

After defining all nodes and edges, we synthesize the complete model using the `Synthesize` utility. During synthesis of any module with incoming `CallEdge`s, the generated prompt automatically includes a description of all dependency functions along with their corresponding C prototypes. Figure 11 illustrates the prompt generated due to the `CallEdge` from `prefixLengthToSubnetMask` to `isMatchPrefixListEntry` when synthesizing the latter. This mechanism is crucial, as it ensures that the LLM is fully aware of all available helper functions and their interfaces, enabling consistent and compositional code generation. Finally, Eywa assembles all synthesized functions into the final program according to their topological ordering in the dependency graph, guaranteeing correctness of compilation and execution order.

```c
#include <stdint.h>
...

typedef struct {
    uint32_t prefix;
    uint8_t prefixLength;

    ...

} Route;

typedef struct {
    uint32_t prefix;
    uint8_t prefixLength;
    uint32_t le;
    uint32_t ge;
    bool any;
    bool permit;
} PrefixListEntry;

// a function that takes as input the prefix
// length and converts it to the corresponding
// unsigned integer representation
//
// Parameters:
//     maskLength: The length of the prefix
//
// Return Value:
//     The unsigned integer representation
//     of the prefix length

uint32_t prefixLengthToSubnetMask(
uint32_t maskLength);

// A function that takes as input a prefix
// list entry and a BGP route advertisement.
//
// If the route advertisement matches the
// prefix, then the function should return
// the value of the permit flag.
//
// In case there is no match, the function
// should vacuously return false.
//
// Parameters:
//     route: Route to be matched
//     pfe: Prefix list entry
//
// Return Value:
//     True if the route matches the prefix
//     list entry

bool isMatchPrefixListEntry(Route route,
PrefixListEntry pfe) {

...

}
```

Figure 11: Prompt for the `prefixLengthToSubnetMask` to `isMatchPrefixListEntry` dependency in Figure 10.

D System Prompt

Figure 12 shows the system prompt used for all the LLM calls for our experiments.

```
Your goal is to implement the C function provided by
the user. The result should be the complete
implementation of the code, including:
  1. All the import statements needed, including those
     provided in the input. All the imports from the
     input should be included.
  2. All the type definitions provided by the user.
     The type definitions should NOT be modified
  3. ONLY write in the function that has 'implement me'
       written in its function body.
  4. If any additional function prototypes are
     provided, you can use them as helper functions.
     There is no need to define them. You can assume
     they will be done later by the user.
  5. Do NOT change the provided function
     declarations/prototypes.
  6. Whenever you define a 'struct', write it in one
     line. Do not put newline. e.g. struct{int x; int
          y;}

DO NOT add a `main()` function or any examples, just
     implement the function.
DO NOT USE fenced code blocks, just write the code.
DO NOT USE C strtok function. Implement your own.


Example Input:
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <klee/klee.h>
#include <stdio.h>
typedef uint32_t myint;
myint add_one(myint x) {
  // implement me
}

Example Output:
#include <stdint.h>
...
myint add_one(myint x) {
    return x + 1
}
```

Figure 12: The system prompt which we used for all LLM calls.

E SMTP Server Model

Figure 13 shows the EYWA generated code for SMTP server model.

F TCP Model

Figure 14 shows the TCP state transition code generated by EYWA. Figure 15 shows the state graph.

```c
char* smtp_model(State state, char* input){

char* response = malloc(512 * sizeof(char));

switch(state) {
 case INITIAL:
  if(strcmp(input,"HELO")==0){
    strcpy(response,"250 Hello");state=HELO_SENT;
  }
  else if(strcmp(input,"EHLO")==0){
    strcpy(response, "250-Hello\n250-SIZE
     512\n250-8BITMIME\n250 OK");state=EHLO_SENT;
  }
  else{
    strcpy(response,"503 Bad sequence of commands");
  }break;
 case HELO_SENT:
 case EHLO_SENT:
  if(strncmp(input,"MAIL FROM:",10)==0{
    strcpy(response, "250 OK");
    state=MAIL_FROM_RECEIVED;
  }
  else{
    strcpy(response, "503 Bad sequence of commands");
  }break;
 case MAIL_FROM_RECEIVED:
  if(strncmp(input,"RCPT TO:",8)==0){
    strcpy(response, "250 OK");state=RCPT_TO_RECEIVED;
  }
  else{
    strcpy(response,"503 Bad sequence of commands");
  }break;
 case RCPT_TO_RECEIVED:
  if(strcmp(input,"DATA")==0){
    strcpy(response,"354
    End with <CR><LF>.<CR><LF>");state=DATA_RECEIVED;
  }
  else{
    strcpy(response, "503 Bad sequence of commands");
  }break;
 case DATA_RECEIVED:
  if(strcmp(input,".")==0){
    strcpy(response,"250 OK");state=INITIAL;
  }
  else{
    strcpy(response,"");
  }break;
 case QUITTED:
  strcpy(response,"221 Bye");state=INITIAL;break;
 default:
  strcpy(response,"500 error, command
     unrecognized");break;
 return response;
}
```

