# A Formal Framework for End-to-End DNS Resolution

Paper #683, 12 pages body, 25 pages total


## ABSTRACT

Despite the importance of DNS, numerous new attacks and vulnerabilities were recently discovered. The root of the problem is the ambiguity and tremendous complexity of DNS protocol specifications, amid a rapidly evolving Internet infrastructure. To interrupt the vicious break-and-fix cycle for improving the DNS infrastructure, we instigate a fundamental approach: we construct the first formal semantics of end-to-end name resolution, a set of facilities for formal analyses of both qualitative and quantitative properties, and a automated tool for exploring new DoS attacks. Our formal framework represents an important step toward a fundamentally secure and reliable DNS infrastructure.


## 1 INTRODUCTION

The Domain Name System (DNS) plays a central role in the Internet's functionality, availability, and security. After decades of incremental extensions to DNS, the complexity has reached epic dimensions. In fact, since the two initial RFCs defining the basics of DNS [37, 38], there have been over 300 RFCs published (even beyond the DNS Camel list [17]) to refine its design, implementation and operation (e.g., [6, 40]), to introduce new features (e.g., security and privacy enhancements [7, 11]), and above all, to elucidate many of its intricacies which surface only in real deployments (e.g., [12, 21, 26, 29]).

The complexity of DNS makes it notoriously difficult to manage and operate. Name resolution failures, as a result of misconfiguration, attacks, or operation errors, have caused large-scale outages [52]. In particular, denial of service (DoS) attack vectors [1, 13, 34] have been regularly discovered in the last decade, e.g., iDNS [34], DNS Unchained [13], NXNS [1], and TsuNAME [39].

As a principled way to mitigate this situation, a formal treatment of DNS is promising and highly desirable. As a representative of emerging formal efforts, Kakarla et al. [27] provide the first and only protocol-level formalization of DNS, and based on that, a static verifier called GRoot. It can identify common configuration errors in a set of zone files prior to their deployment. However, this seminal work comes with several limitations. First, their model abstracts away crucial aspects of DNS resolution, especially resolver-side logic including caching, recursive queries for nameserver names, data sanitization, etc. Second, their formalization is not accompanied by an executable model for computation and communication, which is essential to capture subtle dynamics in DNS. Third, their framework only supports the analysis of a certain class of correctness properties for DNS configuration. These together severely limit the scope of problems GRoot can identify in the intricate name resolution process.

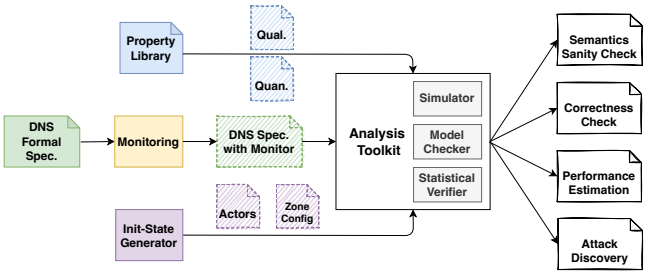


**Figure 1: Overview of our formal framework.**

In this paper, we present a comprehensive formal framework for rigorous analysis of DNS protocols. Figure 1 depicts an overview of the framework. It is based on Maude [16], a formal specification language and analysis tool that has been successfully applied to a wide range of distributed and networked systems [9, 31–33, 46, 48, 49, 51]. At the core of our framework is the most complete DNS semantic model to date. It captures all essential aspects of *end-to-end* name resolution: resolver cache, recursive subqueries, data credibility ranking, query name minimization, etc.; these are precisely the features that can be abused for DoS attacks [1, 13, 34, 39].

During the course of formalizing RFCs, we have identified and resolved a number of ambiguities in the forms of equivocacy and under-specification. We show that some of them, when interpreted in common ways by DNS implementors, can lead to serious DoS attacks. In this regard, our executable and mathematically precise semantics can serve as a *reference implementation*.

For a separation of concerns between semantics (the formalization of DNS resolution) and analysis, we develop our framework in a *modular* and *extensible* fashion. It comprises: (1) a monitoring mechanism that allows flexible definition of properties over the semantics; (2) a library of both *qualitative* properties (subsuming all those considered by GRoot [27]), and *quantitative* properties (e.g., amplification factor and query success ratio); and (3) an initial state generator that enable automation.

With the complete DNS semantics and rich facilities, our framework is versatile in formal analysis tasks: (1) simulation (by Maude's built-in simulator) for sanity check of semantics, quick system testing, etc.; (2) temporal model checking (by Maude's built-in LTL model checker) that can exhaustively explore the state space with respect to a correctness property such as the absence of rewrite blackholing; and (3) statistical verification (by the PVeStA statistical model checker [5]) that allows estimating system performance, e.g., query duration/latency, and analyzing mixed properties, e.g., *"What is the probability for a client query to be answered under DoS attacks?"*

As an appealing use case of our framework, we use it to develop a tool for automated analysis of DoS vulnerabilities in DNS. Our findings include the rediscovery of existing attacks [1, 13, 39] and the identification of multiple new attacks that can achieve large amplification effects. We have confirmed these attacks on popular DNS implementations including BIND, Unbound, and PowerDNS. The measurement results on these implementations are highly consistent with the predictions of our model, which demonstrates the accuracy and predictive power of our framework.

## 2 PRELIMINARIES

For background information on DNS, we refer readers to Appendix D.

**Maude and Actors.** Maude [16] is a rewriting-logic-based executable formal specification language. It also supports machine-checkable and automated formal analysis, including simulation, linear temporal logic (LTL) model checking [15], and statistical model checking (SMC) [2, 42]. The Maude system has been very successful in analyzing high-level designs of a wide range of distributed and networked systems [9, 31–33, 46, 48, 49, 51].

In Maude the basic units of specification are called modules. A Maude module specifies a rewrite theory [36] consisting of an equational theory [23], specifying the system's data types and the operations on them, and a collection of rewrite rules, modeling the system dynamics. We summarize Maude's syntax used in this paper and refer readers to [16] for details. Operators (or functions) can have user-definable syntax, with '`_`' denoting argument positions, as in `_+_`. Conditional rewrite rules are introduced with the keyword `crl`. Comments start with '`***`'. In addition to commonly used data types such as numbers, lists, sets, and maps, Maude also supports parameterized, customized data types. As we will see in Section 3, these data types facilitate our modeling of advanced DNS features such as cache.

We adopt the *actors* paradigm [3] in our formalization of DNS using Maude. Actors are a popular model for distributed systems where distributed objects communicate through asynchronous message passing. When an actor receives a message, it can change its state, send messages, and create new actors. These actions are determined by the received message and the actor's internal state.

In a nutshell, a system state, modeled as a Maude term called *configuration*, is a *multiset* of actors and messages built up with an empty syntax multiset union operator `__`. An actor is represented as a term `<` $addr$ `:` $ActorType$ `|` $att_1 : val_1, \ldots, att_n : val_n$ `>` where $addr$ is the unique address (or identifier) of the actor of type $ActorType$, and $val_1$ to $val_n$ are the current values of the attributes $att_1$ to $att_n$. A message `(to` $receiver$ `from` $sender$ `:` $content$`)` is also a term, consisting of the sender, receiver, and message content.

The following shows an example conditional rewrite rule that defines a system transition:

```
1 crl [l] : *** l is a rule label
2     (to svr from clt : w)
3     < svr : Server | a1 : x, a2 : y >
4   => *** system state from above to below
5     < svr : Server | a1 : x + w, a2 : y >
6     (to clt from svr : y - w) if y - w > 0 .
```

A message sent by the client actor `clt` (line 2) is consumed, provided the `if` condition holds (line 6), by the server actor `svr` (line 3), whose attribute `a1` is updated to `x + w` (line 5) with the incoming message's content `w` (line 2), and a new message is sent back to the client (line 6). Such a transition (resp. rule) is called a local transition (resp. rule) as it only mentions relevant, partial system state, e.g., the client actor is omitted in the rule. We adopt this modeling style in formalizing DNS as it renders concise and more readable rewrite rules.

## 3 THE DNS FORMAL SEMANTICS

In this section we present our formalization of end-to-end DNS resolution in Maude. The entire executable model[1] consists of 28 rewrite rules for the non-deterministic model 30 rules for the probabilistic model.

### 3.1 Scope and Assumptions

Our formalization includes the processing and exchange of protocol-level DNS messages by different actors during resolution, but not management tasks such as configuration, zone transfers [30], or dynamic updates [47]. Table 1 summarizes the RFCs covered by our formalization.

For the data representation, we choose an abstraction level that is reasonably close to actual implementations.

[1] https://github.com/SigComm23/dns-framework

**Table 1: The RFCs covered by our DNS formalization. The last column indicates which actor is mainly affected by the algorithmic changes introduced by an RFC.**

| RFC | Description | Main algorithmic change |
|---|---|---|
| 1034 [37] | Core specification | Both |
| 1035 [38] | Core specification | Both |
| 2181 [21] | Clarifications | Resolver (data ranking) |
| 2308 [6] | Negative caching | Resolver |
| 4592 [29] | Wildcards | Nameserver |
| 6604 [26] | RCODE clarifications | Both |
| 6672 [40] | DNAME redirection | Both |
| 8020 [12] | NXDOMAIN clarification | Resolver |
| 9156 [11] | QNAME minimization | Resolver |

For example, we model the different sections and RCODE in a DNS response, but not the status flags.

Regarding different types of DNS servers, we maintain a strict separation between (caching) *recursive resolvers* and *authoritative nameservers*. Although it is not prohibited for a single server to run in both recursive and authoritative mode [37], this is discouraged today even by the implementations that support it [45].

Regarding protocol dynamics, we refrain from any abstractions that alter the number of protocol-level messages exchanged between the actors. For example, a resolver must look up the nameserver's address before sending it a message, a crucial aspect ignored in the GRoot semantics [27]. We abstract away aspects of lower-layer protocols such as the maximum UDP packet size.

## 3.2 Formalizing DNS

We have developed two accompanying models for DNS: (1) an untimed, nondeterministic model mainly for qualitative analysis such as LTL model checking, and (2) a timed, probabilistic model mainly for quantitative analysis such as statistical verification. We use (1) to illustrate our formalization of DNS and describe how we can transform (1) into (2) in Section 3.3.

*3.2.1 Modeling Considerations.* The advanced DNS features such as caching and referrals and the complex system dynamics due to the subtle interactions of these features render a faithful modeling of DNS nontrivial. We underline our three major considerations for achieving a faithful model of DNS:

(i) Based on our insight into the RFCs, we consider the authoritative nameserver algorithm as a *stateless* function, which can therefore be modeled by a recursive *equation* in Maude, and the resolver algorithm to be *stateful* (e.g., a resolver must maintain global and per-query states due to referrals or query rewriting), which can therefore be specified by *rewrite rules*.

(ii) To faithfully model communication with respect to protocol-level messages, we adopt the actors paradigm (see Section 2) so that clients, resolvers, and nameservers can communicate in our model via asynchronous message passing.

(iii) We consider an abstract, yet sufficiently refined model to capture advanced DNS features such as resolver subqueries, caching, and QMIN. Together with (ii), this allows us to explore sophisticated DoS attacks that leverage these features (see Section 6).

Note that the state-of-the-art DNS formalization, GRoot, suffers from a limited coverage of DNS semantics (e.g., without considering caching, subqueries, or QMIN), thus incapable of performing certain useful formal analyses (see Section 6).

```
1 < root ,
2   (a . root-server . net . root |-> addrRoot) >
```

**Figure 2: A SBELT instance specified in Maude.**

*3.2.2 Actors, Messages, and Configurations.* Following the actors paradigm, we naturally model DNS in an object-oriented style; the DNS system state, represented by a configuration in Maude, is a multiset of a number of clients, resolvers, nameservers, and messages such as queries and subqueries traveling between these actors.

**Actors.** A client is modeled with two attributes: a list of queries that it intends to send and the address of the (single) resolver it connects to. We model the tree representation for zones (assumed by the nameserver algorithm) using a list which a nameserver stores.[2]

Unlike clients and nameservers, the local state of a recursive resolver is far more complex. We model its attributes by including both static (i.e., SBELT) and dynamic (i.e., caches, per-query state, pending queries, work budget) states. The following shows two examples which also utilize Maude's *parameterized*, *customized* data types.

