Oracle-based Protocol Testing with Eywa

Siva Kesava Reddy Kakarla
Microsoft Research
sivakakarla@microsoft.com

Ryan Beckett
Microsoft
ryan.beckett@microsoft.com

ABSTRACT

We present *oracle-based testing* a new technique for automatic black-box testing of network protocol implementations. Oracle-based testing leverages recent advances in LLMs to build rich models of intended protocol behaviour from knowledge embedded in RFCs, blogs, forums, and other natural language sources. From these models it systematically derives exhaustive test cases using symbolic program execution. We realize oracle-based testing through Eywa, a novel protocol testing framework implemented in Python. To demonstrate Eywa's effectiveness, we show its use through an extensive case study of the DNS protocol. Despite requiring almost no effort, applying Eywa to the DNS resulting in the discovery of 26 unique bugs across ten widely used DNS implementations, including 11 new bugs that were previously undiscovered despite elaborate prior testing with manually-crafted models.

1 INTRODUCTION

Networked systems depend on the correct and performant operation of dozens or even hundreds of protocols from the physical layer to the application layer. Despite pervasive standardization, protocol hardware and software implementations still frequently suffer from bugs due to coding mistakes, misinterpretations of specifications, unsound optimizations, unforeseen corner cases, poor data structure choices, and more. Bugs in protocol implementations lead to security vulnerabilities [8, 16–18, 38, 54, 57], network outages and service disruptions [2, 20, 21, 33, 48, 60, 70], or even poor network performance [35]. For example, in 2022 a bug in Akamai's DNS software caused a global outage for nearly 30,000 sites [48], and recently, mishandling of a corrupted BGP attribute in router software caused invalid routes to be propagated throughout the Internet before shutting down BGP sessions in remote networks [21].

To ensure the correctness of protocol implementations, practitioners today rely primarily on manual testing via unit and regression test suites [38]. While convenient, manual testing is often incomplete as humans miss subtle corner cases that can lead to problematic behavior. Protocols are often *complex*, with numerous corner cases that are difficult to implement correctly and efficiently, or whose standards specifications may even be ambiguous.

As an alternative to manual testing, the software engineering research community has long championed automatic testing via fuzzing (semi-random mutation) [7, 27] and symbolic execution (exhaustive search by solving path conditions) [9, 29, 30, 42] that generate tests systematically. Unfortunately, these techniques suffer from known limitations. Fuzzing is often unable to navigate complex conditions in code [5], and symbolic execution requires access to & modification of source code (often unavailable for protocol implementations) and also suffers from the path explosion problem for large code bases [30].

To circumvent these limitations, researchers have proposed model-based testing (MBT) to automatically generate tests for protocols. MBT also uses symbolic execution to systematically derive tests, but does so on a high-level model of the protocol's behavior rather than low-level source code. This approach is surprisingly effective, and researchers have used MBT to find numerous bugs in implementations of QUIC [46, 47], DNS [38, 45], and various LTE [36] protocols.

Since hand-written protocol models are typically orders of magnitude simpler than program source code, MBT helps ameliorate the infamous path explosion problem, and also applies to closed-source (black-box) protocol implementations. In exchange, MBT requires substantial effort from users to arduously build the "model" of the protocol under test. For instance SCALE [38], which applied MBT to the DNS protocol required first formalizing the DNS semantics mathematically from RFCs [37], and then later building an executable model of the formalism by hand [38] in over 500 lines of code.

In this work, we ask the question: *"can we get the benefits of MBT without the burden of manually crafting protocol models?"* We answer this question affirmatively by observing that there is a vast body of protocol knowledge defined in RFCs & standards, networking forums & blogs, as well as other online resources and documents. Recent advances in large language models (LLMs) have made it possible to extract knowledge stored in natural language to use for automated testing. However, using LLMs to test network protocols in practice requires overcoming several nontrivial challenges.

Challenge #1 (exhaustive testing): While it is possible to directly ask an LLM to produce tests for a given network protocol, doing so provides no guarantees of coverage or exhaustiveness of the tests. How can we ensure we generate test cases for every protocol corner case?

Challenge #2 (valid inputs): Protocols have highly structured inputs (*e.g.*, a TCP header, a DNS zone, or a BGP route) and generating an invalid input can render a test useless. Moreover, such inputs may have complex conditions (*e.g.*, a DNS zone has exactly one SOA record whose domain name may not be used by other records). Coercing LLMs into given examples in the correct format and of the correct type is not always straightforward. Thus, it is important to be able to enforce that LLM-generated examples conform to potentially complex input constraints.

Challenge #3 (modular testing): Many protocols are highly complex and comprise multiple components. Asking an LLM to give generate end-to-end protocol tests is unrealistic in many cases. Instead, one needs a way to decompose such complex protocols into smaller modular components that are more amenable to LLM testing.

In this paper we introduce a new approach to automatically test network protocols that we call *oracle-based testing* that addresses each of these challenges. Like MBT, oracle-based testing builds a model of a protocol and uses symbolic execution to exhaustively generate tests for the protocol. Unlike MBT, oracle-based testing uses an LLM to construct the protocol model, removing the human from the loop.

The tester (user) provides only a high-level description of a protocol component (*i.e.*, what part of the protocol to test) as well as its input and output types. The oracle then gives back a semi-exhaustive set of test cases for that component. Control over the protocol component and type signature to test allows users to scope their tests to what the LLM can reasonably handle and facilitates *modular testing (#3)*.

