

Eywa: Automating Model Based Testing using LLMs

Rajdeep Mondal*
UCLA

Rathin Singha*
UCLA

Todd Millstein
UCLA

George Varghese
UCLA

Ryan Beckett
Microsoft Research

Siva Kesava Reddy Kakarla
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 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 32 unique bugs across widely used DNS, BGP, and SMTP implementations, 15 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, 20–22, 45, 61, 64], outages [4, 26, 27, 40, 55, 67, 79], and performance problems [42]. For example, in 2022 a bug in Akamai’s DNS software caused a global outage for nearly 30,000 sites [55]. Recently, mishandling of a corrupted BGP attribute in router software caused invalid routes to spread before shutting down BGP sessions in remote networks [27].

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, in MBT the user formulates a simplified model of a protocol – for instance as a logical specification, abstract state machine, or a reference implementation – and then uses existing program analysis techniques such as symbolic execution [10, 36]

to enumerate test cases from this model that cover a wide variety of protocol behaviors. Researchers have used MBT to find numerous bugs in implementations of QUIC [53, 54], BGP [72], DNS [45, 52], and various LTE [43] protocols.

Since handwritten protocol models are typically orders of magnitude simpler than the implementation’s source code, symbolic execution scales well on these models and ameliorates the infamous path explosion problem [10] that arises when analyzing source code directly. MBT also applies to testing proprietary protocol implementations where the source code is unavailable but the standards are available. However, a key bottleneck to using MBT is that users must carefully build a model of the protocol under test. This requires reading multiple RFCs to understand the detailed protocol behavior, and to translate that understanding into either a formal model or a reference implementation. Hence, applying MBT to any new protocol incurs a significant up-front cost before testing, thereby limiting its applicability in practice.

In this work, we ask: “*can we automatically derive protocol models for testing using LLMs?*” We answer this question affirmatively by exploiting recent advances in large language models (LLMs) to use their knowledge base for automated testing. For instance, as part of their training data, LLMs have ingested substantial protocol knowledge from RFCs, standards, networking forums & blogs, as well as other online resources and documents. However, using LLMs to test network protocols in practice requires overcoming several non-trivial challenges.

- (1) **LLM errors:** LLMs make mistakes and can easily hallucinate to produce flawed protocol models.
- (2) **Valid inputs:** Protocol inputs can have complex requirements and data dependencies. For instance, creating a valid TCP header, a DNS zone, or a BGP route to test is non-trivial, and invalid inputs can render a test useless.
- (3) **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 and increases the likelihood of model errors.
- (4) **Protocol state:** While DNS is stateless, BGP depends on earlier routes, and protocols like SMTP and TCP use state machines that require the right sequence of inputs to drive an implementation to “deeper” states.

*These authors contributed equally to this work

In this paper, we describe an approach that uses LLMs to automatically derive protocol models for MBT while addressing each of the challenges described above. In our approach, the user provides (1) a short description of which protocol component they wish to test, (2) its input and output types, and (3) a graph that specifies how the different components (or “modules”) should be composed together. Our testing framework then automatically generates a model for each protocol module, and it uses the given graph to stitch these modules together to produce the end-to-end protocol model. The testing framework then performs symbolic execution on this model to systematically enumerate test cases for the protocol that the user can then run against real protocol implementations.

Our approach effectively resolves the challenges identified earlier. First, because LLMs can produce flawed models (*challenge #1*), rather than relying on the model output we use *differential testing* on multiple implementations; if the tests produce different outputs in different protocol implementations, often some implementation is buggy. Differential testing is particularly natural for protocol testing, where we often have many implementations that must interoperate.

Second, by decomposing complex protocols like BGP into smaller modules that are composed through an explicit dependency graph, the LLM can scale to construct larger protocol models (*challenge #3*). When constructing each protocol module, the LLM has access to the type signatures and descriptions of depended upon modules, but not their implementation details. Enforcing validity constraints on protocol inputs (*challenge #2*) is achieved through a simple form of sequential composition, where the user defines one module to enforce the validity constraints and pushes its output into a second module that assumes the inputs are valid.

Finally, we handle stateful protocols (*challenge #4*) through a separate invocation of the LLM to generate a state graph of the protocol. 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 the test cases. We use this generated state graph to search for inputs that lead to particular protocol states and then use that input sequence to drive the protocol implementation to those states as needed for different tests.

We have instantiated our approach in a tool called Eywa, which is implemented as a Python library. Given short descriptions of protocol modules and a dependency graph, Eywa automatically constructs a protocol model through multiple modular invocations of an LLM to produce executable C code. Eywa simultaneously compiles a *symbolic test harness* to integrate this model with the Klee [10] symbolic execution engine. By invoking Klee on the end-to-end protocol model, Eywa obtains a high-coverage set of test

cases and translates them back to Python for the library user. The user can then apply these tests to real protocol implementations.

