A Formal Framework for End-to-End DNS Resolution

Si Liu[*] ETH Zürich
Huayi Duan[*] ETH Zürich
Lukas Heimes ETH Zürich
Marco Bearzi ETH Zürich
Jodok Vieli ETH Zürich
David Basin ETH Zürich
Adrian Perrig ETH Zürich

ABSTRACT

Despite the central importance of DNS, numerous attacks and vulnerabilities are regularly discovered. The root of the problem is the ambiguity and tremendous complexity of DNS protocol specifications, amid a rapidly evolving Internet infrastructure. To counteract the vicious break-and-fix cycle for improving DNS infrastructure, we instigate a foundational approach: we construct the first formal semantics of end-to-end name resolution, a collection of components for the formal analyses of both qualitative and quantitative properties, and an automated tool for discovering DoS attacks. Our formal framework represents an important step towards a substantially more secure and reliable DNS infrastructure.

CCS CONCEPTS

• **Networks → Application layer protocols**; **Denial-of-service attacks**; **Formal specifications**; • **Software and its engineering → Formal methods**.

KEYWORDS

DNS, Formal Semantics, Maude, Statistical Model Checking, DoS

ACM Reference Format:
Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. 2023. A Formal Framework for End-to-End DNS Resolution. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3603269.3604870

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, its complexity has reached epic dimensions. In fact, since the two initial RFCs defining the basics of DNS [36, 37], 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 such as security and privacy enhancements [7, 11], and above all, to elucidate many of its intricacies, which surface in real deployments (e.g., [12, 21, 25, 29]).

[*]Joint first authors with equal contribution.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604870

The complexity of DNS makes it notoriously difficult to manage and operate. Name resolution failures, resulting from misconfiguration, attacks, or operation errors, have caused large-scale outages [46, 52]. In particular, denial of service (DoS) attack vectors [1, 13, 33] have been regularly discovered in the last decade, e.g., iDNS [33], DNS Unchained [13], NXNS [1], and TsuNAME [38].

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. [26] provide the first and only protocol-level formalization of DNS, upon which they build a static verifier called GRoot. Their verifier can identify common configuration errors in a set of zone files prior to their deployment. However, this seminal work has several limitations. First, their model abstracts away crucial aspects of DNS resolution, especially the resolver-side logic including caching, recursive queries for nameserver names, data sanitization, etc. Second, their formalization is not accompanied by an executable model, which is essential for a sanity check of the semantics and automated formal analysis. Third, their framework only supports the analysis of a certain class of correctness properties for DNS configuration, but not quantitative properties such as amplification factors, query latency, and query success ratios. These together limit the scope of problems GRoot can identify in the intricate name resolution process.

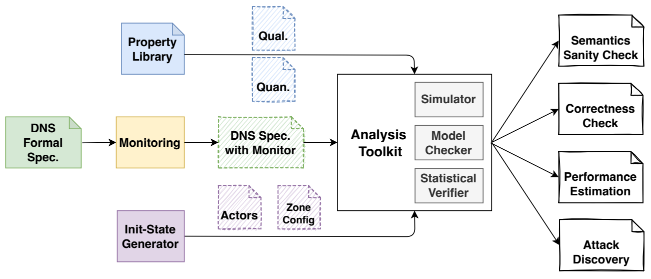

Figure 1: Overview of our formal framework.

In this paper we present a comprehensive formal framework for rigorously analyzing DNS protocols, depicted in Figure 1. Our framework 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, 32, 50, 51]. At our framework's core is the most complete DNS semantic model to date. It captures all essential aspects of *end-to-end* name resolution including the resolver cache, recursive subqueries, data credibility ranking, query name minimization, etc. Modeling these features is important as they can be abused for DoS attacks [1, 13, 33, 38].