*Example 1: Safety Belt.* The static state consists of a list of root nameservers, namely "safety belt" or SBELT, with their addresses. This attribute serves as a static fallback when the resolver does not have information to guide the nameserver selection. We customize SBELT using a compound data structure for storing the *mapping* of nameservers of a zone to their addresses. Figure 2

[2]This leads to no loss of information as the tree representation could be reconstructed from the list at any time.

```
cacheEntry(< a . target-ans . com . root, cname, inf, b . target-ans . com . root >, 5)
:: cacheEntry(< b . target-ans . com . root,a, inf,addrNScom >, 5)
```

**Figure 3: A positive cache instance specified in Maude.**

```
(to rAddr from cAddr : query(1, a . target-ans . com . root, a))

< cAddr : Client | queries: query(2, c . target-ans . com . root, a), resolver: rAddr, ... >

< rAddr : Resolver | cache: the cache instance in Figure 3, sbelt: the SBELT instance in Figure 2, ... >

< addrRoot : Nameserver |
      db: < com . root, ns, 3600, ns . com . root > < ns . com . root, a, 3600, addrNScom >, ... >

< addrNScom : Nameserver | ... >        < addrNStargetANS : Nameserver | ... >
```

**Figure 4: An example (intermediate) system state, consisting of one client, one resolver, three nameservers, and two messages sent from the client to the resolver (one message is pending).**

shows one SBELT instance modeled in Maude: if the resolver processes a query with a cold cache or no related cached entries, it will fallback to ask one of the root nameservers. We set the preferred nameserver of `root` to be `a.root-server.net.root`. This root nameserver corresponds to the nameserver with the associated address `addrRoot`.

*Example 2: Caches.* One important part of the dynamic state is the resolver's cache. There are different types of caches: a normal (positive) cache stores resource records that the resolver received in responses; the negative cache stores NXDOMAIN and NODATA responses. They are defined analogously. For example, using a *parameterized* list, we define a positive cache as `List{cacheEntry}` in Maude with an associative concatenation operator `::`; a cache entry `cacheEntry` is a pair that consists of a record and an associated credibility score, indicating the "relative likely trustworthiness" of the data. Figure 3 shows one positive cache instance specified in Maude. The first cached entry displays a `cname` record received by a resolver, whose owner name is `a.target-ans.com.root` and target name is `b.target-ans.com.root`. The resolver then tries to resolve the target name and receives an `A` record, corresponding to the second entry of the cache. This `A` value is the address of the `com` nameserver. In both cached entries, the *Time-To-Live* (`TTL`) is set to `inf`, meaning that the cached record is never expired. Note here this is the case for our nondeterministic, untimed model; we can assign real values to the TTL in our probabilistic model (see section 3.3).

**Messages.** DNS messages, including (sub)queries and responses, are modeled in the same format (`to` *receiver* `from` *sender* `:` *content*) and only differ in the message content. See Figure 4 for an example client query.

**Configurations.** A system state, whether an initial state, an intermediate state, or a final state, is modeled in Maude as a term of configuration. Putting the above two examples together, we show an instance (intermediate) configuration in Figure 4 with one client containing a "pending" query (line 3), one resolver (line 5), three nameservers (lines 7–10), and one query sent from the client to the resolver (line 1).[3] The client actor has type `Client`, address `cAddr`, and the address of the resolver to be queried `rAddr`. The client `cAddr` already sent the query with identifier `1`. Another outgoing query by client `cAddr`, with its identifier `2` and the record type `a`, is waiting for the resolver to consume. The caches of `root`, `com` and `target-ans` are populated with database entries that link one to another and records they are authoritative of. Cached entries resulting from the resolution of the first query are contained in the caches of resolver `rAddr`. This results in two entries in the resolver cache, displayed in the Figure 3. Client `cAddr` buffers the other query `[2, 'c . 'target-ans . 'com . root, a]`. When this query is issued out, the value of the attribute `queries` in the client becomes `nil`.

*3.2.3 DNS Dynamics.* We use rewrite rules to specify DNS dynamics which take a (partial) configuration, representing the current system state, and transform it into a different configuration, representing next system state. To distinguish our formalization with refined resolvers and caching, Figure 5 depicts an example rule which

[3]We omit other attributes, marked by "...", for the simplicity of the presentation.

```
1  crl [resolver-recv-ans-for-client] :
2   (to RSV from NS : response(ANS, ...))
3   < RSV : Resolver | cache: CACHE, ... >
4  =>
5   < RSV : Resolver | cache: updt(CACHE'), ... >
6   (to ADDR from RSV : RESP)
7  if *** Create temporary cache from response
8       CACHE_TMP := updateCacheAuthAns(ANS, ...)
       /\ *** logical conjunction
9       *** Cache hit in temporary cache (i.e.,
       the response answers the query)
10      RESP := responseFromCache(CACHE_TMP, ...)
       /\
11      *** Cache the data from the response
12      CACHE' := updateCacheAuthAns(ANS, ...)
13     /\ ...
```

**Figure 5: Snippet of the rewrite rule corresponding to Step 4, Case A of the resolver algorithm from RFC 6672.**

corresponds to Step 4, Case A of the resolver algorithm from RFC 6672:

> If the response answers the question or contains a name error, cache the data as well as return it back to the client.

This rule has 74 LOC with nontrivial auxiliary functions and rule conditions. For the simplicity of the presentation, we only show the most important snippet with respect to positive caching.

This rule applies for a response that authoritatively answers a client query. More specifically, a temporary cache is created from the data contained in the response (line 8), which is then used for the lookup (line 10). Note that we cannot perform the lookup directly on the actual cache as case A of the resolver algorithm should only consider the data in the response, not in the cache. Also note that we look only at the data in the answer section (`ANS`, line 2) for the temporary positive cache as the entire rule is concerned with *authoritative answers*. The function `updateCacheAuthAns` (line 12) used for the creation of the temporary cache inserts the records from the answer section. Finally, we insert the data from the response into the actual cache and use this updated cache on the right-hand side of the rule (line 5).

### 3.3 Probabilistic Model

The model defined thus far is untimed and nondeterministic, which is not well-suited for (1) capturing full TTL semantics and timeouts in DNS or (2) quantitative analysis with, e.g., SMC. Hence, we transform it into a timed, probabilistic model by following the systematic methodology in [4]. The idea is to assign to each message a delay sampled from a continuous probability distribution,[4] which determines the firing of the rule receiving the message. To schedule the delayed messages in time (multiple delayed message may appear in a single system state), we implement in Maude a scheduler actor with a global clock. The elapse of message delays, as well as advancing the global clock, is maintained by the scheduler. Moreover, unlike the untimed model where a cached record is never expired (TTL set to `inf`), we can now explore the TTL space by configuring different values (e.g., 3600 time units by default in our experiments).

## 4 RESOLVING AMBIGUITIES

The formalization of our executable and verifiable DNS semantics forces us to iron out any imprecision in RFCs. Ambiguities in specifications are often at the root of implementation bugs, configuration errors, and security vulnerabilities. For illustration, we discuss three representative ambiguous cases relevant to the new attacks we discovered (Section 6). Appendix B shows how we resolved other imprecisions.

### 4.1 Ambiguities in Resolver Algorithm

**Resolver subqueries.** One of the most perplexing aspects of DNS resolution is the subqueries generated by a resolver to resolve the names of nameservers themselves – reflecting the recursive nature of name resolution. Such subqueries are internal to a recursive resolver and hidden from clients; yet, from an authoritative nameserver's point of view, resolvers' subqueries are no different from any type of queries it may receive. An important decision to make is thus whether resolver subqueries are treated as special case during resolution.

One particular question arises whether CNAMEs should be followed in subqueries, a situation that occurs when an NS record points to an alias. By the initial definition [38], the target of an NS record must be a domain name but not an IP address, without any other explicit restriction. Later, it is clarified that nameserver aliases are forbidden [21, §10.3]:

> The domain name used as the value of a NS record, or part of the value of a MX record must not be an alias.

However, the specification also advocates a *robustness principle* [37, §3.6.2] to reduce the risk of failure:

[4] In our experiments we use the lognormal distribution that characterizes the network latency in realistic deployments [8, 22].

> ... domain software should not fail when presented with CNAME chains or loops; CNAME chains should be followed and CNAME loops signalled as an error.

Divergent treatments are observed in major implementations: BIND immediately aborts a query if a nameserver name is found to be an alias, whereas Unbound, PowerDNS and Knot all follow the CNAMEs as usual. We choose to follow the robustness principle in our model. This enables us to analyze practical systems that are more forgiving and unveil more problematic resolution behaviors that otherwise remain hidden.

**Data credibility.** A DNS answer can contain varying records that answer the corresponding query with different levels of relevance and credibility. There must be proper policies in place for resolvers to decide which records to accept and cache and how to use them. A data ranking policy is specified for this purpose [21, §5.4.1]. Below are two rules from the policy (a larger number indicates a higher level of credibility):

> 5. The authoritative data included in the answer section of an authoritative reply.
> 2. Data from the answer section of a non-authoritative answer, and non-authoritative data from the answer section of authoritative answers.

When multiple relevant records are present in an answer, the one with the highest rank takes precedence. If a received record coincides with a cached one, the one of a higher rank should be retained in the cache. Given that resolvers can receive all sorts of answers from nameservers also with varying levels of trustworthiness (e.g., well-behaving, misconfigured, or malicious), it is difficult to interpret and implement these rules precisely and comprehensively. To complicate the matter further, in many situations a resolver necessarily needs to use data of the lowest credibility level, such as glue records, to function correctly.

In our formalization, we resolve any equivocal cases by mapping them to exactly one possible path of the resolver's execution, strictly following the data ranking policy. This, in turn, has helped us address imprecision in the resolution algorithm. Furthermore, to enable fine-grained resolution policies and the analysis of diverse interpretations of data credibility rules, we introduce two configuration options, `rsvMinCredClient` and `rsvMinCredResolver` (Table 2); they control the minimum credibility of data that a resolver will accept for client requests or the resolver's subqueries.

From the same RFC [21, §5.4.1], there is one further remark of particular interest about data credibility:

> Note that the answer section of an authoritative answer normally contains only authoritative data. However when the name sought is an alias only the record describing that alias is necessarily authoritative ... Where authoritative answers are required, the client should query again, using the canonical name associated with the alias.

It suggests that when receiving a CNAME record of the queried name, a resolver (in the role of client) should always query again the rewritten name because only that record is necessarily authoritative. As a corollary, a resolver should always query each name on a CNAME chain. However, a dilemma arises when all names on the chain are in the same authoritative zone and are packed into a single answer. In this case, the resolver's behavior is left undecided: either to discard all but the first CNAME records and repeatedly query until the last one, or accept all these records as authoritative and process the rewrite chain locally, without sending extra queries. In fact, all the DNS implementations we tested follow the first approach; we also adopt the strict definition that only the record describing an alias is authoritative in our semantic model. Nonetheless, with the default value (2) for `rsvMinCacheCredClient` and `rsvMinCacheCredResolver`, the model becomes more forgiving and can capture the other case as well, indicating our framework's configurability and flexibility.

**QMIN Limits.** While conceptually simple, QMIN substantially complicates the resolution process. In order to find proper zone cuts, a recursive resolver may send multiple probing queries to the same nameserver, with one more label added in each iteration. It is explicitly mentioned [11, §2.3] that:

> When using QNAME minimisation, the number of labels in the received QNAME can influence the number of queries sent from the resolver. This opens an attack vector and can decrease performance. Resolvers supporting QNAME minimisation MUST implement a mechanism to limit the number of outgoing queries per user request.

Nonetheless, among other ambiguities (Appendix B.2), it remains under-specified how to deal with those limits when resolver subqueries are triggered for a client request, and when CNAME or DNAME records are received by the resolver. Since QMIN's primary purpose is to improve privacy, our model opts to enforce the limit for each query regardless of a query's nature. This is also the approach taken by all resolver software used in our measurement.

**Table 2: A partial list of the configuration options supported by our semantic model**

| Option | Definition | Default |
|---|---|---|
| `rsvMinCredClient` | The minimum credibility requirement [21] for data served to a client | 2 |
| `rsvMinCredResolver` | The equivalent credibility requirement for resolver subqueries. | 2 |
| `maxMinimiseCount` | The MAX_MINIMIZE_COUNT parameter to limit extra work for QMIN [11]. | 10 |
| `minimiseOneLab` | The MINIMIZE_ONE_LAB parameter from the same mechanism above | 4 |
| `rsvTimeout` | Whether and how long a resolver applies a timeout for each query it sends. | false, 20.0 |
| `rsvOverallTimeout` | Whether and how long a resolver applies an overall timeout for a client request. | false, 100.0 |
| `workBudget` | The max number of *any* queries that a resolver *sends out* for a client request. | 75 |
| `maxFetch` | Whether and how much to enable MaxFetch($k$) [1] to limit concurrent subqueries. | false, 1 |

### 4.2 Semantics Sanity Check

We have extensively validated the correctness of our DNS semantics using unit tests and numerous simulation runs. In the model, the rewrite rules use around 150 auxiliary functions to specify resolver operations (cache lookup and manipulation, query state initialization and update, the creation of subqueries, etc.), and recursive equations are used to define nameserver operations (DNAME substitution, wildcard processing and glue additions, etc.). To ensure their correctness, we defined over 470 unit tests covering both normal and edge-case inputs. The number of test cases for a function varies depending on its complexity, e.g., 45 test cases for the core function of the authoritative nameserver algorithm alone. Moreover, during our automated attack analysis, we have generated millions of randomized configurations. Our tool always checks if the execution ends in a well-formed state of both the resolver (no pending/blocked queries, etc.) and the client (all queries have been sent). This gives high assurance of the semantics' correctness.