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 [45, 72], Eywa constructed similar models with minimal user input. Using Eywa, we revealed 32 unique bugs across popular DNS, BGP, and SMTP implementations. Of these bugs, 15 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 showcase Eywa’s utility that found 32 unique bugs across popular DNS, BGP and SMTP implementations, including 15 previously undiscovered bugs.
- A small case study of SMTP that shows how to handle stateful protocols by using the LLM separately to generate a state machine of the protocol as a graph, and using graph traversal 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 [45] 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. To achieve these results however, SCALE required extensive manual mathematical formulation of the DNS RFCs [44] and later arduous manual construction of an executable model [45]. 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 11 new DNS bugs that SCALE missed, all with minimal user effort.

```

# 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("[a-z*(\.[a-z]*)*", query)
da = eywa.FuncModule("dname_applies", "If a DNAME record matches a query.", [query, record, result])
ra = eywa.FuncModule("record_applies", "If a DNS record matches a query.", [query, record, result])
# Create the dependency graph to connect the modules.
g = eywa.DependencyGraph(); g.Pipe(valid_query, ra); 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")

```

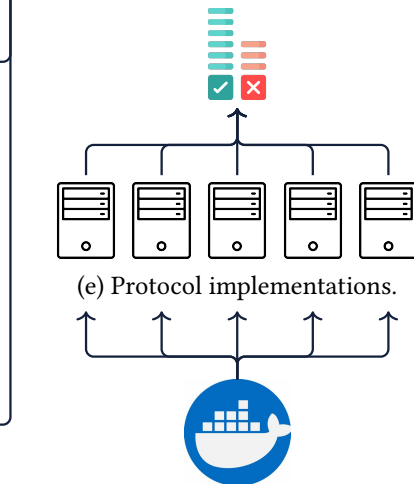
(a) User input to EYWA to generate a model for DNS lookup.

```

// Pre-defined modules implemented by Eywa.
bool valid_query(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.



(e) Protocol implementations.

(d) Differential testing setup.

Test inputs (4)

```

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

```

(c) Klee invocation.

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.

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 1a, 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 (`rtype`), 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 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 tests cases, each of which is a list of arguments and the expected result, for example:

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

How it works. Eywa first prompts the LLM to implement each function module in C code (circle1) 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

```
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;
    }
    // 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.

C. Eywa also creates a main function that enables symbolic execution by declaring symbolic inputs for the model that the LLM produces (circle2).

2.2 Dealing with imperfect models

LLMs can make mistakes, and an imperfect model can result in test inputs that are either invalid or otherwise not useful. 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. Hence even if some LLM-generated models are incorrect or incomplete, those errors can be compensated for by other generated models.

Second, as described earlier we employ differential testing to find bugs, comparing the results of multiple implementations on each test case. This means that we do not rely on the LLM-generated model’s result, but rather we simply use it to generate inputs that drive the model down different execution paths. As a result, the model itself can also be simpler than would otherwise be required. In our example, we only model the logic that determines whether a resource record applies to a given query, but we need not model the logic of 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 §4.3).

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 [23] 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>`. The DNAME record above matches the query, so nameserver implementations are expected to return the DNAME record along with a synthesized CNAME record, as follows:

```
*.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.

2.4 Limitations of the approach

For Eywa to be effective, the LLM must have a strong understanding of the protocol that it will be modeling. For many protocols (e.g., DNS, BGP, ICMP, etc.) many LLMs today already understand them well due to the vast amount of knowledge widely available for the protocols. Hence in our running example we only had to provide a single sentence to describe the goal of our `record_applies` function. For new protocols deployed beyond the LLMs training cutoff date or for proprietary protocols whose specifications are not publicly available, users must provide additional protocol-specific documentation, for instance by fine-tuning the LLM or providing things like protocol RFCs within Eywa’s prompts.

3 THE EYWA LIBRARY AND RUNTIME

In this section we describe Eywa’s design and runtime.

3.1 Eywa Architecture

Eywa’s architecture comprises several interacting components whose workflow is illustrated in Figure 3. To start, the user (extreme left) uses the Eywa library to define protocol modules (functions) with arguments and return values and composes them together through a dependency graph, as shown in the example from Figure 1. From these inputs, Eywa then internally invokes its *Prompt Generator* and *Symbolic Compiler* to produce the model.

The *Prompt Generator* builds an LLM prompt for each `FuncModule` declared by the user. It invokes the LLM on each prompt to produce a corresponding C function. The *Symbolic Compiler* uses the dependency graph to build the *Symbolic Harness*, which is the main function that represents the full model. The function’s inputs are declared symbolic using the Klee API, to make the code amenable to symbolic execution. The *Symbolic Harness* also produces the C functions for predefined modules like the `RegexModule`. The complete model consists of the outputs of both the *Prompt Generator* and *Symbolic Compiler*. This code synthesis process is repeated k times to generate k different models. Finally, the *Test Generator* invokes Klee on these models, extracts the generated test cases from Klee, 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 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

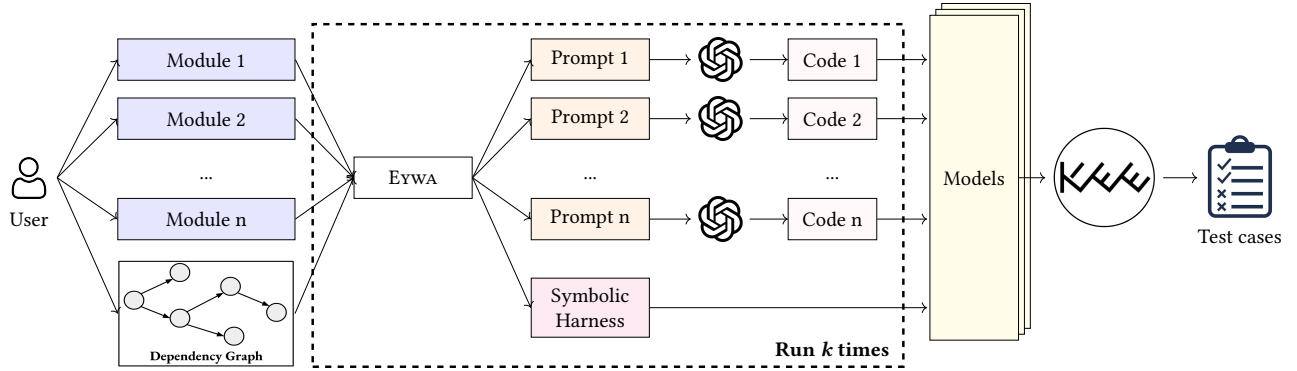


Figure 3: Architecture of EYWA. It takes modules and dependencies and produces k different protocol models.

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

Figure 4: Summary of EYWA types and abstractions.

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 are shown in Figure 4. The library allows for the creation of function arguments with standard types such as booleans, characters, strings, fixed bit-width integers, enums, arrays, structs, as well as type aliases that allow for associating custom names with types (e.g., to help the LLM understand a type’s meaning). For types with potentially unbounded size (e.g., `eywa.String()`), users must also provide hints to bound the size of that type to limit the size and number of test cases that EYWA will produce. Users create functions by giving them a name, a natural language description of their purpose, and a list of arguments. Each argument also has a name and description as well as an associated type.

3.3 EYWA Support for Modularity

As discussed in Section §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 Prompt Generator

Given a set of modules along with the dependency graph for a model, EYWA invokes its *Prompt Generator* to build an LLM prompt that will lead the LLM to implement each module and then proceeds to combine them 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 to guide the behavior of the LLM.

Consider again the example model defined using the EYWA library 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:

```

#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 DNS 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.

- 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 an example of a valid input and output.

To see additional examples, refer to [Appendices C](#) and [D](#).

3.5 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.

3.6 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 EVALUATION

4.1 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 [58]. 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 a $k = 10$ 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.

4.2 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.

Overview of models. We constructed eight models of the DNS using Eywa (shown in [Table 2](#)) and tested different parts of the protocol for authoritative nameservers. To better understand when Eywa performs well and when it does not,

Protocol	Tested Implementations
DNS	BIND [19], COREDNS [13], GDNSD [5], NSD [49] HICKORY [32], KNOT [23], POWERDNS [18] TECHNIUM [85], YADIFA [28], TWISTED [50]
BGP	FRR [16], GOBGP [17], BATFISH [30]
SMTP	AIOSMTPD [2], SMTPD [24], OPENSMTDP [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.

we attempted to craft a diverse set of models. We created (1) simple models that test matching behavior for individual DNS records, (2) end-to-end models that capture the entire DNS lookup behavior, and (3) specialized models to target unusual or corner case behavior.

Specifically, the first four models (CNAME, DNAME, WILDCARD, IPV4) each determines whether a DNS query matches a single record type in a DNS zone. The FULLLOOKUP model implements the entire DNS authoritative lookup procedure for a query and a zone file. Two other models, AUTHORITY and RCODE, are similar to FULLLOOKUP but return only a part of the DNS response – in particular, the authoritative flag and return code respectively, rather than the full DNS response. Finally, we include a LOOP model that tests a corner case of the DNS protocol. In particular, we ask Eywa to create a model that counts the number of times a DNS query is rewritten for a given zone file. This forces Eywa to explore inputs that result in recursively looking up an answer multiple times or even indefinitely. These models have a greater number of test cases than FULLLOOKUP because of the same Klee timeout for all of them. However, FULLLOOKUP is more complex and requires more time to generate a test case.

```
#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.

For the Border Gateway Protocol (BGP), we modeled several features and their combinations. Specifically our models capture the logic of BGP confederations (CONFED), how route-reflector servers interact with clients and non-clients and external ASes (RR), how route-maps and prefix lists process BGP route advertisements (RMAP-PL), and the combination of route reflectors with route-maps (RR-RMAP).

Finally, for the Simple Mail Transfer Protocol (SMTP), we created a model of an SMTP server. The model responds to incoming messages/requests depending on the server state, including emitting an appropriate error message for invalid requests. The model is implemented as a function that takes two parameters, state and input, and returns the corresponding server response. Figure 6 shows the prompt generated by Eywa and Figure 7 shows the resulting code from the LLM.

Overview of testing setup. Seven of the ten DNS implementations that we tested were also tested by SCALE [45]. For those seven we additionally tested both the old versions (prior to bug fixes) of each implementation along with current versions. Doing so allowed us to identify the overlap between bugs found by Eywa and SCALE as well as any new bugs that were yet undiscovered. To test each DNS implementation we use differential testing as described in Figure 1. We created scripts to initialize a Docker [57] container for each DNS implementation and version. Each container serves a single zone file as an authoritative zone, and we used the dnspython [14] library to construct DNS queries and send them to each container. After receiving the response, we compared the answers to find which parts differ (e.g., answer, authoritative section, flags, additional section, return code).