To generate *exhaustive tests (#1)*, rather than ask LLMs to directly test protocols, we instead use LLMs for a task at which they excel — writing code. Specifically, we task the LLM with building a *partial* reference protocol implementation. We then use existing symbolic execution tools for a task at which they excel — automatically generating test inputs that systematically explore every path through this model.

Finally, because the LLM generates code rather than producing arbitrary text output, the test generated are guaranteed to be of the correct type (*valid inputs (#2)*). To enforce additional validity constraints on inputs (*e.g.*, the format of a string or value of a protocol field), we provide novel abstractions to augment the LLM with additional *symbolic constraints*, which are then enforced by the symbolic execution engine for all LLM model tests.

We realize our vision of oracle-based testing in a tool called Eywa, which is implemented as a Python library. Given a description of a (component of a) protocol, Eywa automatically constructs a model of that component using an LLM as executable C code. Eywa simultaneously compiles a *symbolic test harness* to integrate this model with the Klee [9] symbolic execution engine. By invoking Klee Eywa obtains

an exhaustive set of test cases and translates them to Python for the library user. The user can then apply these tests to real protocol implementations in any way they want. The architecture in shown in Figure 3.

We evaluate the effectiveness of Eywa through an extensive case study of the DNS protocol. Compared to prior model-based testing approaches for DNS that required months of effort to craft DNS models that accurately encode the RFC intent [38], Eywa constructed similar models in just a few tens of lines of code and with little to no prior protocol knowledge. Using Eywa, we supplied eight prompts resulting in the generation of roughly 100K test cases. These tests revealed **26** unique bugs across the ten popular DNS implementations evaluated. Of these bugs, **11** were were previously undiscovered despite the extensive testing by prior work.

Summary. To summarize, our contributions are:

- We introduce oracle-based testing, a new approach to network protocol testing that combines LLMs with traditional symbolic-execution-based test generation.
- We develop Eywa, a library that implements oracle-based testing, providing novel abstractions for describing protocol components and their constraints.
- We demonstrate how to compile Eywa specifications by generating specialized LLM prompts and combining the resulting models with a *symbolic test harness*.
- We showcase the utility of Eywa through an extensive case study of the DNS protocol. Eywa found 26 unique bugs across ten popular DNS implementations, including 11 previously undiscovered bugs.

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 [38] 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 (Kubernetes). To achieve these results however, SCALE required extensive manual mathematical formulation of the DNS RFCs [37] and later arduous manual construction of an executable model [38]. We give an overview of Eywa in the context of DNS testing and show how it achieves results similar to SCALE, including finding 11 new bugs that SCALE missed, all with minimal user effort.

2.1 Testing the DNS with Eywa

To test a protocol, users must describe each component of that protocol that they wish to model using Eywa's library abstractions. Specifically, users create models by defining

```
domain_name = eywa.String(maxsize=3) # Define a domain name type for DNS.
label_re = re.choice(re.text('*'), re.chars('a', 'z')) # Create a regex constraint for domain names.
valid_re = re.seq(label_re, re.star(re.seq(re.text('.'), label_re))) # Format "label.label.label".

query = eywa.Arg("dname1", domain_name, "The DNS query domain name.") # Define function args.
dname = eywa.Arg("dname2", domain_name, "The DNAME record domain name.")
result = eywa.Arg("result", eywa.Bool(), "If the DNS DNAME domain name applies to the query name.")
dname_applies = eywa.Func( # Define the model as a function with a description.
    "dname_applies",
    "a function that checks if a DNAME (provides redirection from a part of the DNS name
    tree to another part of the DNS name tree) DNS record name applies to a DNS domain name query.",
    [query, dname, result],
    precondition=(query.matches(valid_re) & dname.matches(valid_re)))

inputs = eywa.generate_tests(dname_applies, timeout="300s") # Generate tests from model.
```

(a) Model of DNS DNAME resolution built in Eywa.

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <klee/klee.h>
```

```
bool dname_applies(char* dname1, char* dname2) {
    ...
}
```

①

(e) Protocol implementations.

```
int main() {
    char x0[4];
    char x1; char x2; char x3;
    char x5[4];
    char x6; char x7; char x8;
    bool x10;
    klee_make_symbolic(&x1, sizeof(x1), "x1");
    ...
    klee_make_symbolic(&x10, sizeof(x10), "x10");
    x0[0] = x1; x0[1] = x2; x0[2] = x3; x0[3] = '\0';
    x5[0] = x6; x5[1] = x7; x5[2] = x8; x5[3] = '\0';
    Regex r = ...;
    klee_assume(match(&r, x0));
    klee_assume(match(&r, x5));
    dname_applies(x0, x5);
}
```

②

(d) Differential testing setup.