## 5 FORMAL ANALYSIS

In this section we describe how to use our framework for various types of formal analysis. As shown in Figure 1, we provide three executable tools: simulator, model checker, and statistical verifier. They take as input the DNS semantics augmented by a monitoring mechanism (Section 5.2), an initial state (Section 5.1), and for the latter two tools, some property of interest (Section 5.3). Depending on the type of analysis, the output can be an final system state with useful information for subsequent analysis (*simulation*), a verified qualitative property or counterexample (*model checking*), or performance estimations (*statistical model checking*).

### 5.1 Initial State Generation

We implement a *configurable* initial state generator to facilitate automated analysis. The generator takes as input a range of parameters and output a Maude module, which can be used to invoke one of our analysis tools. Two classes of parameters are allowed: (1) one to configure actors' behavior (especially the resolver) and (2) the other to control the generation of zone files and how they are assigned to nameservers. Table 2 provides a partial list of the first class of parameters pertinent to our discussion in Section 6. The second class of parameters include the number of zones, allowed record types, the depth of names, and the probability of non-existent names, to list a few. We present our random zone generation algorithm in Appendix A.1. While in principle the generator can be implemented with Maude as well, we have chosen to develop it in Python for efficiency.

### 5.2 Monitoring

We define properties using a global *monitor object*, which is attached to the rewrite rules and records statistics of system executions. This mechanism allows us to maintain a strict separation of concerns between the semantics (the formalization of DNS resolution) and the analysis performed with it. Furthermore, it suffices for the monitor to record an *abstraction* of the system state, which consists of only the information relevant for the properties, while omitting many details from the full state. Finally, the analysis part can be easily *extended* without touching the semantics: one simply defines new attributes and updates them in the monitor part of rewrite rules. Appendix A.2 provides additional details.

### 5.3 Property Library and Formal Analysis

We support the analysis of two types of properties:

(1) *Qualitative* properties, e.g., those considered by GRoot [27] and beyond. They are used mainly to check the correctness of a modeled system using Maude's LTL model checker [16].
(2) *Quantitative* properties, e.g., the amplification factor that a query induces at the resolver, the probability that a client request is successfully answered,

```
1  *** definition of query success ratio
2  op qrySuccRatio : Config -> Float .
3  eq qrySuccRatio(< M : Monitor |
4       qryAnswered: N, qryFailed: N', STATE > C)
5   = float(N + (- N')) / float(N) .
6
7  *** QuaTEx interaction with Maude
8  eq val(1,C) = qrySuccRatio(C) .
9
10 *** QuaTEx formula for query success ratio
11 querySuccessRatio() =
12     if { s.sat(0) } then { s.rval(1) }
13         else # querySuccessRatio() fi ;
14 eval E[querySuccessRatio()] ;
```

**Figure 6: QuaTEX formula for query success ratio.**

and average query latency. They are used for performance estimations using Maude's simulator or the PVeStA statistical verifier [5].

We specify (1) using LTL formulas [15] while (2) in QuaTEx [4], a quantitative temporal logic that extends *probabilistic computation tree logic* [25] by supporting real-valued expressions. Below we briefly discuss one example for each type of property. The details on their specifications, as well as the library of predefined properties offered by our framework, can be found in Appendix A.3.

**Rewrite Blackholing.** When a QNAME is rewritten (by CNAMEs or DNAMEs) to a non-existent name, rewrite blackholing occurs and signals a configuration error. A correctness property is thus defined as whether a given configuration (potentially with multiple zones) contains any rewrite blackhole. We can specify the property over a response log maintained by the monitor, checking whether there is a response with a non-empty answer section and NXDOMAIN code.[5] Our model checker can then take this property and verify whether it is violated for given zones and queries.

**Query Success Ratio.** An important performance metric in DNS is the fraction of *successful* queries (i.e., those not answered with a SERVFAIL response). Measuring such ratios can help us analyze, e.g., the effects of DoS attacks. A nameserver overwhelmed by a DoS attack will generate a lot of SERVFAIL responses. Defining the ratio is straightforward with the numbers of answered and failed queries stored in the monitor.

Figure 6 shows the definition of this property, together with the QuaTEx formula[6] for statistical model checking with PVeStA. The definition is straightforward with the numbers of answered and failed queries stored in the monitor. Line 10 defines the temporal operator `querySuccessRatio()`: PVeStA returns the ratio (computed by the Maude function to which `rval(1)` refers; line 7) if all client queries are finished (checked by the predicate `sat(0)`) in the current state `s`. Otherwise, it evaluates `querySuccessRatio()` on the *next* state denoted by `#`. Line 11 computes the *expected* ratio.

In one of our case studies, we statistically measure the query success ratio under the NXNS attack [1]. With 0.05 error margin and 95% statistical confidence[7], PVeStA returns 0.71, i.e., roughly 71% of legitimate client queries are still answered successfully. With a stronger attack setting that doubles the rate of malicious client queries, the success ratio decreases to 0.52. We further implement the MAXFETCH(K) mitigation [1], which limits the number of resolver subqueries per client query to $k$. and analyze the stronger attack setting again. For $k = 5$, we obtain a success ratio of 0.86, while for $k = 1$, a ratio of 0.97, i.e., almost all client queries are answered successfully, suggesting that this mitigation is indeed effective against the NXNS attack.

## 6 AUTOMATED ATTACK ANALYSIS

In this section we demonstrate the predictive power of our framework with one important application: automated attack analysis. In particular, we are interested in DoS vulnerabilities inherent in DNS at the design level. Our results include the (re)discovery of both recent attacks and *new vulnerabilities* that can lead to large amplification effects. We have validated and confirmed all our new attacks on real DNS software.

### 6.1 Methodology

The problem of identifying DoS vulnerabilities in DNS consists in finding a combination of nameserver configurations and client queries that leads to potential attacks. The nameserver configurations are given by the zone files and their allocation to nameservers. An attack occurs when resolution of the given client query (or queries) produces an undesirable effect, e.g., large amplification of query load at the resolver or a long duration during which the query occupies resources at the resolver.

**Challenge.** The ultimate goal of any automated attack discovery approach is to exhaustively explore the search space to find all possible attacks. In our case, this would entail exploring all possible nameserver configurations

[5]The answer section of a NXDOMAIN response can only be non-empty due to followed CNAME or DNAME records.

[6]A QuaTEx formula is mainly a query of the expected value of a path expression interpreted over an execution path with state expressions interpreted over states. See [4] for details.

[7]An SMC analysis returns the expected value $\bar{v}$ of a QuaTEx query w.r.t. two parameters $\alpha$ and $\delta$: $\bar{v}$ is obtained such that, with $(1-\alpha)$ statistical confidence, it lies in the interval $[\bar{v} - \frac{\delta}{2}, \bar{v} + \frac{\delta}{2}]$.

and queries. Unfortunately, this is intractable because of the huge the configuration space: the number of possible zone files is bounded only by the maximum length of DNS names (254 characters), the allowed character set (alphanumerical plus hyphen-minus), and possible record types (several tens); meanwhile, zones can be arbitrarily assigned to nameservers.

An interesting question is whether the query space is tractable if we fix the nameserver configurations. Clearly, the number of possible queries is also huge because the QNAME can be a non-existent name. Yet, it may be possible to treat some queries as equivalent when they are resolved in the same way and produce the same result. Kakarla et al. defined such a notion of *query equivalence classes (EC)* [27]. However, while this mechanism is manually proved sound and complete with respect to their oversimplified DNS semantics, it is unsound in our more realistic model (explained in Appendix C.1).

**Our approach.** We develop a highly effective attack analysis approach based on the following insights. First, attacks depend mostly on the contents of a few malicious zones, whereas the configuration of the benign part of the DNS namespace is irrelevant. Second, many interesting interactions can be uncovered by relatively small zone files, consisting of only a few names. Third, while the GRoot EC is unsound with respect to our model, they still provide an effective heuristic for the exploration of the query space. Lastly, whether or not an initial state leads to an attack rarely depends on the particular non-deterministic choices made during execution (e.g., message delays). More often, it suffices to analyze *one* possible execution to determine if the configuration is interesting from the perspective of attack discovery.

In our automated approach, a user provides the *benign* part of an initial state, consisting of a resolver and multiple nameservers. Based on that, our tool uses the random zone generator to create a number of *malicious* zone files and adds them to the benign initial state by introducing additional nameservers serving these zones. For this augmented configuration, the tool then computes GRoot ECs and creates a separate initial state for each EC, with a client sending a representative query from the EC. Afterwards, the tool invokes our simulator with each state for execution, computing the metric of interest. The resulting value is then compared to a user-supplied threshold. If it exceeds the threshold, the execution is considered a successful attack and the initial state is stored for later inspection.

**Table 3: Default limits enforced by resolver implementations**

| | BIND 9.18.4 | PowerDNS 4.7 | Unbound 1.16 |
|---|---|---|---|
| Max subqueries | 5 | 5 | 6 |
| Max CNAME chain | 17 | 12 | 12 |
| QMIN iterations | 5 | 10 | 10 |

## 6.2 Re-discovery of Known Attacks

With a simple setup, our tool can already identify known DoS vulnerabilities in DNS. For the benign part of the initial state, we use a `benign.com` zone consisting of several basic SOA, NS, A records. We also provide a root, `com` and `net` zones, as well as a `root-servers.net` zone (which contains the information for the single root server `a.root-servers.net`), all hosted by separate nameservers. This simulates a minimal realistic setup of the DNS namespace. We configure the tool to record an exploration run as suspicious whenever the number of queries sent by the resolver for one client request exceeds 15, a moderate threshold to discern amplification attacks [41]. For our semantic model, we use the default configuration (Table 2) except that we disable QMIN to avoid noise (it is considered later).

We discuss three attacks below, explaining the usage and configuration of our tool along the way. This serves as the basis to analyze more complicated attacks.

**NXNS attack.** We first focus on a single malicious zone. Our tool already reports a potential attack after a few tens of runs for the following zone.

```
< atk1.com., SOA, 3600 >
< r.atk1.com., NS, nxdomain0-1.benign.com. >
< r.atk1.com., NS, nxdomain1-1.net. >
< r.atk1.com., NS, nxdomain2-1.ns.com. >
...
< ss.c.atk1.com., DNAME, *.benign.com. >
< qq.c.atk1.com., DNAME, net. >
```

The zone contains an unusual number of NS records for `r.atk1.com` pointing to non-existent nameservers. The amplification is due to the subqueries that the resolver generated for each of those nameservers, as confirmed by the monitor's query/response logs. This is precisely the mechanism underlying the NXNS attack [1], although ideally the non-existent nameservers are all located below a victim domain. Since there is no explicit victim in our setup, different names from the benign zones are used to construct the nameserver names.

**TsuNAME attack.** If we generate initial states with *two* random malicious zones, the following is reported as a potential attack configuration:

```
--- zone atk1.com
< rr.j.atk1.com., NS, nxdomain0-1.o.atk2.net. >
```

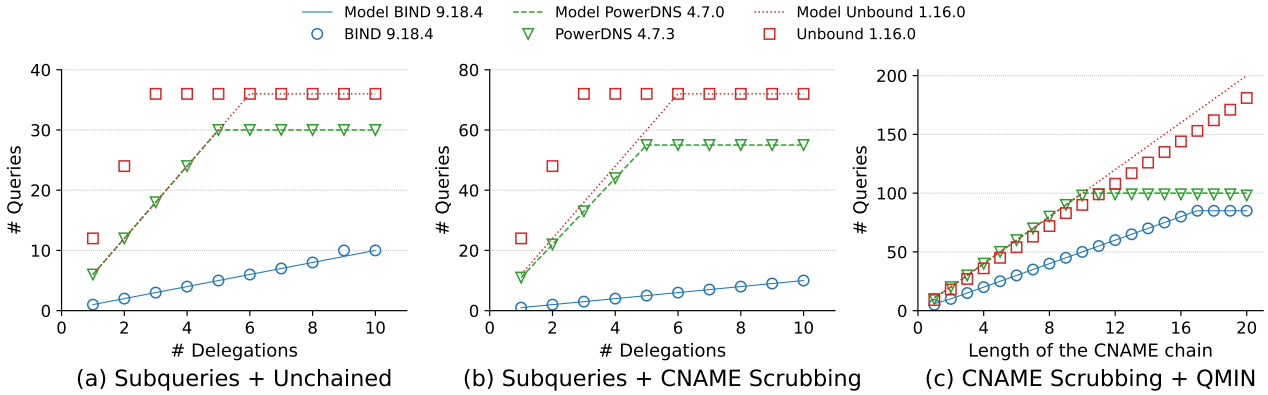


**Figure 7: Measurement results of our new attacks that produce large amplification by combining vulnerabilities.**

```
--- zone atk2.net
...
< o.atk2.net., NS, nxdomain7-1.benign.com. >
< o.atk2.net., NS, nxdomain8-1.ns.net. >
< o.atk2.net., NS, nxdomain9-1.rr.j.atk1.com. >
```

At first sight, the second zone looks similar to the one shown above for the NXNS attack. However, the reported amplification for is much higher for `vv.o.atk2.net`. This is due to a circular dependency: one of the nameservers for `vv.o.atk2.net` is `nxdomain9-1.rr.j.atk1.com`, which is hosted by a nameserver under the zone `atk2.net` itself. We can observe TsuNAME-like behavior [39] with repeated queries sent to the `atk1.com` zone until the resolver's `workBudget` is eventually exceeded.