```

char* smtp_server_response(State state, char* input) {
    char* response = malloc(512 * sizeof(char));
    if(state == 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");
        }
    } else if(state == HELO_SENT || state == 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");
        }
    } else if(state == 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");
        }
    } else if(state == 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");
        }
    } else if(state == DATA_RECEIVED) {
        if(strcmp(input, ".") == 0) {
            strcpy(response, "250 OK");
            state = INITIAL;
        } else {
            strcpy(response, "");
        }
    } else if(state == QUITTED) {
        strcpy(response, "221 Bye");
        state = INITIAL;
    } else {
        strcpy(response, "500 error, command unrecognized");
    }
    return response;
}

```

Figure 7: Eywa generated code for SMTP server Model.

Since many tests can trigger the same bug, to make it easier to identify unique root causes, we take each implementation whose response is not part of the majority group and classify the “reason” for the disagreement as a tuple that abstracts the relevant parts that differ. For instance, a response that returns code NXDOMAIN instead of NOERROR might have the tuple (COREDNS, rcode, NXDOMAIN, NOERROR). For each such unique tuple we manually inspected the failing tests and reported any bugs found with the implementation maintainers.

For the BGP implementations we set up a Docker network containing three nodes R1, R2 and R3 connected in series. We run the BGP implementation under test on R2 and R3 but run an ExaBGP [15] instance on R1 so that we can inject routes into R2. We then check 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 in similar fashion. 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, as described earlier each test case is an (state, input) pair. Hence running each test case requires us to first drive each SMTP implementation into the desired state. To achieve this, Eywa uses *another LLM call* to transform the generated SMTP server code into a state transition graph (see Figure 8). For each test case, Eywa uses breadth-first search (BFS) on the state graph to determine an input sequence that will transition the server from the initial state to the desired state for that test, and this input sequence is prepended to the test’s input. The server implementations are hosted at IP 127.0.0.1 and port 8025; another process on the same machine generates the input requests for the servers. After executing each test case, the server is reset to its initial state.

4.3 Results

In this section, we describe 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.

1. Eywa produces tests quickly. 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. On the other hand, all our BGP models are bounded in size so Klee always terminated within 5-10 seconds (well before the timeout).

2. Eywa produces high-quality models. We manually inspected Eywa’s generated implementations to assess their quality. For each of the models from Table 2, we found that Eywa produced models that largely capture the intended protocol semantics faithfully. For instance, for the DNS FULLLOOKUP task, which is the most challenging DNS task since it asks for a full model of DNS nameserver lookup, the LLM correctly implemented the behavior of each record type including DNAME (suffix rewrite), CNAME (exact rewrite),

Protocol	Implementation	Description	Bug Type	New bug?
DNS	BIND	Sibling glue record not returned.	Wrong Additional	○
DNS	BIND	Inconsistent loop unrolling.	Wrong Answer	●
DNS	COREDNS	Wildcard CNAME and DNAME loop.	Server Crash	○
DNS	COREDNS	Sibling glue record not returned.	Wrong Additional	○
DNS	COREDNS	Returns SERVFAIL yet gives an answer.	Wrong Answer	●
DNS	COREDNS	Missing record for CNAME loop.	Wrong Answer	●
DNS	COREDNS	Returns a non-existent out-of-zone record.	Wrong Answer	●
DNS	COREDNS	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
DNS	COREDNS	Wrong RCODE for empty non-terminal wildcard.	Wrong Return Code	●
DNS	GDNSD	Sibling glue record not returned.	Wrong Additional	○
DNS	HICKORY	Wildcard CNAME and DNAME loop.	Server Crash	○
DNS	HICKORY	Incorrect handling of out-of-zone record.	Wrong Answer	●
DNS	HICKORY	Wildcard match only one label.	Wrong Answer	○
DNS	HICKORY	Wrong RCODE for empty non-terminal wildcard.	Wrong Return Code	●
DNS	HICKORY	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	●
DNS	HICKORY	Glue records returned with authoritative flag.	Wrong Flags	○
DNS	HICKORY	Authoritative flag set for zone cut NS records.	Wrong Flags	○
DNS	KNOT	DNAME record name replaced by query.	Wrong Answer	●
DNS	KNOT	Wildcard DNAME leads to wrong answer.	Wrong Answer	●
DNS	KNOT	Error in DNAME-DNAME loop Knot test.	Faulty Knot Test	○
DNS	KNOT	DNAME not applied recursively.	Wrong Answer	○
DNS	KNOT	Incorrect record synthesis when '*' is in query.	Wrong Answer	○
DNS	NSD	DNAME not applied recursively.	Wrong Answer	○
DNS	NSD	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
DNS	POWERDNS	Sibling glue record not returned due to wildcard.	Wrong Additional	●
DNS	TECHNITIUM	Sibling glue record not returned.	Wrong Additional	○
DNS	TECHNITIUM	Synthesized wildcard instead of applying DNAME.	Wrong Answer	●
DNS	TECHNITIUM	Invalid wildcard match.	Wrong Answer	○
DNS	TECHNITIUM	Nested wildcards not handled correctly.	Wrong Answer	●
DNS	TECHNITIUM	Duplicate records in answer section.	Wrong Answer	○
DNS	TECHNITIUM	Wrong RCODE for empty nonterminal wildcard.	Wrong Return Code	●
DNS	TWISTED	Empty answer section with wildcard records.	Wrong Answer	○
DNS	TWISTED	Missing authority flag and empty authority section.	Wrong Flags	○
DNS	TWISTED	Wrong RCODE for empty nonterminal wildcard.	Wrong Return Code	●
DNS	TWISTED	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
DNS	YADIFA	CNAME chains are not followed.	Wrong Answer	○
DNS	YADIFA	Missing record for CNAME loop.	Wrong Answer	●
DNS	YADIFA	Wrong RCODE for CNAME target.	Wrong Return Code	○
BGP	FRR	Prefix list matches mask greater than or equals.	Wrong Policy	○
BGP	FRR	Confederation sub AS equal to peer AS.	Wrong Policy	●
BGP	FRR	Replace-AS not working with confederations.	Wrong Policy	●
BGP	GoBGP	Prefix set match with zero masklength but nonzero range.	Wrong Policy	○
BGP	GoBGP	Confederation sub AS equal to peer AS.	Wrong Policy	●
BGP	BATFISH	Local preference not reset for EBGp neighbor.	Wrong Policy	●
BGP	BATFISH	Confederation sub AS same as peer AS.	Wrong Policy	●