```
["a.*", "*", False]
["c.a", "a", True]
["c", "a", False]
```

④

③

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

(b) Implementation and symbolic harness generated by Eywa.

(c) Klee invocation.

Figure 1: Example of how Eywa generates tests. It uses the LLM to implement the function logic ①. It then compiles a symbolic test harness for Klee ② and invokes Klee to perform symbolic execution ③ and generate test cases. Tests generated from the Eywa runtime, are then executed for every implementation ④.

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 DNAME record type feature used in DNS to redirect queries across domains.

In the Python code, the first line defines a DNS domain name type (domain_name) as a string, and limits the size of the string to 3 characters to limit the size of tests. The code defines the model as a function dname_applies that tests whether a DNS query's domain name applies to a DNAME record's domain name. The three definitions (query, dname, result) give the function arguments and return value respectively and include natural language descriptions of their meaning. The user also provides a description of the function's purpose as well as any constraints on the arguments (precondition). The last line eywa.generate_tests takes a function definition as well as a timeout (300 seconds) and returns a set of test cases in the form of a list of values for each function argument (*e.g.*, ["a.*", "*", False]) where "a.*" is the query domain name, "*" is the DNAME record domain name, and False is the return value.

How it works. So how does EYWA come up with these tests? From this function definition, EYWA first prompts the LLM to implement the function description as a protocol model in C code (①). It takes into account the input and output argument type definitions and descriptions. EYWA also simultaneously generates a specialized symbolic test harness, also in C, to initialize symbolic inputs for the model the LLM produces (②). In the example, it creates two zero-terminated symbolic strings of up to 3 characters each to represent the query and DNAME domain names and passes them to the LLM implementation (dname_applies).

Next, EYWA assembles a complete program and invokes Klee (③) in an isolated Docker container to perform symbolic execution and enumerate all values for symbolic inputs to the model that result in a different execution path (set of evaluations of conditional branches in the code). These results are then translated back from C into Python and passed back to the user in the (inputs) variable.

Given the returned inputs, the user can then proceed to test them against actual protocol implementations in any way they wish. For DNS, we use differential testing [38] to craft DNS zone files and queries from the test inputs and then compare each DNS nameservers response to find bugs (④). This is the same approach taken by SCALE [38].

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 (simplified) implementation returned by the LLM is shown in Figure 2. This implementation

```c
bool dname_applies(char* dname1, char* dname2) {
    int len1 = strlen(dname1);
    int len2 = strlen(dname2);

    // If the DNAME domain name is longer than
    // the domain name, it cannot be a match.
    if (len2 > len1) {
        return false;
    }

    // Compare the domain names in reverse order.
    for (int i = 1; i <= len2; i++) {
        if (dname1[len1 - i] != dname2[len2 - i]) {
            return false;
        }
    }

    // If the DNAME domain name is equal to the
    // domain name, it is a match.
    if (len2 == len1) {
        return true;
    }                           Model bug!

    // If the character before the DNAME
    // domain name is a dot, it is a match.
    if (dname1[len1 - len2 - 1] == '.') {
        return true;
    }