**Unchained attack.** One could also hope to similarly rediscover the Unchained attack [13], which forces a resolver to follow a CNAME chain split between two zones hosted by different nameservers. However, because of the many possibilities for RRsets, we will have to generate a large number of zones before a long CNAME chain is generated. Yet, our tool allows us to flexibly set a different focus for each attack exploration run. If we perform such a run with a focus on CNAME records in random zone generation and additionally restrict the *values* of records to contain only names in other malicious zones, the tool will report suspicious Unchained behavior.

## 6.3 Discovery of New Attacks

We now show that minor tweaking of our tool's parameters leads to the discovery of more complex interactions of DNS features that produce larger amplification than the previous vulnerabilities. As demonstrated by the new attacks reported below, our tool can analyze amplification in terms of both *query load* and *query delay*.

For space reason, we provide example malicious zone files in Appendix C.3. To validate the attacks, we created a testbed (see Appendix C.2) using real DNS implementations that mirrors the configuration of our model. Table 3 summaries these implementations' default parameters relevant to our discussion.

**Subqueries + Unchained.** Our analysis of Unchained focuses on CNAMEs in random zone generation. By additionally allowing NS RRsets for leaf zones, our tool reports attack cases with high amplification factor: the large delegation leads to the creation of multiple resolver subqueries, and each of them triggers an Unchained-style CNAME chain. The amplification is thus *multiplied*.

Figure 7 (a) reports our measurement results for this attack with varying size of the malicious NS RRset. For each resolver implementation, we tune the `maxFetch` parameter in our model to match its subquery limit so that we obtain comparable behavior. In addition, we use CNAME chains of the appropriate length to match the limits enforced by the corresponding implementations.

The data measured is the number of queries received by the victim nameserver (one of the two nameservers involved in an Unchained attack). As can be seen, BIND and PowerDNS match our model's prediction almost exactly. BIND is not vulnerable to this attack, because it does not follow CNAME chains starting from nameserver names. PowerDNS limits the maximum CNAME chain length it follows to 12 and the maximum number of subqueries it sends to 5; it reaches its overall limit as expected, since only half of the queries for the CNAME chains goes to the victim. Unbound generates even more queries than predicted by our model. Inspection of the query logs show that Unbound has a fallback mechanism that attempts resolution of the nameserver names

twice. With a subquery limit of 6, it already reaches the maximum amplification at three delegations.

**Subqueries + CNAME Scrubbing.** As mentioned earlier (Section 4.1), with a strict interpretation of data credibility rules, a resolver will not trust an entire CNAME chain received in a single response but continue to query the canonical name of the first record. This is unofficially referred to as *CNAME scrubbing* in the DNS development community. We can enable this behavior for our model by increasing `rsvMinCredClient` and `rsvMinCredResolver` to 5. With this change, the tool reports zones similar to the previous attack, except that the CNAME chains pointed to by the NS records can now be contained within a single zone, rather than split across two or more zones as in Unchained.

In theory, this attack should achieve twice as high amplification factor as the previous attack. This is confirmed by our measurement results depicted in Figure 7 (b). Again, both PowerDNS and Unbound are vulnerable to the attack but BIND is not. To the best of our knowledge, despite being more powerful than Unchained, the exploitation of CNAME scrubbing for amplification has never been reported before.

**CNAME Scrubbing + QMIN.** In our analysis of all previous attacks the QMIN feature is disabled. We now turn it on by setting a non-zero value for `maxMinimiseCount`. Other settings for random zone generation include keeping CNAME scrubbing enabled, prohibiting NS RRsets for leaf zones, and increasing the depth of names (i.e., more labels). We also increase the resolver's `workBudget` in our model. With this configuration, our tool reports the most powerful attack so far: every name on a CNAME chain triggers multiple QMIN probing queries sent to the victim nameserver.

All three resolvers are highly susceptible to this attack, as shown in Figure 7 (c). Depending on their implementation of the QMIN algorithm, the attack can incur up to 10 queries for each element on a long chain. BIND and PowerDNS hit a maximum number of queries of 85 and 100, respectively, because they impose overall resource budgets for each client request. Interestingly, Unbound keeps generating queries in some of our test cases even if it exceeds its limit on CNAME chains. After investigation, we additionally discovered a serious bug in Unbound where it does not enforce the maximum length of CNAME chains for record types other than A. These findings again demonstrate the value of our framework in efficiently identifying subtle problems overlooked in DNS implementations.

**Amplified Slow DoS.** In addition to common amplification attacks, our tool can also be applied to more advanced analyses with the *time dimension* factored in. A resolver maintains complex state for each ongoing query. A *slow DoS attack* keeps a query "alive" in a resolver for as long as possible, occupying resources that can otherwise be used for legitimate queries.

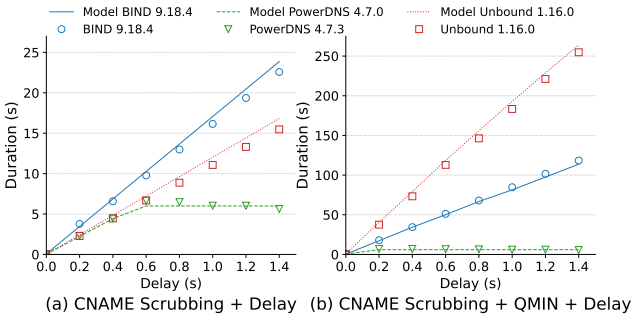


**Figure 8: Measurement results of our new slow DoS attacks that prolong query processing time at a resolver.**

Our analysis now focuses on producing such slow queries that decelerate the resolution of a malicious client query. To facilitate this, we introduce an artificial delay in the responses of attacker-controlled nameservers and then generate zone files where the client queries remain unanswered for a long time. The artificial delay must be smaller than the resolver timeout for each query (`rsvTimeout`), otherwise the nameserver is deemed unreachable. We also adjust our model's `rscOverallTimeout` parameter so that it is higher than the threshold set for attack alarm. Another important consideration here is that the resolver's query duration depends on probabilistic message delays. Therefore, we use statistical model checking to estimate the duration, rather than a single simulation (see Section 5.3).

With these settings, our tool automatically reports slow DoS attacks for malicious zone files which force the resolver to, in the resolution of one client query, send multiple queries *sequentially*, e.g., when long CNAME chains or deep names (with QMIN turned on) are encountered.

We again validate these findings on different resolver implementations, analyzing how long a malicious query can be kept ongoing at the resolver. To this end, we measure the time between the first and last query received at the malicious nameserver. This allows us to obtain comparable measurements for different implementations without the need to consider resolver internals. Moreover, we need to establish a relationship between the abstract time units used in our model and the real time observed in our experiments. While this is non-trivial in general, it is straightforward in our case because the artificial delays are orders of magnitude larger than the normal

message delays. We can thus base the relation solely on the artificial delay, and equate one time unit with one millisecond of real time.

Figure 8 depicts our measurement results with varying artificial delay for the attacks enabled by CNAME scrubbing (a) and additionally QMIN (b). The maximum artificial delay we show is 1.4 seconds, which is the largest value for which we were able to obtain meaningful results for all implementations. For larger values, some implementations do not make any progress in resolution due to, e.g., a query timeout of 1.5 seconds. If an implementation has a maximum query duration (i.e., 12 seconds for PowerDNS), we also set our model's `rscOverallTimeout` accordingly.

As can be seen from the plots, real resolvers behave closely to our model. The small gaps observed are due to the small variance in our testbed's network conditions. In general, the total query duration at all resolvers increases as more delay is added by the malicious nameserver. The difference in slope for is due to the different limits enforced by the implementations. Comparing the two plots, we can see that QMIN significantly intensifies the slow DoS attack. A single malicious client query can take up the resources of Unbound by over 250 seconds. Unbound suffers from much larger amplification than BIND when QMIN is enabled. This is because Unbound has a higher limit for QMIN iterations, which outweighs its slightly lower limit for CNAME.

Our findings are concerning. These amplification vulnerabilities can be exploited to launch highly effectively DoS attacks at low costs. Using our framework, DNS developers and operators can proactively discover and fix serious security vulnerabilities.

## 7 RELATED WORK

**Formalizing DNS semantics.** GRoot [27] provides the first formalization of a subset of DNS semantics. We already discuss its limitations in Section 1. Applying parts of the GRoot semantics, its authors also created SCALE [28], a test case generator to find RFC compliance errors in nameserver implementations. While SCALE allowed for the identification of implementation bugs, it focuses solely on the local processing logic at authoritative nameservers and does not consider the complete name resolution process with a recursive resolver.

**Verification of DNS Implementations.** Son et al. [44] formalized the cache implementations of popular open-source resolvers, modelling aspects such as data credibility requirements and bailiwick rules. In comparison, our model is more comprehensive, by considering the complete resolver algorithm at the design level.

IRONSIDES [14] is an authoritative nameserver implementation that is formally verified to be free of dataflow errors, runtime exceptions, and similar problems. While this makes it provably invulnerable to single-packet DoS attacks, it does not help against attacks at the protocol level, which is the focus of our automated attack analysis. In addition, the use of formal methods in IRONSIDES is unrelated to the DNS semantics, which means that it cannot serve as a formalization of the specification.

**Formal analysis of DoS attacks.** Formal analysis of a system's availability aspects is generally less mature in comparison to confidentiality or integrity properties. This is mainly due to the inherent quantitative nature of DoS attacks and the fact that the common Dolev-Yao intruder model [20] is too strong for this purpose. Meadows [35] introduced a cost-based framework for the analysis of DoS attacks where an attacker's power can be metered and limited. Urquiza et al. [46] refined this intruder model by explicitly considering timing aspects, which allows to capture more sophisticated attacks such as slow DoS and attacks targeting computational resources or memory. Our framework offers the same capabilities.

Amplification attacks have been analyzed by Shankesi et al. [43], where they leverage model checking to automate the search for DoS attacks on the VoIP session initiation protocol. Our framework also provides a built-in model checker. For DNS in particular, Deshpande et al. [19] modelled the classic bandwidth amplification attack as a Continuous Time Markov Chain to analyze different countermeasures. However, their highly abstract model cannot capture more sophisticated DoS attacks.

## 8 CONCLUSION AND FUTURE WORK

A formal framework is a necessary approach to achieve the desirable goal of a DNS infrastructure with strong security properties, especially given the intrinsic complexity of today's DNS specifications and configurations. Establishing a mathematically rigorous semantics can help identify and resolve ambiguities. The power of a formal framework has become apparent as it could automatically discover known and new attacks. The following directions are promising to pursue as future work: extend the model to cover DNSSEC and DoH/DoT, support multiple zones at a single nameserver, and to reduce the search space in our attack discovery approach.

## ETHICAL ISSUES STATEMENT

We have followed common practices for responsible disclosure of the new discovered DoS vulnerabilities. The affected entities have taken swift action, confirming the severity of the new issues.

## REFERENCES


[1] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. 2020. NXNSAttack: Recursive DNS Inefficiencies and Vulnerabilities. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 631–648. https://www.usenix.org/conference/usenixsecurity20/presentation/afek

[2] Gul Agha and Karl Palmskog. 2018. A Survey of Statistical Model Checking. *ACM Trans. Model. Comput. Simul.* 28, 1 (2018), 6:1–6:39. https://doi.org/10.1145/3158668

[3] Gul A. Agha. 1985. *Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence)*. Ph. D. Dissertation. University of Michigan, USA. http://hdl.handle.net/2027.42/160629

[4] Gul A. Agha, José Meseguer, and Koushik Sen. 2006. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *Electron. Notes Theor. Comput. Sci.* 153, 2 (2006), 213–239. https://doi.org/10.1016/j.entcs.2005.10.040

[5] Musab AlTurki and José Meseguer. 2011. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6859)*, Andrea Corradini, Bartek Klin, and Corina Cîrstea (Eds.). Springer, 386–392. https://doi.org/10.1007/978-3-642-22944-2_28

[6] Mark P. Andrews. 1998. Negative Caching of DNS Queries (DNS NCACHE). *RFC* 2308 (1998), 1–19. https://doi.org/10.17487/RFC2308

[7] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. 2005. Protocol Modifications for the DNS Security Extensions. *RFC* 4035 (2005), 1–53. https://doi.org/10.17487/RFC4035

[8] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC'10*. ACM, 267–280.

[9] Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. 2018. Survivability: design, formal modeling, and validation of cloud storage systems using Maude. *Assured cloud computing* (2018), 10–48.