SMTP	AioSMTPD	Server accepting request without appropriate headers	Input Validation	●

Table 3: The bugs found in the DNS, BGP, and SMTP protocol implementations tested by EYWA.

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

and wildcard (partial match). The LLM did not perfectly implement the “closest encloser” semantics of DNS lookup (doing so would require creating a complex data structure), instead typically performing a “first-match” semantics by iterating through the list of zone records sequentially. While technically incorrect, this is a close approximation of the true behavior and produced an effective set of test cases when combined with symbolic execution.

For BGP, EyWA’s models again were largely accurate. With careful prompt engineering, we could even force the LLM to cover missed corner cases. However, for BGP confederations, we observed that the LLM struggled to update the AS paths properly, despite trying multiple prompt variations. However,

because we employ differential testing and ignore model results, the model’s logic still produced useful test cases. Finally, for the SMTP server, EyWA almost always generated a perfect model.

In several cases, the LLM originally implemented a flawed model due to its misunderstanding of the side effects of the C strtok function. As a simple fix, we updated the system prompt to instruct the LLM to avoid this function, which resolved the issue. Finally, across all of our experiments the LLM only produced a single C model that failed to compile.

3. EyWA finds many implementation bugs. To evaluate the quality of tests generated by EyWA, we compared the bugs found by prior work with manual models, specifically with SCALE[45] for DNS and MESSI [72] for BGP. Compared to SCALE or MESSI, which used a carefully designed DNS/BGP model constructed manually from RFCs, EyWA requires orders of magnitude less modeling effort by leveraging LLMs. On the other hand, we expected the hand-written model to be more effective at revealing bugs.

We present the results from differential testing in Table 3. The table lists the bugs identified in each of the tested protocol implementations, along with a description of the bug and its type/effect. Additionally, we record whether each bug was found by test cases produced by the prior SCALE or MESSI tool, or whether it would have been found had the SCALE tests been run on the newly added implementations.

In the comparative analysis of DNS implementations, EyWA successfully identified a total of **38** bugs across ten different implementations. **26** were unique (*i.e.*, after removing those same bugs that affect multiple implementations) of which **15** were also found by SCALE and **11** represent new bugs that SCALE was unable to find. Comparatively, SCALE revealed only **22** unique bugs, of which **7** were not found by EyWA. *Hence, EyWA actually found more bugs than SCALE despite requiring little to no modeling effort.*

Comparing MESSI [72] with EyWA for BGP protocol implementation testing, EyWA replicated 2 bugs in prefix lists and route-filtering found by MESSI. More interestingly, we used EyWA to model two BGP features, confederation and route reflection, that were not in the manually written MESSI model and hence were not tested. Overall, EyWA found **5** unique bugs, of which **3** were new across all implementations.

Finally, our experiment with SMTP servers demonstrates how EyWA handles stateful protocols. Despite the relative simplicity of the server model, we found **1** new bug, and with minimal human effort.

4. 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 that identified a bug went beyond the bounds that we set for the EYWA-generated tests, for example involving multiple records in a zone file.

For BGP confederations, 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 arises when R tries 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. 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.

5 RELATED WORK

EYWA is related to several lines of prior work:

Automated testing. Work in automatically testing hardware and software is often categorized by their visibility into the source code. Black box testers [12, 29, 33, 62, 68, 80] for instance, generate random tests for implementations whose source code or specification is unavailable. Grey-box testers [7, 8, 11, 31, 34, 66, 77] use lightweight program instrumentation to obtain coverage feedback to guide future tests. White-box testers [10, 35–37] use heavyweight symbolic methods with access to source code to directly solve program path constraints. Each approach has limitations – black and grey-box approaches often get “stuck” on complex conditions and can result in low test coverage. Conversely, white-box approaches directly solve for inputs that meet complex conditions but (1) suffer from path explosion due to code complexity, (2) require access to source code, and (3) often require non-trivial source modification to be applied effectively [10, 36, 37, 63].

Another approach is model-based or specification-based testing [25, 45, 47, 53, 54, 75]. Model-based testing applies to black-box implementations by using a reference model or specification to generate tests. It sidesteps many of the aforementioned issues but requires users to manually craft a model, which is a non-trivial undertaking.

Protocol testing. Some testing work has directly targeted protocol implementations due to their importance and complexity [3, 6, 33, 38, 45, 48, 60, 63, 65, 72]. Our approach builds on model-based testing approaches for DNS [45] and BGP [72], which combine symbolic execution on the model with differential testing of implementations. However, rather than manually creating the protocol models, EYWA offloads the vast majority of the modeling task to LLMs through an API for modular and declarative model construction. This approach not only reduces the human burden significantly but

also makes EYWA useful for testing many protocols, while the prior tools each targets a single protocol.

Other approaches attempt to marry some software engineering testing techniques, such as grey-box fuzzing, with a learned protocol *state* (e.g., the protocol state machine). For instance, Pulsar [33] and AFLNet [65] combine black-box and grey-box testing respectively with protocol state machine learning from example traces. Rather than trying to “reverse engineer” a protocol from its usage, EYWA instead uses an LLM to directly encode a protocol’s state and logic.