    return false;
}
```

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

actually has two issues. First, it never checks if the domain names are valid. As a result, Klee will find only unhelpful inputs such as ["a..", "..", True]. The query "a.." for instance, is not a valid domain name according to the DNS RFCs since dots may not be separated by empty labels [52]. While one could ask the LLM to additionally check for valid inputs, doing so magnifies the complexity of the task that the LLM must perform.

To solve this problem, EYWA allows users to attach constraints to function arguments that force the symbolic execution engine to only consider the subset of inputs to the model that meet the constraints. In Figure 1 we define a regular expression valid_re that specifies a domain name as a dot-separated sequence of string labels, where each label is either alphanumeric or a special character "*". When constructing the symbolic test harness, EYWA encodes these constraints in

C thereby forcing Klee to solve for inputs that satisfy them (*e.g.*, `klee_assume(match(&r, x0))`).

The second issue comes from a logical error in the model itself. Consider again the model implementation in Figure 2. The highlighted region of code is in fact incorrect – a DNAME record can only apply to the DNS query if it is shorter than the query. In many cases like this however, such mistakes may actually be harmless or even helpful. In this case, the logical error will actually result in generating an extra test for the case where the lengths of the query and DNAME record domain names are equal, which is actually a very useful corner case to test. While the LLM-generated model may have the wrong answer, the test can still be used to find bugs in the real implementations, *e.g.*, by comparing their behavior to other implementations (*e.g.*, see §5.5).

2.3 Finding implementation bugs

Oracle-based testing relies on the same principles that make model-based testing effective – the use of simple models combined with symbolic execution enables exhaustive test generation for a subset of a protocol's functionality. However, compared to prior work that requires substantial manual effort to first formalize and then build these models, we instead use LLMs to automatically extract and build models for us (*e.g.*, in just a few lines of code as in Figure 1).

One might wonder if the quality of the model, and hence the test cases, will suffer if the LLM produces a poor model. However, even for this simple example, Eywa actually found a new bug [39] in Knot authoritative nameserver implementation [19] with the example shown previously. In the example, Eywa generated the DNAME record using which we constructed the following zone file:

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

The DNS query Eywa generated was ⟨a. ∗ .test., CNAME⟩. The DNAME record is applied to rewrite any queries ending with ∗.test. to end with a.a.test., so the the query should be rewritten using the DNAME record by synthesizing a CNAME record. The nameserver implementations are expected to return the DNAME record, along with the synthesized CNAME record, in the answer section of the response as follows:

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

Knot was synthesizing the CNAME correctly, but instead of returning the DNAME record from the zone file, it was creating a new DNAME record with the query name as:

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

This is incorrect behavior as a resolver checking whether the rewrite of the query was legitimate will fail as DNAME record in the Knot response will not apply to the query name as DNAME match is possible only with sub-domains. After discovering the bug, we filed the issue on the Knot Gitlab source code, and the developers responded positively fixing the issue in a week.

2.4 Limitations of the approach

For oracle-based testing 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. For new protocols deployed beyond the LLMs training cutoff date or for proprietary protocols whose specifications are not publicly available, oracle-based testing will be ineffective.

In such cases, users can still use oracle-based testing if they can augment the LLM with protocol-specific documentation. For instance, by fine-tuning the LLM or by including context, *e.g.*, protocol-specific documents and specifications, within Eywa's prompts. This work raises no ethical concerns.

3 THE EYWA LIBRARY AND RUNTIME

In this section we describe Eywa's design and implementation. In §5, we discuss a case study of Eywa's use to find bugs in the DNS protocol and to evaluate Eywa's performance and effectiveness.

3.1 Eywa Architecture

Eywa's architecture comprises several interacting components as illustrated in Figure 3. To start, the user (top) uses the Eywa library to define a protocol model (function) with its arguments, result, and any validity constraints similar to the example from Figure 1. From these inputs, Eywa then invokes its *Symbolic Compiler* and *Prompt Generator*.

The *Symbolic Compiler* is responsible for taking the function definition and validity constraints and generating the *Symbolic Harness*. The *Symbolic Harness* initializes each of the symbolic function parameters by constructing a symbolic value of the appropriate types, translates the function pre-conditions/constraints into Klee assumptions that are added as path constraints to the symbolic executor, and calls the model (function) produced by the LLM.

The *Prompt Generator* builds an LLM prompt from the function definition and frames the prompt as a completion problem. The prompt includes translated C types for each of the user-defined types as well as a function documentation
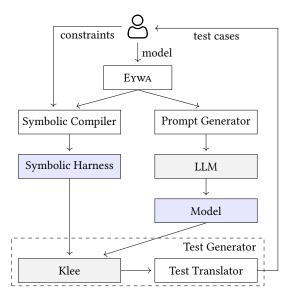
Figure 3: Architecture of Eywa. White boxes are Eywa components, blue boxes are generated code, and grey boxes represent 3rd party components.

string to assist the LLM. It additionally generates a system prompt to guide the behavior of the LLM and produce code that is amenable to symbolic execution. The result is the *Model* as C code that the LLM produces.

Finally, the *Test Generator* takes the resulting symbolic harness and model and combines them with the appropriate preamble. It invokes the C compiler and Klee commands to instrument the code for symbolic execution and then runs Klee on the resulting LLVM bytecode. Finally, it extracts the generated test cases from Klee and passes them through the *Test Translator*, which converts the values back into Python values for use by the user.

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 TCP, one could create a function that takes (1) a TCP header representing a received packet, and (2) the current state of the TCP protocol for the host receiving the packet, and returns (3) the new TCP packet to send and (4) the new state. A summary of the main modeling abstractions are show 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()),

Example Feature	Description
`eywa.Bool()`	A boolean value
`eywa.Char()`	A character value
`eywa.String(maxsize=5)`	A string of size
`eywa.Int(bits=5)`	A 5-bit unsigned int
`eywa.Enum("name", ["A", "NS"])`	An enum value
`eywa.Array(Bool(), 3)`	An array of 3 bools
`eywa.Struct("name", dst=Int())`	A struct with 1 field
`eywa.Alias("result", Bool())`	A type alias
`eywa.Arg("name", Int(), "desc")`	A function arg
`eywa.Func("name", "desc", args)`	A function.

Figure 4: Summary of Eywa types and abstractions.

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, 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.

Finally, users may optionally provide constraints over the function arguments. Eywa supports many kinds of constraints, including arithmetic, (in)equality, and comparison constraints for integers, regular expression constraints for strings, and more. Eywa overloads common python operators to allow for natural constraint descriptions (see §3.4).

3.3　Eywa Prompt Generator

Given a model definition, Eywa invokes its *Prompt Generator* to build an LLM prompt that will lead the LLM to implement the model. Eywa builds two prompts: a system prompt to guide the behavior of the LLM, and a user prompt that frames the implementation task as a completion problem.

Consider the example model defined using the Eywa library in Figure 5. The example implements the main DNS query lookup functionality that takes a user zone (collection of DNS resource records) and a DNS query, and generates a DNS response. The model defines several new data types to assist with this implementation, including DNS record type, record, zone, query, and response data types.

User Prompt. To create the user prompt, Eywa translates each of the user-defined model types into C data structures for the LLM to assume (*e.g.*, **struct**, **enum**, **typedef**). It then translates the function signature into C using these definitions and adds a function documentation string based on the user descriptions of the function as well as each of the parameters. We show the resulting prompt that is generated for Figure 5 in Figure 6. From this prompt, the LLM will naturally predict (complete) the rest of the function.

```
domain_name = eywa.String(3)
record_type = eywa.Enum("RecordType", ["A", "AAAA", "NS", "TXT", "CNAME", "DNAME", "SOA"])
record = eywa.Struct("ResourceRecord", record_type=record_type, dname=domain_name, rdata=eywa.String(3))
zone = eywa.Alias("Zone", eywa.Array(record, 2))
query = eywa.Struct("DNSQuery", record_type=record_type, dname=domain_name)
response = eywa.Alias("Response", eywa.String(20))

a1 = eywa.Arg("zone", zone, "The zone as a collection of resource records.")
a2 = eywa.Arg("query", query, "The input query to lookup which has the domain name and record type")
a3 = eywa.Arg("response", response, "The response to the DNS query.")