[10] Stephane Bortzmeyer. 2016. DNS Query Name Minimisation to Improve Privacy. *RFC* 7816 (2016), 1–11. https://doi.org/10.17487/RFC7816

[11] Stephane Bortzmeyer, Ralph Dolmans, and Paul Hoffman. 2021. DNS Query Name Minimisation to Improve Privacy. *RFC* 9156 (2021), 1–11. https://doi.org/10.17487/RFC9156

[12] Stephane Bortzmeyer and Shumon Huque. 2016. NXDOMAIN: There Really Is Nothing Underneath. *RFC* 8020 (2016), 1–10. https://doi.org/10.17487/RFC8020

[13] Jonas Bushart and Christian Rossow. 2018. DNS Unchained: Amplified Application-Layer DoS Attacks Against DNS Authoritatives. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11050)*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer, 139–160. https://doi.org/10.1007/978-3-030-00470-5_7

[14] Martin C. Carlisle and Barry S. Fagin. 2012. IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities. In *2012 IEEE Global Communications Conference, GLOBECOM 2012, Anaheim, CA, USA, December 3-7, 2012*. IEEE, 839–844. https://doi.org/10.1109/GLOCOM.2012.6503217

[15] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. 2018. *Model checking, 2nd Edition*. MIT Press. https://mitpress.mit.edu/books/model-checking-second-edition

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[17] PowerDNS contributors. [n. d.]. DNS Camel. https://powerdns.org/dns-camel/. Accessed 2022-06-01.

[18] Wouter B. de Vries, Quirin Scheitle, Moritz Müller, Willem Toorop, Ralph Dolmans, and Roland van Rijswijk-Deij. 2019. A First Look at QNAME Minimization in the Domain Name System. In *Passive and Active Measurement - 20th International Conference, PAM 2019, Puerto Varas, Chile, March 27-29, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11419)*, David R. Choffnes and Marinho P. Barcellos (Eds.). Springer, 147–160. https://doi.org/10.1007/978-3-030-15986-3_10

[19] Tushar Deshpande, Panagiotis Katsaros, Stylianos Basagiannis, and Scott A. Smolka. 2011. Formal Analysis of the DNS Bandwidth Amplification Attack and Its Countermeasures Using Probabilistic Model Checking. In *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, Taghi M. Khoshgoftaar (Ed.). IEEE Computer Society, 360–367. https://doi.org/10.1109/HASE.2011.57

[20] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207. https://doi.org/10.1109/TIT.1983.1056650

[21] Robert Elz and Randy Bush. 1997. Clarifications to the DNS Specification. *RFC* 2181 (1997), 1–14. https://doi.org/10.17487/RFC2181

[22] Avishek Ghosh and Kannan Ramchandran. 2018. Faster Data-access in Large-scale Systems: Network-scale Latency Analysis under General Service-time Distributions. In *56th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2018, Monticello, IL, USA, October 2-5, 2018*. IEEE, 757–764.

[23] Joseph A. Goguen and José Meseguer. 1992. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theor. Comput. Sci.* 105, 2 (1992), 217–273. https://doi.org/10.1016/0304-3975(92)90302-V

[24] Suzanne Goldlust. [n. d.]. Understanding views in BIND 9, with examples. https://kb.isc.org/docs/aa-00851. Accessed 2022-07-15.

[25] Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects Comput.* 6, 5 (1994), 512–535. https://doi.org/10.1007/BF01211866

[26] Donald E. Eastlake III. 2012. xNAME RCODE and Status Bits Clarification. *RFC* 6604 (2012), 1–5. https://doi.org/10.17487/RFC6604

[27] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd D. Millstein, and George Varghese. 2020. GRooT: Proactive Verification of DNS Configurations. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, Henning Schulzrinne and Vishal Misra (Eds.). ACM, 310–328. https://doi.org/10.1145/3387514.3405871

[28] 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

[29] Edward P. Lewis. 2006. The Role of Wildcards in the Domain Name System. *RFC* 4592 (2006), 1–20. https://doi.org/10.17487/RFC4592

[30] Edward P. Lewis and Alfred Hoenes. 2010. DNS Zone Transfer Protocol (AXFR). *RFC* 5936 (2010), 1–29. https://doi.org/10.17487/RFC5936

[31] Si Liu. 2022. All in one: Design, verification, and implementation of SNOW-optimal read atomic transactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–44.

[32] Si Liu, José Meseguer, Peter Csaba Ölveczky, Min Zhang, and David A. Basin. 2022. Bridging the semantic gap between qualitative and quantitative models of distributed systems. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 315–344. https://doi.org/10.1145/3563299

[33] Si Liu, Peter Csaba Ölveczky, and José Meseguer. 2016. Modeling and analyzing mobile ad hoc networks in Real-Time Maude. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 34–66.

[34] Florian Maury. 2015. The "Indefinitely" Delegating Name Servers (iDNS) Attack. https://indico.dns-oarc.net/event/21/contributions/301/attachments/272/492/slides.pdf. Accessed 2022-04-30.

[35] Catherine A. Meadows. 2001. A Cost-Based Framework for Analysis of Denial of Service Networks. *J. Comput. Secur.* 9, 1/2 (2001), 143–164. https://doi.org/10.3233/jcs-2001-91-206

[36] José Meseguer. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science* 96, 1 (1992), 73–155.

[37] Paul V. Mockapetris. 1987. Domain names - concepts and facilities. *RFC* 1034 (1987), 1–55. https://doi.org/10.17487/RFC1034

[38] Paul V. Mockapetris. 1987. Domain names - implementation and specification. *RFC* 1035 (1987), 1–55. https://doi.org/10.17487/RFC1035

[39] Giovane C. M. Moura, Sebastian Castro, John S. Heidemann, and Wes Hardaker. 2021. TsuNAME: exploiting misconfiguration and vulnerability to DDoS DNS. In *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021*, Dave Levin, Alan Mislove, Johanna Amann, and Matthew Luckie (Eds.). ACM, 398–418. https://doi.org/10.1145/3487552.3487824

[40] Scott Rose and Wouter C. A. Wijngaards. 2012. DNAME Redirection in the DNS. *RFC* 6672 (2012), 1–22. https://doi.org/10.17487/RFC6672

[41] Christian Rossow. 2014. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society. https://www.ndss-symposium.org/ndss2014/amplification-hell-revisiting-network-protocols-ddos-abuse

[42] Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2005. On Statistical Model Checking of Stochastic Systems. In *CAV (LNCS, Vol. 3576)*. Springer.

[43] Ravinder Shankesi, Musab AlTurki, Ralf Sasse, Carl A. Gunter, and José Meseguer. 2009. Model-Checking DoS Amplification for VoIP Session Initiation. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5789)*, Michael Backes and Peng Ning (Eds.). Springer, 390–405. https://doi.org/10.1007/978-3-642-04444-1_24

[44] Sooel Son and Vitaly Shmatikov. 2010. The Hitchhiker's Guide to DNS Cache Poisoning. In *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 50)*, Sushil Jajodia and Jianying Zhou (Eds.). Springer, 466–483. https://doi.org/10.1007/978-3-642-16161-2_27

[45] Chuck Stearns, Vicky Risk, Suzanne Goldlust, and Cathy Almond. [n. d.]. BIND Best Practices - Recursive. https://kb.isc.org/docs/bind-best-practices-recursive. Accessed 2022-07-19.

[46] Abraão Aires Urquiza, Musab A. AlTurki, Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. 2019. Resource-Bounded Intruders in Denial of Service Attacks. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 382–396. https://doi.org/10.1109/CSF.2019.00033

[47] Paul Vixie, Susan Thomson, Yakov Rekhter, and Jim Bound. 1997. Dynamic Updates in the Domain Name System (DNS UPDATE). *RFC* 2136 (1997), 1–26. https://doi.org/10.17487/RFC2136

[48] Anduo Wang, Alexander J. T. Gurney, Xianglong Han, Jinyan Cao, Boon Thau Loo, Carolyn L. Talcott, and Andre Scedrov. 2014. A reduction-based approach towards scaling up formal analysis of internet configurations. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. IEEE, 637–645. https://doi.org/10.1109/INFOCOM.2014.6847989

[49] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1439–1453. https://doi.org/10.1145/3319535.3345662

[50] Zheng Wang. 2018. Understanding the Performance and Challenges of DNS Query Name Minimization. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*. IEEE, 1115–1120. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00155

[51] Thilo Weghorn, Si Liu, Christoph Sprenger, Adrian Perrig, and David Basin. 2022. N-Tube: Formally Verified Secure

Bandwidth Reservation in Path-Aware Internet Architectures. In *CSF 2022*. IEEE. https://doi.org/10.5281/zenodo.5856306

[52] Wikipedia. [n. d.]. 2021 Facebook outage. https://en.wikipedia.org/wiki/2021_Facebook_outage.

# A FRAMEWORK DETAILS

## A.1 Random Zone Generation

An essential component of our initial state generator is the generation of random zones. We create a random zone using the following sequence of steps.

(1) Choose the underlying tree uniformly at random among all non-isomorphic rooted unlabelled trees with the given number of nodes.
(2) Choose labels for each node of the tree, ensuring uniqueness among sibling nodes. At the moment, we do not generate wildcard labels, although this could easily be changed.
(3) For each node, choose which RRsets are present. Note that the allowed combinations depend on the type of node, e.g., a non-terminal node cannot have a DNAME record, and the apex cannot have a CNAME.
(4) For each RRset, choose its size (i.e., the number of records). For many types, the number of records is either limited to one (CNAME, DNAME) or does not influence resolution apart from the contents of the final answer (TXT, A, etc.). However, for NS RRsets, the number of records is highly relevant because the resolver may have to create subqueries to resolve unknown addresses. Thus, we choose the size of NS RRsets among the (arbitrary) candidates 1, 2, 5, or 10.
(5) For records that have names as values (CNAME, DNAME, NS), choose among the names appearing elsewhere (e.g., in the benign zones, other malicious zones, or even the zone itself). Note that one might also wish to allow *prefixes* of existing names, even if they do not have any RRsets associated. Similarly, records can point to non-existent names, which are created by adding a number of non-existent labels to an existing name.[8] The values of address or TXT records are filled with the dummy values `1.2.3.4` or `'...'`, respectively. This step is retried until the zone is free of pathological errors such as in-zone CNAME loops.
(6) Finally, add the mandatory SOA and NS records to the zone, as well as the address record for the authoritative nameserver.

[8]This existing name can be the root label, thus any non-existent name is possible.

## A.2 Monitor Specification

In our model the monitor is an actor just like nameservers or resolvers, with different attributes that keep track of relevant information. Note that there will only be *one* monitor in the configuration.

```
op Monitor : -> ActorType .

op clQueryLog:_ : TupleAddrQueryList ->
    Attribute .
op clRespLog:_ : TupleAddrRespList -> Attribute
    .
...
```

Figure 9 exemplifies the monitoring mechanism using a rewrite rule. The monitor is specified just like nameservers or resolvers with two example attributes: `clQueryLog` keeps track of the queries sent by clients, while `clRespLog` records the responses received by them. Note that we use custom data structures to store the information, in this case lists of *address-query* or *address-response* pairs. The response that the client receives is added to `clRespLog` on line 8 while the new query sent to the resolver to `clQueryLog` on line 7.

```
--- zone records
< example.com., SOA, soaData(testTTL) >
< example.com., NS, ns1.example.com. >
< example.com., NS, ns2.example.com. >

--- nameserver addresses
< ns1.example.com., A, addrNS1example >
< ns2.example.com., A, addrNS2example >

--- other records
< www.example.com., A, 1.2.3.4 >
< alias.example.com., CNAME, www.example.com. >
```

```
--- other records are identical
< alias.example.com., CNAME, nxdomain.example.
    com. >
```

## A.3 Predefined Property Library

Table 4 lists our predefined properties that interesting in common scenarios. We explain the definition of one correctness property below.