LLM-based testing. Recent breakthroughs in natural language processing and program understanding with LLMs have led researchers to reconsider the possibility of automatically testing software using AI models [39, 46, 69, 71, 74, 76, 78, 81, 82, 84]. Most of these works directly ask LLMs to write unit tests for software or fuzz programs (i.e., mutate strings). 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 [56] enhanced the stateful protocol fuzzer AFLNet [65] using LLMs. Like EYWA, ChatAFL leverages LLM knowledge to improve testing, but it directly modifies the protocol messages that are used as 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 [83] or generating network configurations [41]. In more recent work [70], 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.

6 CONCLUSION

Despite the excitement, a thorny problem with using LLMs for critical applications is dealing with the errors they make. If LLMs are used to generate code, for instance, we must test [51] or verify [59, 73] the output, and iterate if needed. By contrast, in this paper we argue that if LLMs are used to *test* software, errors in the models and tests produced by the LLM can be finessed 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 distinct purposes: to construct protocol models, and also to generate state graphs to find efficient sequences to drive protocols to desired states. Our approach greatly reduces the burden of manually creating a protocol model, applies to black-box implementations, requires no source code changes, and produces tests that find deep functional correctness bugs.

Our testing framework Eywa, based on these principles, found 32 unique bugs in protocol implementations, including 15 new bugs previously undiscovered by manually constructed model-based testing tools. We could quickly build models for BGP features (confederations and route-reflectors) that had not been tested by earlier work (MESSI [72]) and also rapidly crafted an SMTP server model from scratch.

Our SMTP experience showed us that LLMs can also be used to drive protocols to specified states for testing, but we have only scratched the surface. We hope to explore this capability further to automatically test more complex stateful protocols like TCP and HTTP.

ACKNOWLEDGMENTS

We thank the developers of various open-source implementations for their feedback on our bug reports. The authors, Rajdeep Mondal, Rathin Singha, Todd Millstein and George Varghese, were partially supported by NSF Grant 2402958.

REFERENCES

- [1] 2024. OpenSMTPD. <https://www.opensmtpd.org/>. (2024).
- [2] aiosmtpd community. 2024. aiosmtpd - An asyncio based SMTP server. <https://aiosmtpd.aio-libs.org/en/latest/>. (2024).
- [3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.
- [4] Scott Berinato. 2023. All systems down. <https://www.computerworld.com/article/2581420/all-systems-down.html>. (2023). Accessed: 2023-9-29.
- [5] Brandon L Black and Community. 2023. GDNDS. <https://gdnsd.org/>. (2023).
Eywa commit: <https://github.com/gdnsd/gdnsd/tree/877e15cf55593fa618d2009027e928d5f52da775>.
- [6] Gregor V Bochmann and Alexandre Petrenko. 1994. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. 109–124.
- [7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.
- [8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
- [9] Chris Brook. 2023. 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. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
- [11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
- [12] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.
- [13] CoreDNS community. 2023. CoreDNS. <https://coredns.io/>. (2023).
Eywa commit: <https://github.com/coredns/coredns/tree/45923b6e12a2eabaf55d7380e6df4e7354a1207>
SCALE commit: <https://github.com/coredns/coredns/tree/6edc8fe7f6c2f57844c8ee7f7f5deef71085ebe8>.
- [14] Dnspython Community. 2023. Dnspython. <https://dnspython.readthedocs.io/en/latest/index.html>. (2023).
- [15] ExaBGP community. 2024. ExaBGP. <https://github.com/Exa-Networks/exabgp>. (2024).
- [16] FRR community. 2024. The FRRouting Protocol Suite. <https://frrouting.org/>. (2024).
Version: <https://github.com/FRRouting/frr/releases/tag/frr-10.1.2>.
- [17] GoBGP community. 2024. GoBGP. <https://github.com/osrg/gobgp>. (2024).
- [18] PowerDNS Community. 2023. PowerDNS. <https://www.powerdns.com/>. (2023).
Eywa commit: <https://github.com/PowerDNS/pdns/tree/8314f12e92a8b75e33438bc7c16c6430028fbef9>
SCALE commit: <https://github.com/PowerDNS/pdns/tree/a03aaad7554483ee6efe72a81eda00a9d1a94fe5>.

- [19] Internet Systems Consortium. 2023. 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/dbcf683c1a57f4987e329fca183cb39d20ca3a4>.
- [20] curl. 2023. FTP Server Response Buffer Overflow. <https://curl.se/docs/CVE-2000-0973.html>. (2023). Accessed: 2023-9-29.
- [21] curl. 2023. FTP shutdown response buffer overflow. <https://curl.se/docs/CVE-2018-1000300.html>. (2023). Accessed: 2023-9-29.
- [22] cyberstanc. 2023. 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.
- [23] CZ.NIC. 2023. 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/89aaeb729a0856fefaed11c114ebb8a5a3f4ed2>.
- [24] Python developer community. 2024. SMTPD Python library. <https://docs.python.org/3.10/library/smtpd.html>. (2024).
- [25] 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. 2024. 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)*. Association for Computing Machinery, New York, NY, USA, 351–357. <https://doi.org/10.1145/3663529.3663854>
- [26] Jim Duffy. 2023. BGP bug bites Juniper software. <https://www.networkworld.com/article/2289950/bgp-bug-bites-juniper-software.html>. (2023). Accessed: 2023-9-29.
- [27] Tushar Subhra Dutta. 2023. BGP Error Handling Flaw Leads to Prolonged Network Outage. <https://cybersecuritynews.com/bgp-error-handling-flaw/>. (2023). Accessed: 2023-9-29.
- [28] EURid.eu. 2023. YADIFA. <https://www.yadifa.eu/>. (2023). Eywa commit: <https://github.com/yadifa/yadifa/tree/9bb6facead9e7ba222962b2980f85fa6ba02e465> SCALE commit: <https://github.com/yadifa/yadifa/tree/dc5bed2fb8ec204af9b65eeb91934c2c85098cbb>.
- [29] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 337–350.
- [30] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, USA, 469–483.
- [31] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.
- [32] Benjamin Fry and Community. 2023. 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>.
- [33] Hugo Gascon, Christian Wressneger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. 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*. Springer, 330–347.