dns_query_lookup = eywa.Func(
    "dns_lookup",
    "a function that implements a DNS query lookup on a given zone. Assume there is no cache and
    there is only the input zone file to answer the query. Assume the zone file is valid and all
    the record domain names in the zone are not relative domain names. The zone file should satisfy
    all the DNS RFC constraints and should be valid. The input query has a domain name and record
    type. The function should have CNAME, DNAME, NS and wildcard cases. It returns the DNS response.",
    [a1, a2, a3])
```

Figure 5: Example Eywa model for DNS lookup that takes a Zone and DNS query and returns a DNS response.

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 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.

Adding these requirements helps ensure that the LLM generates a valid program that compiles in almost every case.

3.4 Eywa Symbolic Compiler

The *Symbolic Compiler* is responsible for generating the *Symbolic Harness* like the one shown in Figure 1. We use the `klee_make_symbolic` command to build symbolic inputs for every base type (`bool`, `char`, `int`, `enum` *etc.*) and then construct more complex types from these base types.

Eywa constructs arrays by declaring a new array of the corresponding size, creating each symbolic element and then assigning each element to the array. Eywa similarly initializes structs by creating a symbolic value for each field and

then assigning those fields to the new struct. Nested symbolic types like arrays of structs are constructed recursively.

Enforcing constraints. To enforce user constraints, Eywa translates them into C expressions that evaluate to boolean values and allows Klee to solve them. The translation is straightforward – for instance, a Python constraint such as: `(a1 > 3) & a2.matches(re.star(re.chars("a", "z")))` will generate the harness code:

```
Regex r1; r1.lo = 'a'; r1.hi = 'z';
Regex r2; r2.op = STAR; r2.left = r1;
klee_assume(a1 > 3);
klee_assume(match(&r2, &a2));
```

For regular expressions, Eywa constructs a regular expression in C and calls a custom `match` function to check if the string matches the regular expression. The `match` function is a minimal regular expression matching implementation that we have written by hand in around 60 lines of code (**??**) and that is amenable to symbolic execution. Regular expression constraints are general and users can use them to enforce various conditions such as the length of the string, any string prefix or suffix constraint and more. Klee will encode the regex match condition and other conditions symbolically and solve them as part of its path exploration.

3.5 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

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

typedef String Response;

typedef enum {
    A, AAAA, NS, TXT, CNAME, DNAME, SOA
} RecordType;

typedef struct {
    RecordType record_type;
    String dname;
    String rdata;
} ResourceRecord;

typedef ResourceRecord Zone[3];

typedef struct {
    RecordType record_type;
    String dname;
} DNSQuery;

// a function that implements a DNS query lookup
// on a given zone. Assume there is no cache and
// there is only the input zone file to answer
// the query. Assume the zone file is valid and
// all the record domain names in the zone are
// not relative domain names. The zone file
// should satisfy all the DNS RFC constraints
// and should be valid. The input query has a
// domain name and record type. The function
// should have CNAME, DNAME, NS and wildcard
// cases. It returns the DNS response.
//
// Parameters:
//    zone: The zone as a collection of resource
//          records which can have CNAME, DNAME,
//          NS or wildcard records.
//    query: The input query to lookup which has
//           the domain name and record type
//
// Return Value:
//    The response to the DNS query.
Response dns_lookup(Zone zone, DNSQuery query) {
```

Figure 6: LLM prompt for the example in Figure 5

the *Symbolic Harness* and *Model* 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 back to the user as feedback. Users can use this feedback to update their function definitions 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 C value. The *Test Generator* then serializes these values back to the Eywa library, which walks over the base values and reconstructs any richer types (*e.g.*, struct, array) from these values. The resulting Python type will correspond to the declared type. For an array, Eywa returns a Python list, and for a struct it returns a Python map from each field name to its corresponding value. Eywa also captures the output value of the model and returns a list of values, one for each function argument (and output).

4 IMPLEMENTATION

The Eywa library is implemented in roughly 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 [51]. 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. The library invokes the LLM k times to generate k implementations and then aggregates the resulting test cases across in order to give the LLM more chances to produce a "good" implementation and improve test coverage. For our experiments, we use a $k = 10$ and temperature $\tau = 0.6$ (see §5.6). 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 CASE STUDY: DNS

We study the effectiveness of Eywa through a major case study with the DNS protocol. We choose DNS because it is a complex protocol with numerous implementations as well as rich configuration, policies, and semantics. DNS was also already the target of prior manual model-based testing tools, which gives us a baseline against which we can understand the effectiveness of oracle-based testing with Eywa.

5.1 Test setup

Using Eywa, we tested ten of the most widely used DNS implementations: BIND [15], COREDNS [12], GDNSD [3], HICKORY [25], KNOT [19], POWERDNS [14], TECHNITIUM [76], YADIFA [22], and TWISTED NAMES [44]. Of these implementations, seven were also tested by SCALE [38] (all except GDNSD, TECHNITIUM, and TWISTED NAMES). For those seven implementations tested by SCALE, 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. SCALE used the implementations' code as of October 2020 and for the latest versions, we took the code as of October 2023[1].

To test each implementation we follow the differential testing design described in Figure 1. We created scripts to initialize a working Docker [50] container for each DNS implementation and version pair. Each container serves a single zone file as an authoritative zone, and we used the dnspython [13] 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) and group the equivalent implementations.

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.