**Rewrite Blackholing.** Recall that the monitor keeps track of the responses received by clients in the attribute `clRespLog` (Figure 9). Hence, we can simply check if there is a response in `clRespLog` that has a non-empty answer section and a NXDOMAIN response code. As shown in Figure 10, the state predicate `hasRewriteBlackhole` is a simple wrapper that extracts the value of the monitor's `clRespLog` attribute (from the entire system state) (lines 2–3) and passes it to the predicate `rewriteBlackhole`

```
rl [client-recv-resp-send-next] :
  < MON : Monitor | clQueryLog: TAQL, clRespLog: TARL, STATE' >
  < CL : Client | queries: (query(ID', QNAME, QTYPE) QS),  resolver: RSV, STATE >
  {T, to CL from RSV : response(ID, NAME, ANS, AUTH, ADD, RCODE)}
=>
  < MON : Monitor |
      clQueryLog: tupleAddrQuery(CL, query(ID', QNAME, QTYPE)) TAQL,
      clRespLog: tupleAddrResp(CL, response(ID,NAME, ANS, AUTH, ADD, RCODE)) TARL,STATE' >
  < CL : Client | queries: QS, resolver: RSV, STATE >
  [delay, to RSV from CL : query(ID', QNAME, QTYPE), drop?] .
```

**Figure 9: Example rule for the monitoring mechanism**

| # | Property | Description | Comment |
|---|---|---|---|
| 1 | Delegation Inconsistency | The NS or A records in a referral response differ from the records in an authoritative response. | |
| 2 | Lame Delegation | The resolver is referred to a nameserver that is not authoritative for the zone and cannot provide a referral. | We check for REFUSED responses. Note that a referral "back up" in the hierarchy is not considered a lame delegation here. |
| 3 | Glueless Delegation | NS records in a referral response are not accompanied by glue records. | Note that this checks for any delegation with missing glue records, not for missing *required* glue. |
| 4 | Non-Existent Domain | The query is answered with NXDOMAIN. | |
| 5 | Cyclic Zone Dependency | The delegations involved in the query's resolution are cyclically dependent. | |
| 6 | Rewrite Loop | The query is rewritten to itself. | |
| 7 | Domain Overflow | The query name at some point exceeds the maximum domain length. | This can happen due to DNAME substitution. |
| 8 | Answer Inconsistency | The query can produce different answers. | This can happen due to inconsistent zone files. |
| 9 | Zero TTL | Resolution of the query involves records with a TTL of 0. | |
| 10 | Rewrite Blackholing | The query is rewritten to a name that does not exist. | |
| 11 | Repeated Query | The same query is sent to the same nameserver multiple times during resolution of a client query. | Captures circular dependencies, QNAME minimization inefficiencies, or insufficient TTLs |
| 12 | Domain Overflow at Nameserver | The nameserver sends a YXDOMAIN response because of a domain overflow after DNAME substitution. | This captures only the overflows at *nameservers*, but not those at *resolvers* (due to cached DNAMEs). |
| 13 | Inconsistent RRsets | Different nameservers have inconsistent RRsets for the same name and type. | This static property can be checked without considerin |
| 14 | Amplification Factor | the number of messages received by victim divided by the nuber of messages sent by attacker | |
| 15 | Query Duration | How long the resolution of a client query takes | |
| 16 | Query Success Ratio | The probability for a client query to be answered | |

**Table 4: Our library of predefined properties.**

```
1  *** extract partial info, i.e., clRespLog, from the monitor, thus the entire system state
2  op hasRewriteBlackhole : Config -> Bool .
3  eq hasRewriteBlackhole(< ADDR : Monitor | clRespLog: TARL, STATE > C) = rewriteBlackhole(TARL) .
4
5  *** define rewrite blackholing
6  op rewriteBlackhole : TupleAddrRespList -> Bool .
7  eq rewriteBlackhole(tupleAddrResp(ADDR, response(ID, NAME, ANS, AUTH, ADD, RCODE)) TARL)
8   = if ANS =/= nil and RCODE == 3 then true else rewriteBlackhole(TARL) fi .  --- 3: NXDOMAIN
9  eq rewriteBlackhole(nilTARL) = false .
10
11 *** atomic proposition in LTL
12 op propRewriteBlackhole : -> Prop .    eq C |= propRewriteBlackhole = hasRewriteBlackhole(C) .
13
14 *** command for invoking LTL model checking
15 red modelCheck(initConfig, [] ~ propRewriteBlackhole) .
```

**Figure 10: LTL model checking of the rewrite blackholing property.**

which then checks all entries for a non-empty answer section and a NXDOMAIN RCODE (lines 6–9). The atomic proposition in LTL can then be specified in Maude syntax (line 12), where `|=` defines the satisfaction relation with respect to the state predicate `hasRewriteBlackhole`. Finally, we define the LTL formula:
`[] ~ propRewriteBlackhole`,[9] meaning that the rewrite blackhole property is never ("always not") satisfied, before we invoke the LTL model checker (line 15).

In one of our model checking analyses, with the misconfigured zone files for two nameservers the model checker finds the following counterexample, showing that the query from the initial configuration (`initConfig`) can encounter a rewrite blackhole.

# B AMBIGUITY RESOLUTION

## B.1 Resolver Algorithm Clarifications

In the absence of QMIN, the resolver algorithm given in RFC 6672 [40, §3.4.1] is the most refined one. It is shown in listing 1. On a high level, it works as follows: When the recursive resolver receives a client query, it first tries to answer the query from cache. If this is not possible, it creates internal state for the query, which includes identifying the best known nameservers. The resolver then sends the query to one of these nameservers and awaits a response. Once a response arrives, a number of cases are possible: In the best case, the response allows to immediately answer the client query. Otherwise, the response may refer the resolver to a different nameserver, or indicate that the query must be rewritten due to a CNAME or DNAME. In the latter two cases, the resolver updates the query state according to the new information and sends the (updated) query to a new nameserver.

[9] `[]` and `~` are Maude syntax for the "always" operator □ and the "not" operator ¬ in traditional LTL notation [15], respectively.

In addition, there are a number of error cases that must be handled: The nameserver may not respond at all (e.g., because the message was dropped), refuse to answer the query (e.g., because it is not authoritative for the query and cannot provide a delegation), or the resolver may reach a configurable *work limit*, at which point it should abort the query. Of course, the resolver works on many client queries concurrently, and thus needs to maintain state for each.

An important notion is that of *resolver subqueries*, by which we mean queries that the resolver creates itself. These subqueries are necessary when the resolver knows the *names* of nameservers for a query, but not their addresses.

In particular, this means that "return[ing the data] to the client" in case A of step 4 of the resolver algorithm can mean two different things: In case of a client query, the resolver sends a DNS response over the network to the client. For resolver subqueries (where "client" refers to the resolver itself), this is not necessary. Instead, the resolver can directly *use* the data to update the states of the pending queries.

Unfortunately, the resolver algorithm contains a number of ambiguities that need to be resolved before being able to formalize the semantics:

(1) The phrasing of step 4 ("analyze the response, either: . . . ", cf. listing 1) suggests that a response matches exactly one of the given cases. However, some types of responses match the description of multiple cases: For example, a response may contain both a CNAME/DNAME record in the answer section and a referral (for the canonical name) in the authority section, i.e., both cases C/D and B apply.
(2) Similarly, a response may contain a CNAME/DNAME record *and* data for the canonical name that allows

```
1.  See if the answer is in local information or can be synthesized from a cached DNAME; if so,
      return it to the client.
2.  Find the best servers to ask.
3.  Send queries until one returns a response.
4.  Analyze the response, either:
  A.  If the response answers the question or contains a name error, cache the data as well as
        return it back to the client.
  B.  If the response contains a better delegation to other servers, cache the delegation
        information, and go to step 2.
  C.  If the response shows a CNAME and that is not the answer itself, cache the CNAME, change
        the SNAME to the canonical name in the CNAME RR, and go to step 1.
  D.  If the response shows a DNAME and that is not the answer itself, cache the DNAME (upon
        successful DNSSEC validation if the client is a validating resolver).  If substitution of
        the DNAME's target name for its owner name in the SNAME would overflow the legal size for
        a domain name, return an implementation-dependent error to the application; otherwise,
        perform the substitution and go to step 1.
  E.  If the response shows a server failure or other bizarre contents, delete the server from
        the SLIST and go back to step 3.
```

**Listing 1: The most refined resolver algorithm from RFC 6672.**

to "answer the question", i.e., both cases C/D and A apply.

(3) Case A should apply "if the response answers the question", which could be interpreted to cover only *authoritative* answers or any response containing data that is sufficient to answer the question.

(4) Finally, while the resolver algorithm is unambiguous in when the cache should be checked, it appears to largely ignore the fact that the cache may be updated by concurrent queries. An explicit cache lookup is only mentioned when this could provide a benefit even for a sequential resolver. For example, the cache is checked again after query rewriting due to a CNAME or DNAME response, but not after a referral.

We resolve these ambiguities by interpreting the different cases in such a way that any response can only match *one* case. To this end, we impose that case A applies only for authoritative responses that *directly* answer the query (i.e., without query rewriting, and looking only at the answer section). Note that a referral response never matches case A, even if the data contained in the authority and additional sections suffice to "answer the question" (taking into account the data credibility rules). Also note that the cache is *not* checked.

Furthermore, we disambiguate cases B and C/D by restricting case B to *pure* referral responses that apply directly to the QNAME in the query, without any query rewriting. A response to a query with a QTYPE that does not match CNAME or DNAME but with CNAMEs or DNAMEs in the answer section *always* matches one of the cases C or D. In other words, a CNAME/DNAME response with a referral for the rewritten name does *not* match case B. Similarly, a CNAME/DNAME response with data for the rewritten name never matches case A because this data is not authoritative, even if it is sufficient to "answer the question".

Regarding the cache lookups, we adopt a literal interpretation, i.e., the cache is only checked in step 1 of the algorithm.

## B.2 Ambiguities in QMIN

We now address the ambiguities in the QNAME minimization algorithm, which substantially changes a resolver's behavior. It may change both the QNAME and QTYPE that the resolver sends to authoritative nameservers. We now address a number of subtle points regarding these changes. RFC 9156 [11] contains an extended resolver algorithm using QNAME minimization, which we show in listing 2. We omit small parts that are only relevant for types used in DNSSEC or for nameservers that are not compliant with RFC 8020 [12]. Even though this algorithm is already a refined version of an earlier version [10], it still suffers from a number of problems, which we address in the following.

**Avoiding unnecessary final queries.** Recall that obfuscating the original QTYPE by querying for a different type may introduce an additional query once the authoritative nameserver for a name has been discovered. To minimize the additional work that this introduces, RFC 9156 suggests to use the most common QTYPE for obfuscation, which saves the extra query in the end for many queries. However, the omission of this final query is not clearly reflected in the algorithm. In particular, a suitable final response received for the full QNAME would

```
(0) If the query can be answered from the cache, do so; otherwise, iterate as follows:

(1) Get the closest delegation point that can be used for the original QNAME from the cache.

    (1a) [...]
    (1b) For queries with other original QTYPEs, this is the NS RRset with the owner matching
         the most labels with QNAME. QNAME will be equal to or a subdomain of this NS RRset.
         Call this ANCESTOR.

(2) Initialise CHILD to the same as ANCESTOR.

(3) If CHILD is the same as QNAME [...], resolve the original query as normal, starting from
    ANCESTOR's name servers. Start over from step 0 if new names need to be resolved as a result of
     this answer, for example, when the answer contains a CNAME or DNAME [RFC6672] record.

(4) Otherwise, update the value of CHILD by adding the next relevant label or labels from QNAME to
    the start of CHILD. The number of labels to add is discussed in Section 2.3.

(5) Look for a cache entry for the RRset at CHILD with the original QTYPE. If the cached response
    code is NXDOMAIN [...], the NXDOMAIN can be used in response to the original query, and stop.
    If the cached response code is NOERROR (including NODATA), go back to step 3. [...]

(6) Query for CHILD with the selected QTYPE using one of ANCESTOR's name servers. The response can
    be:

    (6a) A referral. Cache the NS RRset from the authority section, and go back to step 1.
    (6b) A DNAME response. Proceed as if a DNAME is received for the original query. Start over
         from step 0 to resolve the new name based on the DNAME target.
    (6c) All other NOERROR answers (including NODATA). Cache this answer. Regardless of the
         answered RRset type, including CNAMEs, continue with the algorithm from step 3 by building
         the original QNAME.
    (6d) An NXDOMAIN response. [...] Return an NXDOMAIN response to the original query, and stop.
         [...]
    (6e) A timeout or response with another RCODE. The implementation may choose to retry step 6
         with a different ANCESTOR name server.
```

**Listing 2: The QNAME minimization algorithm, with some parts omitted that are only relevant for DNSSEC-specific types or nameservers that are not compliant with RFC 8020.**

match case (6c), which states to "continue with the algorithm from step 3 by building the original QNAME". In step 3, the condition is satisfied, i.e., CHILD is the same as QNAME, and we thus need to "resolve the original query as normal, starting from ANCESTOR's name servers". This wording suggests that we directly query the nameservers for ANCESTOR, which would lead to an unnecessary query. Instead, we should first check the cache, and only query ANCESTOR's nameservers in case of a miss.[10]

Whether or not the cache is checked in step (3) also has consequences for the handling of CNAME responses. It is clear that a CNAME received for an incomplete QNAME must not be followed, as indicated in case (6c). However, if the full QNAME was queried, the CNAME can be followed safely *regardless* of the query *type*. However, this optimization is only performed when the cache is checked in step (3).

As a side remark, note that the "referral" response in case (6a) only means *pure* referrals for the queried name, i.e., without any query rewriting applied. For example, a response containing a CNAME for an incomplete QNAME and a referral for the canonical name should not match case (6a).