During the course of formalizing the DNS RFCs, we have identified and resolved a number of ambiguities 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.*

To separate concerns between semantics (the formalization of DNS resolution) and analysis, we develop our framework in a *modular* and *extensible* fashion. In addition to the semantic model, it comprises three components: (1) a monitoring mechanism that supports the flexible definition of properties over the semantics; (2) a library of both *qualitative* properties (subsuming those considered by GRoot [26]), and *quantitative* properties (e.g., amplification factors and query success ratios); and (3) an initial-state generator that triggers automated analysis.

With a DNS semantic model and three components, our framework also integrates a toolkit for different, relevant formal analysis tasks: (1) simulation (by Maude's built-in simulator) for sanity checking the semantics, quick system testing, etc.; (2) temporal model checking using Maude's built-in LTL model checker, which can exhaustively explore the state space with respect to a correctness property such as the absence of rewrite blackholing; and (3) statistical verification using the PVeStA statistical model checker [5]. This 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?"*, with a given statistical confidence.

Our framework can be used to reason about many kinds of relevant aspects of DNS including functional and security properties. In this paper our focus is on the automated analysis of DoS vulnerabilities in DNS. We apply our framework to rediscover existing attacks [1, 13, 38] and identify 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 for these implementations are consistent with our model-based predictions, which attests to the accuracy and predictive power of our framework.

2 PRELIMINARIES

2.1 The Domain Name System

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

DNS Namespace. The DNS namespace is organized in a hierarchical tree structure, where each node has a label. 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 namespace is divided into different administrative units called zones (subtrees). A child zone is created when the authority of a subtree is delegated.

Resource Records. DNS data is stored in so-called *resource records* (RRs). An 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 record's type. In general, multiple records with the same owner name and type can exist, provided they have different data. Such records form an *RRset.* There are various record types, such as TXT, SOA, NS, and A, each serving a different purpose. For example, an A record can store the IPv4 address of a web server or the name of a nameserver.

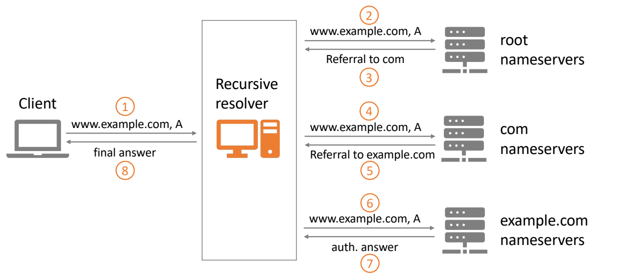

Figure 2: Resolution of the type A query for `www.example.com`, starting from an empty cache at the resolver.

DNS Resolution. When a client makes a DNS request, it typically sends the query to a *recursive resolver*, which performs the resolution on the client's behalf. Recursive resolvers are usually operated by ISPs (Internet service providers) 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.

Example 2.1 (DNS Resolution)**.** Figure 2 illustrate a DNS resolution process. A client starts by sending a type A query for the name `www.example.com` to its recursive resolver (①). Assuming that the recursive resolver has an empty cache, it now sends this query to one of the root nameservers (②).[1]

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 (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` (④ and ⑤). The resolver sends the query to these nameservers, and finally obtains an authoritative answer (⑥ and ⑦) that it sends back to the client (⑧).

Note that if the same query is 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 and here we distinguish two cases: A NXDOMAIN response indicates that the *name* does not exist, which means that there is no resource record (of any type) at this name or at any name below. In contrast, a NODATA response means that the name exists but there is no *data* of the requested type.

[1]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.

Glue Records and Resolver Subqueries. Unlike the root server addresses, the addresses of all lower-level nameservers 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 illustrate 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, it would need 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,[2] 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.

We defer to Appendix A more advanced features of DNS, such as CNAMEs, DNAMEs, wildcard records, and QNAME minimization.

2.2 Maude and Actors

Maude [16] is an executable formal specification language based on rewriting logic. It also supports machine-checked, automated formal analysis, including simulation, linear temporal logic (LTL) model checking [15], and statistical model checking (SMC) [2, 42]. Maude has been successfully applied to analyze high-level designs of a wide range of distributed and networked systems.