To effectively test different DNS implementations using the tests generated by Klee, it was necessary to create well-formed zone files. This involved adding essential records such as SOA and NS records. For instance, in the case of the CNAME model, each test includes only a resource record and a query. Hence, we craft a zone file for a domain named "test" and incorporate SOA and NS records. We modify the generated CNAME record by appending a ".test" suffix, integrating it into the zone file. Similarly, we append ".test" to the query name, ensuring its relevance to the zone file. We also fill in the RDATA values for A and AAAA records with proper IPv4 and IPv6 addresses. In contrast, for models like FULLLOOKUP that involve a complete zone file and a query, our approach slightly differs. Here, our primary check is for the presence of an SOA record. If absent, we add it, aligning with the method used for the CNAME model.

5.2 Eywa models

For test generation, we constructed eight models of the DNS using Eywa. Each of the models is shown in Table 1 and tests a different component of the protocol for authoritative nameservers. The table shows the number of lines of Python code that were needed to define each model in Eywa, including the prompt and constraints. The table also gives the range (both the minimum and maximum) of lines of code in C that Eywa generates for each of the *k*-different modeling attempts that are performed. Finally, we show the total number of unique

---

[1]The exact commits used are mentioned in the references.

Model	Lines of Python	Lines of C	Tests
CNAME	19	222 / 246	435
DNAME	20	209 / 230	269
WILDCARD	19	210 / 238	470
IPV4	19	209 / 229	515
FULLLOOKUP	35	487 / 510	12,281
RCODE	36	487 / 510	26,617
AUTH	35	477 / 504	31,411
LOOP	40	474 / 489	31,453

Table 1: The prompts, models, and outcomes from testing DNS with Eywa.

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.

To better understand when Eywa performs well and when it does not, we attempted to craft a diverse set of models. We created (1) simple modular 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.

The first four tests (CNAME, DNAME, WILDCARD, IPV4) all represent basic tests that determine whether a DNS query matches a single record in a DNS zone. The implementation of each is extremely simple and is similar to that of Figure 1. The number of tests generated is unsurprisingly small given the simplicity of the model (several hundred tests each).

The FULLLOOKUP model attempts to model the entire DNS authoritative lookup procedure for a query and a zone file in a one shot. We include instructions for Eywa to implement the correct semantics for each record type (CNAME, DNAME, A, AAAA) as well as wildcard records and more. Typically, a complete DNS implementation is highly complex [38] and involves walking the domain name hierarchy to find the "closest enclosers" to the user's query while simultaneously respecting various record type semantics and validity conditions. Two other models, AUTHORITATIVE and RCODE, are similar to FULLLOOKUP but return just 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, it asks Eywa for a model that counts the number of times a DNS query will be rewritten for a given zone file. This forces Eywa to explore queries and policies that may result in recursively looking up an answer many times or even indefinitely.

Each of the prompts uses a set of constraints to ensure we generate (mostly) valid DNS queries and zones. These include that: (1) all domain names (both for queries and records) are in the correct format, (2) records with a domain name target (CNAME, DNAME, NS) also have the target in the correct format,

and (3) the size of domain names to be length 3 (allowing for up to 2 labels) and the number of records in a zone file to be at most 2 to avoid state space explosion.

5.3 Eywa produces tests quickly

The running time for Eywa is dominated by the time to test the implementations. Each LLM query took under roughly 20 seconds to complete. For test generation using Klee, the time taken varies based on the complexity of the case. For the initial four cases, Klee usually completes the process in approximately 5-10 seconds without reaching its 5-minute timeout. However, for the later cases that require generating zone files, Klee consistently hits this 5-minute timeout.

The execution of each test across the 17 DNS (implementation, version) pairs took around 10 seconds. This includes setting up each Docker container with the necessary zone file and sending the query to get responses. To optimize this runtime, we reuse these containers across tests to optimize resources and time. If a server in a container fails to respond for a test, we deploy a fresh container. The setup process for each test involves stopping the running DNS nameserver, copying the new zone file and essential configuration files into the container, and then restarting the nameserver, all done in parallel across all 17 implementations. Given the high volume of tests, nearly 100,000 in total, we further parallelized the testing phase by running different container instances simultaneously. Despite these efficiencies, the testing phase took several days to complete.

Next, to evaluate the quality of Eywa models, we attempt to answer two questions: (1) do Eywa models reasonably approximate the protocol's intended behavior, and (2) how effective are Eywa models at generating test cases that reveal real implementation bugs.

5.4 Eywa produces high-quality models

To answer the first question, we qualitatively inspect Eywa's generated implementations by hand. For each of the models from Table 1, we found that LLM produced surprisingly good implementations that mostly capture the intended DNS protocol semantics. For instance, for the most challenging fulllookup task, the LLM correctly implemented the behavior of each record type including DNAME (suffix rewrite), CNAME (exact rewrite), and wildcard (partial match). While it did not perfectly implement the correct "closest encloser" semantics of DNS lookup (as doing so would require creating a complex data structure), the LLM instead frequently performed an incorrect "first-match" semantics by iterating through the list of zone records sequentially. While technically incorrect, this implementation is a close approximation of the true behavior and ended up producing an effective set of test cases when combined with symbolic execution.

In several other cases, the LLM implemented a flawed model due to its misunderstanding of side effects of the C `strtok` function. As a simple fix, updating the system prompt to instruct the LLM to avoid this function easily resolved the issue. In only 1 case did the LLM produce a C model that resulted in a compilation error, and in nearly every other case, the models were also functionally correct, showing the the LLM had a reasonable understanding of the protocol even without additional context (*e.g.*, RFC text).