Figure 13: EYWA generated code for SMTP server model.

```c
char* tcp_state_transition(TCPState state, char* input) {

switch (state) {
 case CLOSED:
  if (!strcmp(input,"APP_PASSIVE_OPEN")) return "LISTEN";
  if (!strcmp(input,"APP_ACTIVE_OPEN")) return
     "SYN_SENT";
  break;

 case LISTEN:
  if (!strcmp(input,"RCV_SYN")) return "SYN_RECEIVED";
  if (!strcmp(input,"APP_SEND")) return "SYN_SENT";
  if (!strcmp(input,"APP_CLOSE")) return "CLOSED";
  break;

 case SYN_SENT:
  if (!strcmp(input,"RCV_SYN")) return "SYN_RECEIVED";
  if (!strcmp(input,"RCV_SYN_ACK")) return "ESTABLISHED";
  if (!strcmp(input,"APP_CLOSE")) return "CLOSED";
  break;

 case SYN_RECEIVED:
  if (!strcmp(input,"APP_CLOSE")) return "FIN_WAIT_1";
  if (!strcmp(input,"RCV_ACK")) return "ESTABLISHED";
  break;

 case ESTABLISHED:
  if (!strcmp(input,"APP_CLOSE")) return "FIN_WAIT_1";
  if (!strcmp(input,"RCV_FIN")) return "CLOSE_WAIT";
  break;

 case FIN_WAIT_1:
  if (!strcmp(input,"RCV_FIN")) return "CLOSING";
  if (!strcmp(input,"RCV_FIN_ACK")) return "TIME_WAIT";
  if (!strcmp(input,"RCV_ACK")) return "FIN_WAIT_2";
  break;

 case FIN_WAIT_2:
  if (!strcmp(input,"RCV_FIN")) return "TIME_WAIT";
  break;

 case CLOSE_WAIT:
  if (!strcmp(input,"APP_CLOSE")) return "LAST_ACK";
  break;

 case CLOSING:
  if (!strcmp(input,"RCV_ACK")) return "TIME_WAIT";
  break;

 case LAST_ACK:
  if (!strcmp(input,"RCV_ACK")) return "CLOSED";
  break;

 case TIME_WAIT:
  if (!strcmp(input,"APP_TIMEOUT")) return "CLOSED";
  break;
}
return "INVALID";
}
```

Figure 14: EYWA generated code for TCP state machine

User_Prompt

```
Create a python dictionary that maps the state
transitions: (state,input) --> state
as per the following C code snippet:

tcp_state_transition(State state, char* input) {
    <TCP state transition code generated by LLM>
}
```

Output_Format

```
A python dictionary like
{(state1, input1): state2,
(state3, input2): state4, ...}
```

GPT_Response

~~~
Here is the Python dictionary that maps the
state transitions:

```python
state_transitions = {
    (CLOSED, 'APP_PASSIVE_OPEN'): LISTEN,
    (CLOSED, 'APP_ACTIVE_OPEN'): SYN_SENT,
    (LISTEN, 'RCV_SYN'): SYN_RECEIVED,
    (LISTEN, 'APP_SEND'): SYN_SENT,
    (LISTEN, 'APP_CLOSE'): CLOSED,
    (SYN_SENT, 'RCV_SYN'): SYN_RECEIVED,
    (SYN_SENT, 'RCV_SYN_ACK'): ESTABLISHED,
    (SYN_SENT, 'APP_CLOSE'): CLOSED,
    (SYN_RECEIVED, 'APP_CLOSE'): FIN_WAIT_1,
    (SYN_RECEIVED, 'RCV_ACK'): ESTABLISHED,
    (ESTABLISHED, 'APP_CLOSE'): FIN_WAIT_1,
    (ESTABLISHED, 'RCV_FIN'): CLOSE_WAIT,
    (FIN_WAIT_1, 'RCV_FIN'): CLOSING,
    (FIN_WAIT_1, 'RCV_FIN_ACK'): TIME_WAIT,
    (FIN_WAIT_1, 'RCV_ACK'): FIN_WAIT_2,
    (FIN_WAIT_2, 'RCV_FIN'): TIME_WAIT,
    (CLOSE_WAIT, 'APP_CLOSE'): LAST_ACK,
    (CLOSING, 'RCV_ACK'): TIME_WAIT,
    (LAST_ACK, 'RCV_ACK'): CLOSED,
    (TIME_WAIT, 'APP_TIMEOUT'): CLOSED
}
```
~~~

Figure 15: Second LLM prompt and response to generate the TCP state graph as a Python dictionary.