- [34] github. 2023. google/AFL. <https://github.com/google/AFL>. (2023). Accessed: 2023-9-29.
- [35] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.
- [36] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.
- [37] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.
- [38] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns* 10, 8 (2010), 239.
- [39] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting Greybox Fuzzing with Generative AI. *arXiv preprint arXiv:2306.06782* (2023).
- [40] ipSpace. 2023. 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.
- [41] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. 2021. Hey, lumi! using natural language for {intent-based} network management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 625–639.
- [42] Van Jacobson. 1988. Congestion avoidance and control. *ACM SIGCOMM computer communication review* 18, 4 (1988), 314–329.
- [43] William Johansson, Martin Svensson, Ulf E Larson, Magnus Almgren, and Vincenzo Gulisano. 2014. T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 323–332.
- [44] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 310–328. <https://doi.org/10.1145/3387514.3405871>
- [45] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2022. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 307–323. <https://www.usenix.org/conference/nsdi22/presentation/kakarla>
- [46] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Efficient Mutation Testing via Pre-Trained Language Models. *arXiv preprint arXiv:2301.03543* (2023).
- [47] Sarfraz Khurshid and Darko Marinov. 2004. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering* 11 (2004), 403–434.
- [48] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. 2010. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*. IEEE, 202–208.

- [49] NLnet Labs. 2023. 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>.
- [50] Twisted Matrix Labs. 2023. TwistedNames. <https://twisted.org/>. (2023). Eywa commit: <https://github.com/twisted/twisted/tree/157cd8e659705940e895d321339d467e76ae9d0a>.
- [51] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. 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)*. Curran Associates Inc., Red Hook, NY, USA, Article 943, 15 pages.
- [52] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. 2023. A Formal Framework for End-to-End DNS Resolution. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 932–949. <https://doi.org/10.1145/3603269.3604870>
- [53] Kenneth L McMillan and Lenore D Zuck. 2019. Compositional testing of internet protocols. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 161–174.
- [54] Kenneth L. McMillan and Lenore D. Zuck. 2019. Formal Specification and Testing of QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 227–240. <https://doi.org/10.1145/3341302.3342087>
- [55] Shikhar Mehrotra. 2023. 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.
- [56] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. NDSS.
- [57] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239 (March 2014), 2.
- [58] Microsoft. 2023. Azure OpenAI Service. <https://azure.microsoft.com/en-us/products/ai-services/openai-service>. (2023).
- [59] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to Synthesize Correct Router Configurations?. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 189–195. <https://doi.org/10.1145/3626111.3628194>
- [60] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.
- [61] Lily Hay Newman. 2023. Decades-Old Code Is Putting Millions of Critical Devices at Risk. <https://cybersecuritynews.com/bgp-error-handling-flaw/>. (2023). Accessed: 2023-9-29.
- [62] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
- [63] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. Analyzing protocol implementations for interoperability. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 485–498.
- [64] Ken Pfeil. 2023. Buffer Overflow in Digital Mapping System's POP3 Server. <https://www.itprotoday.com/email-and-calendar/buffer-overflow-digital-mapping-systems-pop3-server>. (2023). Accessed: 2023-9-29.
- [65] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.
- [66] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.
- [67] Erik Romijn. 2023. 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.
- [68] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Small-check and lazy smallcheck: automatic exhaustive testing for small values. *Acm sigplan notices* 44, 2 (2008), 37–48.
- [69] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).
- [70] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 41–47.
- [71] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418* (2023).
- [72] Rathin Singha, Rajdeep Mondal, Ryan Beckett, Siva Kesava Reddy Kakarla, Todd Millstein, and George Varghese. 2024. {MESSI}: Behavioral Testing of {BGP} Implementations. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1009–1023.
- [73] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 134–155. https://doi.org/10.1007/978-3-031-65112-0_7
- [74] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
- [75] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.
- [76] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *arXiv preprint arXiv:2307.04346* (2023).
- [77] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superior: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.
- [78] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221* (2023).
- [79] Wikipedia. 2023. 2022 Rogers Communications outage. https://en.wikipedia.org/wiki/2022_Rogers_Communications_outage. (2023). Accessed: 2023-9-29.
- [80] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 511–522.
- [81] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. *arXiv preprint arXiv:2308.04748* (2023).
- [82] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang.

2021. 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*. 435–450.
- [83] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 272–286.
- [84] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).
- [85] Shreyas Zare and Community. 2023. 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 9. 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.

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 counted the number of tests generated at for each k averaged over 10 runs. We plot this count vs. k in Figure 10 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 11 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