5.5 Eywa produces high-quality tests

For the second question, we compare how many of the bugs in DNS implementations found by prior work SCALE, Eywa was also able to find. Compared to SCALE, which used a carefully designed DNS model constructed manually from RFCs, Eywa requires orders of magnitude less modeling effort by relying on LLMs. On the other hand, we expected the hand-written model to be more effective at revealing bugs.

We show the results from differential testing in Table 2. The table lists the bugs identified for each of the ten implementations tested along with a description of the bug and its type/effect. We additionally record whether this bug was found or would have been found by test cases produced by prior work SCALE.

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 an impressive **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.*

A closer look at the bugs found exclusively by SCALE revealed that four out of the seven missed by Eywa were discovered through tests involving invalid zone files combined with equivalence classes from GRoot [37]. These seven bugs required conditions such as more than two records, domain names with at least three labels, or records located at the zone apex, scenarios that could be covered by Eywa. Specifically, two of these bugs necessitated a scenario where two CNAME records in a zone form a chain, and the query targets the start of this chain with a CNAME query type. Eywa did generate similar test cases with chained CNAME records and queries targeting the chain's start, but these used an A record query type instead. This discrepancy arises because the models generated by LLM did not account for CNAME special cases in the generated code, which would require a profound understanding of DNS RFCs. Another contributing factor is Klee's preference to select the smallest value for a variable when given a choice. In Eywa's RecordType enum

Implementation	Description	Bug Type	New bug?
BIND	Sibling glue record not returned.	Wrong Additional	○
BIND	Inconsistent loop unrolling.	Wrong Answer	●
COREDNS	Wildcard CNAME and DNAME loop.	Server Crash	○
COREDNS	Sibling glue record not returned.	Wrong Additional	○
COREDNS	Returns SERVFAIL yet gives an answer.	Wrong Answer	●
COREDNS	Missing record for CNAME loop.	Wrong Answer	●
COREDNS	Returns a non-existent out-of-zone record.	Wrong Answer	●
COREDNS	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
COREDNS	Wrong RCODE for empty non-terminal wildcard.	Wrong Return Code	●
GDNSD	Sibling glue record not returned.	Wrong Additional	○
HICKORY	Wildcard CNAME and DNAME loop.	Server Crash	○
HICKORY	Incorrect handling of out-of-zone record.	Wrong Answer	●
HICKORY	Wildcard match only one label.	Wrong Answer	○
HICKORY	Wrong RCODE for empty non-terminal wildcard.	Wrong Return Code	●
HICKORY	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	●
HICKORY	Glue records returned with authoritative flag.	Wrong Flags	○
HICKORY	Authoritative flag set for zone cut NS records.	Wrong Flags	○
KNOT	DNAME record name replaced by query.	Wrong Answer	●
KNOT	Wildcard DNAME leads to wrong answer.	Wrong Answer	●
KNOT	Error in DNAME-DNAME loop Knot test.	Faulty Knot Test	○
KNOT	DNAME not applied recursively.	Wrong Answer	○
KNOT	Incorrect record synthesis when '*' is in query.	Wrong Answer	○
NSD	DNAME not applied recursively.	Wrong Answer	○
NSD	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
POWERDNS	Sibling glue record not returned due to wildcard.	Wrong Additional	●
TECHNITIUM	Sibling glue record not returned.	Wrong Additional	○
TECHNITIUM	Synthesized wildcard instead of applying DNAME.	Wrong Answer	●
TECHNITIUM	Invalid wildcard match.	Wrong Answer	○
TECHNITIUM	Nested wildcards not handled correctly.	Wrong Answer	●
TECHNITIUM	Duplicate records in answer section.	Wrong Answer	○
TECHNITIUM	Wrong RCODE for empty nonterminal wildcard.	Wrong Return Code	●
TWISTED	Empty answer section with wildcard records.	Wrong Answer	○
TWISTED	Missing authority flag and empty authority section.	Wrong Flags	○
TWISTED	Wrong RCODE for empty nonterminal (wildcard).	Wrong Return Code	●
TWISTED	Wrong RCODE when '*' is in RDATA.	Wrong Return Code	○
YADIFA	CNAME chains are not followed.	Wrong Answer	○
YADIFA	Missing record for CNAME loop.	Wrong Answer	●
YADIFA	Wrong RCODE for CNAME target.	Wrong Return Code	○

Table 2: The bugs found in different DNS implementations by EYWA.

definition, A appears first, leading to the majority of test cases featuring the A query type. If the enum were defined with CNAME or DNAME as the initial element, it's likely that more relevant test cases would have been generated.

5.6 Hyperparameter analysis

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
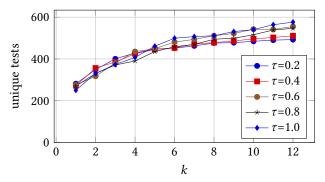
Figure 7: The number of total unique tests vs. the number of attempts k for EYWA for the CNAME model.

at for each k averaged over 10 runs. We plot this count vs. k in Figure 7 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.