**Limiting the number of iterations.** Recall that the process of discovering the authoritative nameserver for a name may involve querying the same nameserver multiple times, with one more label added in each iteration. Clearly, the resolver must enforce some limits on the number of extra iterations performed for this. The mechanism suggested in RFC 9156 [11, §2.3] involves two configuration parameters, called MAX_MINIMISE_COUNT and MINIMISE_ONE_LAB. The first one is the maximum

[10] One might argue that "resolving the original query as normal" implicitly includes a cache lookup; we still think this should be made explicit.

number of extra iterations the resolver performs for a single name, *excluding* the possible additional query for the original QTYPE. To enforce this limit, the resolver splits the total number of labels with unknown zone cuts evenly among the remaining iterations, and may thus add more than one label at a time. The second parameter, which must be strictly smaller than the first one, is the number of iterations where *only one* label should be added, no matter how many labels follow. The motivation is that the largest privacy gains can typically be achieved with respect to nameservers higher up in the hierarchy (e.g., root servers and top-level domain servers). The values recommended in the RFC are 10 for MAX_MINIMISE_COUNT and 4 for the number of guaranteed one-label iterations.

An important point (that is not made explicit in the RFC) is that the maximum number of iterations is enforced *per name*. When the QNAME is rewritten due to CNAMEs or DNAMEs, the limits are reset.[11] In combination with long rewrite chains, the number of extra iterations introduced by QNAME minimization can thus be much larger than MAX_MINIMISE_COUNT, an attack vector that appears to have been overlooked so far and which we exploit in different ways in 6.

**Finding zone cuts (step 5).** Finally, we discuss step (5) of the algorithm. The purpose of this step is to retrieve information on the absence of zone cuts from the cache, which would allow to safely add additional labels without risking revealing unnecessary information. However, there are a number of logical flaws in the specification:

(1) If the resolver had a NXDOMAIN cached for an ancestor of QNAME, it would never reach step (5) of the algorithm. Instead, it would use this cached NXDOMAIN to directly answer the query in step (0).
(2) The DNS specification does not mandate (nor suggest) that a resolver stores zone cut information along with positive authoritative answers. Thus, the positive cache does not convey any information on the *absence* of zone cuts. In particular, it may be the case that CHILD lies in a different zone than ANCESTOR, but the NS records for the zone containing CHILD have expired. Note that there may still be other records from the zone containing CHILD in the cache, either because they have a longer TTL than the NS records or because they were introduced into cache later.
(3) Entries in the *negative* cache, on the other hand, do convey information about the absence of zone cuts (through the associated SOA records). For example, if there is a NODATA cache entry for dept.example.com, and the associated SOA record is for the example.com zone, the resolver can conclude that dept.example.com is not delegated and can indeed go back to step (3) to add more labels.
(4) It is unclear why step (5) mandates to look for a cache entry for the RRSET at CHILD with the *original* QTYPE. As argued above, the positive cache does not help. For the NODATA cache, an entry for *any* QTYPE will provide information on the zone in which the CHILD is contained.

We address these issues by proposing (and later formalizing) our own version of step (5):

```
(5) Look for a NODATA cache entry at CHILD for
    any QTYPE. If there is a hit and the owner
    name of the associated SOA record is equal
    to ANCESTOR, go back to step (3).
```

**Listing 3: Proposed fixed step (5) of the QNAME minimization algorithm.**

# C MORE DETAILS ON AUTOMATED ATTACK ANALYSIS

## C.1 Unsoundness of GRoot ECs

We show that the GRoot's equivalence classes (EC) is *unsound* with respect to our model, or more generally, with respect to any DNS semantics that considers synthesized records.[12] This means even if we use it as to reduce search space in our automated attack approach, there is no guarantee that our analysis in *complete* in the sense that we can find all possible attacks.

To illustrate GRoot EC's unsoundness, we show two queries that are in the same equivalence class, but have different resolution behavior. Consider two zones, example.com and example.net. The authoritative nameservers are ns.example.com and ns.example.net, respectively. The zone for example.com contains the mandatory SOA, NS and address records, as well as a wildcard CNAME record.

```
< example.com, SOA, ... >
< example.com, NS, ns.example.com >
< ns.example.com, A, ... >

< *.example.com, CNAME, a.dname.example.net >
```

[11] While it is conceivable that the limits are not reset upon query rewriting, this would inevitably sacrifice all privacy guarantees for the rewritten name because the resolver cannot allocate iterations to the labels in a yet-unknown rewritten name.

[12] While the GRoot semantics does consider synthesized records, the authors argue that these records can be ignored for the notion of equivalence because the GRoot semantics does not consider caching.

The zone for example.net contains the mandatory records plus a DNAME record, rewriting any domain below dname.example.net to the corresponding domain below example.com:

```
< example.net, SOA, ... >
< example.net, NS, ns.example.net >
< ns.example.net, A, ... >

< dname.example.net, DNAME, example.com >
```

Consider now two queries for the same type with QNAMEs a.example.com and b.example.com. Neither of the names appear in the zone files, thus both queries are contained in the equivalence class $\alpha$.example.com for that type.

Let us first consider the resolution of a.example.com, starting from an empty cache. After following some delegations, the recursive resolver sends the query to the authoritative nameserver for the example.com zone, where it will match the wildcard CNAME record. The resolver thus receives the (synthesized) CNAME `< a.example.com, CNAME, a.dname.example.net >` and rewrites the query to the canonical name (i.e., it changes the SNAME to a.dname.example.net). After following a few referrals for the rewritten name, it sends the query to the authoritative nameserver for the example.net zone, where it will match the DNAME record. The resolver receives a response indicating that a.dname.example.net should be rewritten to a.example.com. At this point, the resolver detects that there is a rewrite loop and signals an error to the client.

Now consider the second query, b.example.com, again starting from an empty cache. When the resolver sends this query to the authoritative nameserver for the example.com zone, it receives a slightly different CNAME record synthesized from the wildcard, namely `< b.example.com, CNAME, a.dname.example.net >`. The resolver again rewrites the query to a.dname.example.net and (eventually) sends this query to the authoritative nameserver for example.net, where it matches the DNAME record as before. However, this time, the DNAME response (indicating that a.dname.example.net should be rewritten to a.example.com) does *not* yet allow the resolver to detect the rewrite loop because it does not know anything about a.example.com. Instead, it has to send another query for that name, and only then will be able to detect the rewrite loop. Clearly, the two queries are not resolved in the same way, and should thus not be in the same equivalence class. We can thus conclude that the GRoot ECs are unsound with respect to our model.

## C.2 Testbed for Attack Validation

For the validation of our new attacks (Section 6), we used a local DNS testbed consisting of different virtual machines (VMs) running Ubuntu Server 20.04 LTS.

One VM provides authoritative nameserver services for the entire DNS hierarchy, i.e., for a (dummy) root zone, `com`, `net`, `example.com`, and `example.net` zones. We used the BIND implementation available through the default package manager on Ubuntu. To simulate multiple authoritative nameservers (each authoritative for only one zone), we made use of BIND *views* [24], which allow us to isolate the different zones from each other and invoke a particular view depending on the destination address of the query.

The other VMs each run one resolver implementation. We configured each resolver with a custom root hints file pointing to the nameserver VM such that the resolvers bootstrap resolution from the dummy root zone. All VMs are connected to each other and the host machine using the *host-only networking* mode provided by VirtualBox. This allows, e.g., to use a tool such as `dig` on the host machine to send the client queries to a resolver VM, which then contacts the nameserver VM.

To simulate delayed responses from malicious nameservers, we made use of the network emulation functionality provided by `netem`.

## C.3 Malicious Zones for New Attacks

In this section, we present example malicious zones generated by our tool during automated attack analysis.

**Subqueries + Unchained.** The attack is triggered by a query for `kk.q.atk2.net`. Upon inspection of its NS RRset, we see that `zz.a.atk1.com` starts an Unchained-style CNAME chain of length 4, and all of `bbbb.rrr.zz.a.atk1.com`, `rrr.zz.a.atk1.com`, and `vvv.tt.a.atk1.com` start chains of length 2 (or a loop in one case).

```
--- zone atk1.com
< atk1.com., A, 1.2.3.4 >
< atk1.com., SOA, 3600 >
< atk1.com., NS, ns.atk1.com. >
< a.atk1.com., CNAME, jjjj.ggg.pp.q.atk2.net. >
< zz.a.atk1.com., CNAME, ggg.pp.q.atk2.net. >
< tt.a.atk1.com., CNAME, ns.atk2.net. >
< rrr.zz.a.atk1.com., CNAME, q.atk2.net. >
< nnn.tt.a.atk1.com., NS, llll.ggg.pp.q.atk2.
    net. >
< nnn.tt.a.atk1.com., NS, xxxx.ggg.pp.q.atk2.
    net. >
< nnn.tt.a.atk1.com., NS, q.atk2.net. >
< nnn.tt.a.atk1.com., NS, ns.atk2.net. >
< nnn.tt.a.atk1.com., NS, pp.q.atk2.net. >
```

```
< vvv.tt.a.atk1.com., CNAME, jjjj.ggg.pp.q.atk2
    .net. >
< bbbb.rrr.zz.a.atk1.com., CNAME, xxxx.ggg.pp.q
    .atk2.net. >
< ns.atk1.com., A, addrNsatk1Com >

--- zone atk2.net
< atk2.net., A, 1.2.3.4 >
< atk2.net., SOA, 3600 >
< atk2.net., NS, ns.atk2.net. >
< q.atk2.net., CNAME, rrr.zz.a.atk1.com. >
< pp.q.atk2.net., CNAME, ns.atk1.com. >
< kk.q.atk2.net., NS, bbbb.rrr.zz.a.atk1.com. >
< kk.q.atk2.net., NS, rrr.zz.a.atk1.com. >
< kk.q.atk2.net., NS, vvv.tt.a.atk1.com. >
< kk.q.atk2.net., NS, atk1.com. >
< kk.q.atk2.net., NS, zz.a.atk1.com. >
< ggg.pp.q.atk2.net., CNAME, bbbb.rrr.zz.a.atk1
    .com. >
< xxxx.ggg.pp.q.atk2.net., CNAME, ns.atk1.com.
    >
< llll.ggg.pp.q.atk2.net., NS, zz.a.atk1.com. >
< llll.ggg.pp.q.atk2.net., NS, a.atk1.com. >
< llll.ggg.pp.q.atk2.net., NS, tt.a.atk1.com. >
< llll.ggg.pp.q.atk2.net., NS, rrr.zz.a.atk1.
    com. >
< llll.ggg.pp.q.atk2.net., NS, vvv.tt.a.atk1.
    com. >
< jjjj.ggg.pp.q.atk2.net., CNAME, ns.atk1.com.
    >
< ns.atk2.net., A, addrNsatk2Net >
```

**Subqueries + CNAME Scrubbing.** The attack is triggered by a query for `i.atk1.com`. Each of its NS names starts a CNAME chain, all within the same zone.

```
--- zone atk1.com
< atk1.com., A, 1.2.3.4 >
< atk1.com., SOA, 3600 >
< atk1.com., NS, ns.atk1.com. >
< atk1.com., CNAME, ns.attacker2.com. >
< ee.e.atk1.com., CNAME, vvv.tt.e.atk1.com. >
< tt.e.atk1.com., CNAME, atk1.com. >
< bbb.ee.e.atk1.com., CNAME, atk1.com. >
< xxxx.ppp.cc.r.atk1.com., CNAME, b.atk1.com. >
< ooo.ee.e.atk1.com., CNAME, ee.e.atk1.com. >
< aaa.tt.e.atk1.com., CNAME, bbb.ee.e.atk1.com.
    >
< vvv.tt.e.atk1.com., CNAME, nn.k.atk1.com. >
< b.atk1.com., CNAME qqq.ss.l.atk1.com. >
< x.atk1.com., CNAME, xxxx.ppp.cc.r.atk1.com. >
< aa.k.atk1.com., CNAME u.atk1.com. >
< i.atk1.com., NS x.atk1.com. >
< i.atk1.com., NS ooo.ee.e.atk1.com. >
< ns.atk1.com., A, addrNsatk1Com >
```

**CNAME Scrubbing + QMIN.** The attack is triggered by a query for `bbbb.ccc.vv.k.atk1.com`, which starts a CNAME chain. Each name on the chain has a few more labels than the zone apex `atk1.com` and therefore incurs multiple QMIN probing queries.

```
--- zone atk1.com
< atk1.com., A, 1.2.3.4 >
< atk1.com., SOA, 3600 >
< atk1.com., NS, ns.atk1.com. >
< gggg.iii.uu.b.atk1.com., CNAME, aaaa.vvv.jj.z
    .atk1.com. >
< bbbb.ccc.vv.k.atk1.com., CNAME, nnnn.aaa.pp.v
    .atk1.com. >
< dddd.ppp.mm.s.atk1.com., CNAME, qqqq.bbb.hh.v
    .atk1.com. >
< wwww.jjj.qq.m.atk1.com., CNAME, mmmm.ccc.xx.k
    .atk1.com. >
< vvvv.yyy.ii.x.atk1.com., CNAME, iiii.ooo.kk.l
    .atk1.com. >
< jjjj.qqq.dd.w.atk1.com., CNAME, vvvv.yyy.ii.x
    .atk1.com. >
< nnnn.aaa.pp.v.atk1.com., CNAME, jjjj.qqq.dd.w
    .atk1.com. >
< ns.atk1.com., A, addrNsAttacke r1Net >
```

**Slow DoS Attacks.** These attacks use the same malicious zones as attacks exploiting CNAME scrubbing and QMIN.

## D DOMAIN NAME SYSTEM (DNS)

The Domain Name System (DNS) is a scalable, decentralized database mapping names to IP addresses and other information. Almost all accesses to resources on the Internet start with a DNS request. For example, when a user visits a website such as `www.example.com`, the browser first issues a DNS request to obtain the address of the server hosting that website.