```
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 9: Eywa’s minimal regular expression implementation that is amenable to symbolic execution.

stanza. Each of the modules is first defined using the prompting technique shown in Figure 1a. We then specify the module dependencies using Eywa’s built-in graph API. We first

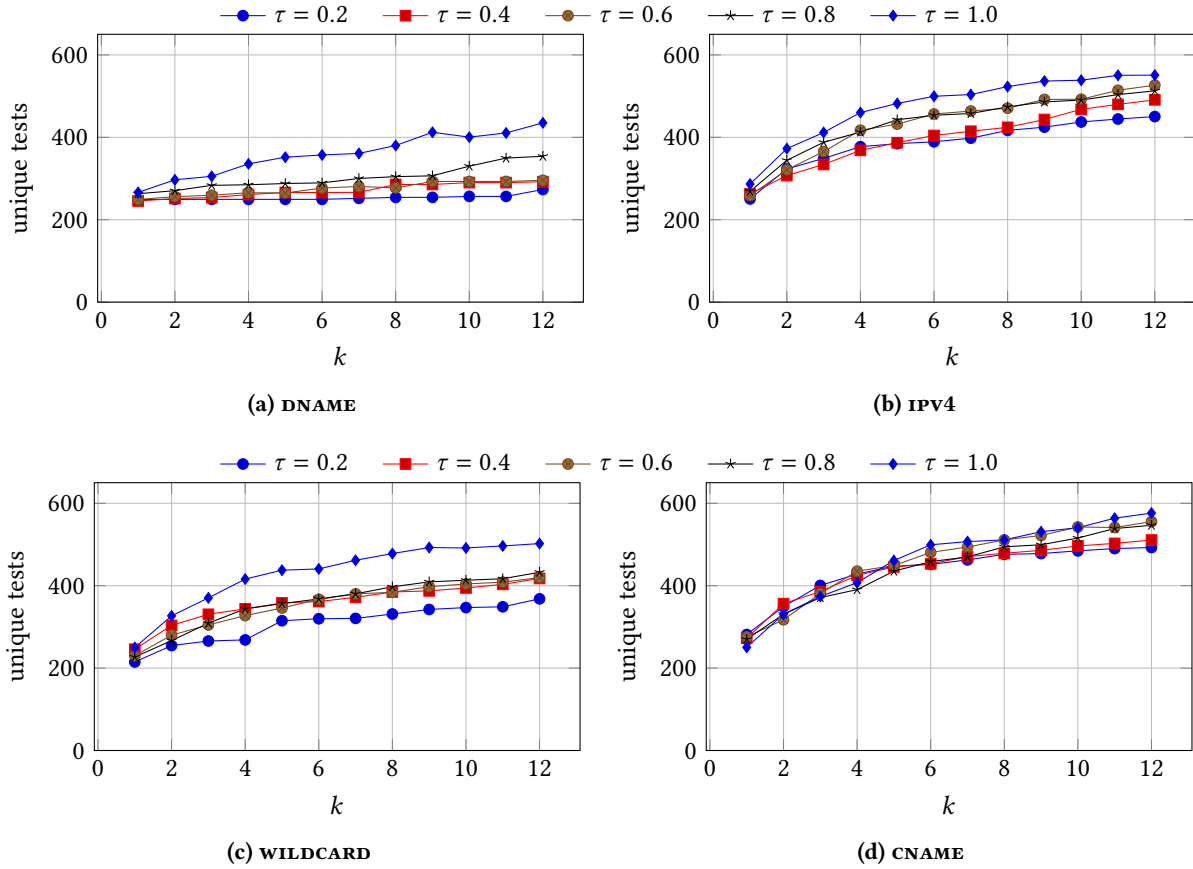


Figure 10: Hyperparameter analysis for DNS models. The plots show how the total number of unique tests vary as we increase the number of runs for different models.

```

# 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(checkValidInputs,isMatchRouteMapStanza)
final_model = g.Synthesize() # synthesize final model

```

Figure 11: EYWA code for building a dependency graph for the BGP RMAP-PL model in Table 2

initialize an empty graph and then add in all the dependencies using the CallEdge and Pipe methods. In this example, we use a Pipe between the functions checkValidInputs and isMatchRouteMapStanza, as the former is used to enforce constraints on the inputs to the latter. The rest of the edges are of the type CallEdge, as in those cases the result from one function is being directly used in another. We need to define multiple CallEdges here because there are hierarchical dependencies. For instance, the function isMatchRouteMapStanza depends on isMatchPrefixListEntry which in turn calls the function prefixLengthToSubnetMask. Also, the reader may observe that unlike the other modules, checkValidInputs is dependent on two modules, namely isValidRoute and isValidPrefixList. Both of them must therefore be specified in a list while using the CallEdge method. Following this, we can synthesize the final model using the Synthesize utility. For synthesizing individual modules with incoming CallEdges, the prompt includes a description of all the dependency functions along with their C prototypes. Figure 12 shows the prompt generated due to the CallEdge from the

```

#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 12: Prompt for the prefixLengthToSubnetMask to isMatchPrefixListEntry dependency in Figure 11.

function named prefixLengthToSubnetMask to the function isMatchPrefixListEntry, when we synthesize a model for the latter. This is crucial because it helps to make the LLM aware of all the utility functions available to the current module to make use of. Finally, Eywa inserts all the functions

into the final program in the correct order according to their topological ordering.

D SYSTEM PROMPT

Figure 13 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 code **for** 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>
#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) {
    return x + 1
}
```

Figure 13: The system prompt which we used for all LLM calls.