6   RELATED WORK

EYWA is related to several lines of prior work:

Automated testing. There is a large body of work related to automatically testing hardware and software. These works are typically categorized according to their level of visibility into the source code. Black box testers [11, 23, 26, 55, 61, 71] for instance, generate random tests for implementations whose source code or specification is unavailable. Grey-box testers [6, 7, 10, 24, 27, 59, 68] use lightweight program instrumentation to obtain coverage feedback to guide future tests. White-box testers [9, 28–30] use heavyweight symbolic methods with access to source code to solve program path constraints directly. Each of these approaches has limitations – black and grey-box approaches often get "stuck" on hard to solve conditions and can result in low test coverage. Conversely, white-box approaches sidestep this problem 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 [9, 29, 30, 56].

Another approach is model-based or specification-based testing [38, 41, 46, 47, 66]. 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 hand craft a model — a non-trivial undertaking. Oracle-based testing is a form of model-based testing that offloads the modeling task to LLMs.

Protocol testing. Protocols in particular have been the target of much work on testing due to their inherent complexity [1, 4, 26, 31, 43, 53, 56, 58]. Many 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 [26] and AFLNet [58] 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 a LLM to directly encode a protocol's state and logic and requires no example traces for learning.

LLM-based testing. Recent breakthroughs in natural language processing and program understanding with LLMs have lead researchers to reconsider the possibility of automatically testing software using AI models [32, 40, 62, 64, 65, 67, 69, 72, 73, 75]. The majority of these works directly ask LLMs to write unit tests for software or to 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. Much of the system complexity lies in coordinating these two processes. More recently, ChatAFL [49] enhanced the stateful protocol fuzzer AFLNet [58] using LLMs. Like EYWA, ChatAFL leverages LLM knowledge to improve testing, but does so by directly modifying the protocol messages. These two approaches are distinct and may 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 [74] or generating network configurations [34]. In a recent workshop paper [63], 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

We introduce *oracle-based testing*, a new approach to automatic black-box protocol testing. Compared to model-based testing, oracle-based testing reduces the burden to manually derive an extensive protocol model. Compared to existing fully automated test generation techniques, oracle-based testing applies to black-box protocol implementations, requires no changes to source code, and produces tests that find deep functional correctness bugs.

We implement an oracle-based testing framework called EYWA and describe its new abstractions, features, and compilation to tests. Evaluating EYWA an the DNS protocol revealed 26 unique bugs in ten different implementations, including 11 new bugs that were previously undiscovered by manually constructed model-based testing tools.

REFERENCES

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

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

[3] Brandon L Black and Community. 2023. GDNSD. https://gdnsd.org/. (2023). Eywa commit: https://github.com/gdnsd/gdnsd/tree/877e15cf55593fa618d2009027e928d5f52da775.

[4] 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.

[5] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] Dnspython Community. 2023. Dnspython. https://dnspython.readthedocs.io/en/latest/index.html. (2023).

[14] 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.

[15] 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/dbcf683c1a57f49876e329fca183cb39d20ca3a4.

[16] curl. 2023. FTP Server Response Buffer Overflow. https://curl.se/docs/CVE-2000-0973.html. (2023). Accessed: 2023-9-29.

[17] curl. 2023. FTP shutdown response buffer overflow. https://curl.se/docs/CVE-2018-1000300.html. (2023). Accessed: 2023-9-29.

[18] 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.

[19] 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/89aaeb729a0856fefaed111c114ebb8a5a3f4ed2.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Hugo Gascon, Christian Wressnegger, 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.

[27] github. 2023. google/AFL. https://github.com/google/AFL. (2023). Accessed: 2023-9-29.

[28] 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.

[29] 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.

[30] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

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

[32] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting Greybox Fuzzing with Generative AI. *arXiv preprint arXiv:2306.06782* (2023).

[33] 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.

[34] 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.

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

[36] 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.

[37] 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

[38] 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

[39] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. 2023. DNAME record returned with query domain name instead of actual name. https://gitlab.nic.cz/knot/knot-dns/-/issues/873. (2023).

[40] 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).

[41] Sarfraz Khurshid and Darko Marinov. 2004. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering* 11 (2004), 403–434.

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

[43] 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.

[44] Twisted Matrix Labs. 2023. TwistedNames. https://twisted.org/. (2023). Eywa commit: https://github.com/twisted/twisted/tree/157cd8e659705940e895d321339d467e76ae9d0a.

[45] 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

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

[47] 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

[48] 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.

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

[50] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239 (March 2014), 2.

[51] Microsoft. 2023. Azure OpenAI Service. https://azure.microsoft.com/en-us/products/ai-services/openai-service. (2023).

[52] Paul Mockapetris. 1987. Domain names - implementation and specification. RFC 1035. (Nov. 1987). https://doi.org/10.17487/RFC1035

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

[54] 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.

[55] 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.

[56] 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.

[57] Ken Pfeil. 2023. 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.

[58] 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.

[59] 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.

[60] 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.

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

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

[63] 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.

[64] 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).

[65] 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).

[66] 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.

[67] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *arXiv preprint arXiv:2307.04346* (2023).

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

[69] 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).

[70] Wikipedia. 2023. 2022 Rogers Communications outage. https://en .wikipedia.org/wiki/2022_Rogers_Communications_outage. (2023). Accessed: 2023-9-29.

[71] 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.

[72] 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).

[73] 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.

[74] 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.

[75] 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).

[76] Shreyas Zare and Community. 2023. Technitium DNS Server. https://technitium.com/dns/. (2023). Eywa commit: https://github.com/TechnitiumSoftware/DnsServer/tree/d4352680b3f14fa2884fc8b7fa9c7772379bbc61.