**DNS Namespace.** The DNS namespace is organized in a hierarchical tree structure. Each node in this tree has a label, and labels are concatenated with a '.' character. A fully qualified domain name uniquely identifies a node in the tree and consists of the labels on the path from the node to the root. The root node has an empty label, which is often omitted (together with the preceding dot character). For example, `www.example.com` is the short form for `www.example.com.`, where the root label is made explicit.

Different entities are authoritative for different parts of the namespace. These parts are called *zones* and represent subtrees of the full DNS tree. Authority over a subtree is *delegated* by the parent zone to the child zone. Figure 11 shows a small portion of the DNS namespace, consisting of the root, the `com` and `net` top-level zones, and the second-level zones `example.com` and `example.net`. The delegation points (or *zone cuts*) are marked with dashed red lines. The names directly below a zone cut are called the *apex* of the subtree rooted there. In the example, the root zone delegates authority over the subtrees rooted at `com` and `net`. The `com` zone again delegates authority over the subtree rooted at `example.com`.

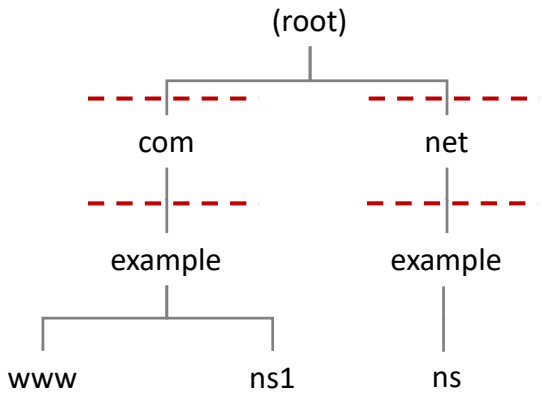


**Figure 11: A small portion of the DNS namespace, with dashed red lines marking zone cuts.**

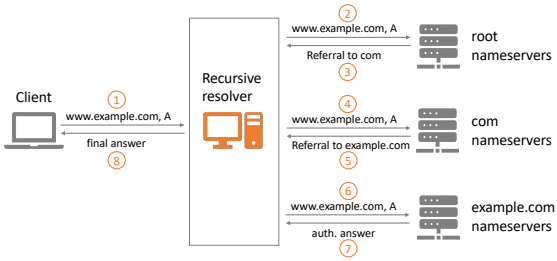


**Figure 12: Resolution of the type a query for `www.example.com`, starting from an empty cache at the resolver.**

However, all of `example.com`, `www.example.com`, and `ns1.example.com` are contained in the `example.com` zone.

**Resource Records.** The actual data is stored in so-called *resource records* (RRs). A RR consists of an *owner name* (the fully qualified domain name of the tree node where this RR resides), a *type*, a *time-to-live* field (TTL), and a *value*, whose structure depends on the type of the record. In general, multiple records with the same owner name and type can exist, as long as they have different data. Such records form an *RRset*.

Some record types are used purely for application data, e.g., a TXT record has arbitrary ASCII text as its value. Other record types, such as SOA or NS, store meta-level information that is necessary to make DNS work. Finally, some record types serve both purposes, e.g., an A record can store the IPv4 address of a web server (application data) or a nameserver (meta-level data).

## D.1 Basic DNS Resolution

When a client such as a web browser makes a DNS request, it typically sends the query to a *recursive resolver*, which performs the resolution on behalf of the client. Recursive resolvers are usually operated by ISPs or other entities such as Google. The recursive resolver may already have the answer in its cache from a previous query, in which case it can immediately respond to the client. Otherwise, it contacts a hierarchy of *authoritative nameservers* to obtain the answer.

Each authoritative nameserver only serves data for a small number of zones. For example, there are a few (widely distributed via IP anycast) root servers that only serve the root zone. Similarly, there are nameservers configured to serve the `com` and `net` zones.

**Example.** We illustrate DNS resolution with an example. Assume that a client wants to resolve the address of `www.example.com`, and hence sends a type A query to its recursive resolver. This is shown as step 1 in figure 12. Assuming that the recursive resolver has an empty cache, it now sends this query to one of the root nameservers (step 2). Note that a resolver must always know the address of at least one root nameserver, otherwise it cannot bootstrap resolution. These root server addresses are configured statically by the resolver operator.

Since the root nameserver is only authoritative for the root zone, it cannot answer the query for `www.example.com`. Instead, it provides a *delegation* (or *referral*) to nameservers that are "closer" to the authoritative data. In this case, it refers the resolver to the authoritative nameservers of the `com` zone, i.e., it sends the NS records for that zone along with the addresses of the `com` nameservers (step 3).

The resolver then sends the query to one of the `com` nameservers. Again, these are not authoritative for the full domain, but instead refer the resolver to the authoritative nameservers for `example.com` (steps 4 and 5). The resolver sends the query to these nameservers, and finally obtains an authoritative answer (steps 6 and 7), which it can send back to the client (step 8).

If the same query needs to be resolved again at a later point, the resolver does not need to contact the nameservers, but can instead answer the query directly from its cache, assuming that the TTL of the records has not expired. Similarly, if the resolver receives a query for a different name under a known zone, it can query directly the best known nameservers instead of starting at the root.

A response may also indicate that the requested data does not exist, where we distinguish two cases: A NXDOMAIN response indicates that the *name* does not exist, which means that there are no resource records (of any type) at this name or at any name below. A NODATA response, on the other hand, means that the name exists but there is no *data* of the requested type.[13]

**Glue records and resolver subqueries.** Note that for resolution to work, the resolver needs to somehow learn the addresses of the nameservers it wishes to contact. As mentioned above, the root server addresses are typically part of the static resolver configuration. However, for all lower-level nameservers, the addresses must be learned during DNS resolution. Typically, this is achieved by including the address records of nameservers in the referral response. These address records are called *glue*.

In the example above, we assumed that all referrals have glue. In fact, in many cases, glue records are *necessary* for resolution to work. To see this, assume that the nameserver for `example.com` is located at `ns1.example.com`. If the `com` server sent a referral to `ns1.example.com` without any addresses, the resolver would have to look up the address of `ns1.example.com` itself. However, the best thing it could do is to ask the `com` nameserver again, which would simply send another glueless referral. To prevent the resolver from getting stuck in such a situation, a delegation to a nameserver in or below the delegated zone *must* include glue records.

If the nameserver is located in a different part of the namespace, glue is not required. For example, assume that the authoritative nameserver for the `example.com` zone is `ns.example.net`. Since this name is not located below `example.com`, the `com` nameserver can send a referral without glue,[14] and the resolver could still make progress by first sending a separate query to look up the address records for this name to a `net` nameserver. We call these separate queries *resolver subqueries* because they are created by the resolver as part of the resolution of a client query.

## D.2 Advanced Features

So far, we have seen simple resolution scenarios where the query name (QNAME) exists and owns resource records of the correct type. But even if this is not the case, the query may still be answered without an error due to features such as CNAMEs, DNAMEs, or wildcard records.

**CNAME and DNAME records.** A CNAME record declares its owner name to be an *alias* of the name in the value (the so-called *canonical name*). An example use of this feature is to make a website accessible with or without the `www` label. When a resolver encounters a CNAME record, it restarts the query for the canonical name. Note that CNAMEs can form chains or even loops when the canonical name of one CNAME record is the owner of another CNAME record. While CNAME chains may have legitimate uses, loops are always considered an error.

A similar, yet different feature are DNAME records [40]. While a CNAME record rewrites a single name to its canonical name, a DNAME record causes the entire subtree *below* the owner name to be rewritten. For example, a DNAME record at `example.com` with value `other.com` would cause a query for `www.example.com` to be restarted for `www.other.com` instead. The concept of restarting a query for a different name due to CNAME or DNAME records is called *query rewriting*.

**Wildcard records.** Another interesting feature are *wildcard* records. A wildcard is not a special type, but rather a special label (`*`) that can match other labels. For example, assume that there is an A record for `*.example.com`, and a client queries for A records at `nx.example.com`. If there are no records (of any type) at the name `nx.example.com`, a wildcard match occurs and the query is answered with a *synthesized* record that has owner name `nx.example.com`, but the value taken from the wildcard record.

The semantics of wildcard records has caused some confusion over the years [29] due to certain non-intuitive interactions. Continuing the above example, if there were records of *another* type, say TXT, at `nx.example.com`, the wildcard record would not match a type A query for this name. In a similar vein, if the wildcard record had type TXT instead of A, the type A query for `nx.example.com` would be answered with a NODATA response, not a NXDOMAIN response as one might expect. Finally, a wildcard "blocks itself", meaning that the wildcard record at `*.example.com` does not match any names below itself. For example, the query for `abc.*.example.com` would be answered with NXDOMAIN, assuming no records at that name or any name below exist.

## D.3 QNAME Minimization

The final and most recent feature that we want to introduce is QNAME minimization [11], an enhancement to improve privacy with respect to the authoritative nameservers of higher parts of the namespace. Recall that in normal DNS resolution, the recursive resolver sends the full QNAME to each nameserver. However, this reveals more information than necessary: For example,

[13] More precisely, NODATA means that there exists an RRset of *any* type at this name or any name below.

[14] In fact, if the nameserver *did* include glue records, they would be rejected by the resolver due to *bailiwick rules*, a mitigation for certain cache poisoning attacks.

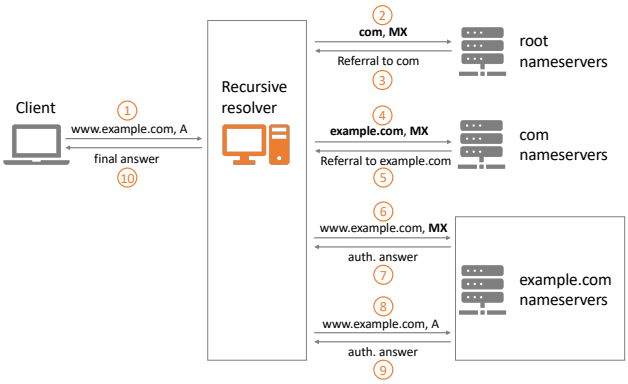


**Figure 13: Resolution of the type a query for `www.example.com` using qname minimization with type mx to obfuscate the original type.**

for the root server to provide a delegation to the `com` nameservers, it would suffice to query for `com` rather than the full `www.example.com`. Similarly, the `com` nameserver only needs to learn that we are interested in *some* name at or below `example.com`, but not which one exactly. Also note that the delegations are independent of the QTYPE, and thus the resolver can use a different type to obfuscate the original one.

**Example.** Figure 13 shows the resolution of the same query as in the previous example, but this time using QNAME minimization, i.e., sending only the minimal number of labels to each nameserver and using the MX type to obfuscate the original QTYPE. The notable differences in resolution are emphasized using bold font. In particular, the resolver sends a query for QNAME `com` and type MX to the root servers in step 2, and similarly abbreviates the QNAME sent to the com nameservers (step 4). Also note that *two* queries are sent to the `example.com` nameservers for the original QNAME: The first one (step 6) is for type MX, so another query is necessary for the original QTYPE once the authoritative nameservers for the full QNAME have been discovered (step 8).

In general, QNAME minimization is complicated by the fact that not every label in a name marks a zone cut: For example, the name `a.b.c.d.example.com` may still be in the `example.com` zone. If QNAME minimization is used, the resolver first needs to discover that there is no delegation between `example.com` and `a.b.c.d.example.com`, by querying the `example.com` nameserver repeatedly with one more label each time. Note that adding more than one label at a time risks revealing unnecessary labels to a non-authoritative nameserver because there could be a zone cut between any two labels. Only once the resolver has determined the authoritative nameserver for the full QNAME, it switches to the original QTYPE, making one final additional query unless the original QTYPE and the QTYPE used for QNAME minimization happen to agree.[15]

In traditional resolution, the resolver would send the full QNAME with the original QTYPE in the first query and immediately receive the desired answer. Clearly, the process of discovering the absence of zone cuts for QNAME minimization introduces an amplification vector that must be dealt with [18, 50]. Possible mitigations include limiting the maximum number of iterations or falling back to the full QNAME after a certain number of labels without delegation.

[15]To maximize the likelihood of this happening, implementations often select a common type such as A to obfuscate the original QTYPE. In our example, we used the MX type to illustrate a case where an extra query is necessary.