Many prevalent formal analysis tools, such as Uppaal [47] and Prism [28], are based on finite automata models. We are not aware of any work on formalizing and analyzing large-scale distributed systems like DNS using such tools. This might be because these object-based distributed systems have intrinsic features that are quite hard or impossible to represent using finite automata. For example, object attributes may contain unbounded data structures; asynchronous message passing and dynamic object creation may increase the number of messages and objects in an unbounded manner. In contrast, Maude provides expressiveness and modeling convenience for our formalization of DNS.

Maude Syntax. In Maude, specifications are structured into modules. A Maude module specifies a rewrite theory [35] consisting of an equational theory [23], which specifies the system's data types and the operations on them, and a collection of rewrite rules, which model the system's dynamics. We summarize Maude's syntax used in this paper and refer readers to the Maude book [16] for details. Operators (or functions) can have user-definable syntax, with '_' denoting the 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 supports parameterized, customized data types. We will use these data types to model advanced DNS features such as caches.

[2]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 [44].

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

RFC	Description	Main algorithmic change
1034 [36]	Core specification	Resolver & Nameserver
1035 [37]	Core specification	Resolver & Nameserver
2181 [21]	Clarifications	Resolver (data ranking)
2308 [6]	Negative caching	Resolver
4592 [29]	Wildcards	Nameserver
6604 [25]	RCODE clarifications	Resolver & Nameserver
6672 [40]	DNAME redirection	Resolver & Nameserver
8020 [12]	NXDOMAIN clarification	Resolver
9156 [11]	QNAME minimization	Resolver

Actors in Maude. We adopt the *actors* paradigm [3] in our formalization of DNS. 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 a *configuration*, is a *multiset* of actors and messages built with an empty syntax multiset union operator `__`. An actor is represented as a term $<addr : ActorType \mid att_1 : val_1, \ldots, att_n : val_n>$, 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   => *** transition 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). Its attribute `a1` is updated to `x + w` (line 5) with the incoming message's content `w`, 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, parts of the system state, e.g., the client actor is omitted in the rule. We adopt this modeling style in formalizing DNS as it results in concise, readable rewrite rules.

3 DNS FORMAL SEMANTICS

In this section, we present our formalization of end-to-end DNS resolution in Maude. The entire executable specification consists of 28 rewrite rules for the non-deterministic model and 30 rules for the probabilistic model.[3]

[3]https://gitlab.ethz.ch/netsec/dns-formalization-maude

3.1 Scope and Assumptions

Our model focuses on the processing and exchange of protocol-level DNS messages by different actors during resolution. Hence, management tasks, such as configuration, zone transfers [30], and dynamic updates [49], are beyond its scope. Table 1 summarizes the RFCs covered by our formalization.

For data representation, we choose an abstraction level that is reasonably close to actual implementations. For example, we model the different sections 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 [36], 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 [26]. We abstract away aspects of lower-layer protocols such as the maximum UDP packet size.

3.2 Formalizing DNS

We develop two models of DNS: (1) an untimed, non-deterministic model that we use for qualitative analysis such as LTL model checking, and (2) a timed, probabilistic model that we use 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. Advanced DNS features, such as caching and referrals, and the complex system dynamics resulting from the subtle interactions of these features render a faithful modeling of DNS challenging. We summarize our main design choices:

(i) Based on our insight into the RFCs, we model the authoritative nameserver algorithm as a *stateless* function, which can thus be formalized by recursive *equations* in Maude. In contrast, we model the resolver algorithm as being *stateful* (e.g., a resolver must maintain global and per-query state due to referrals or query rewriting), and therefore we specify it using *rewrite rules*.
(ii) To faithfully model the communication of protocol-level messages, we adopt the actors paradigm (see Section 2.2) so that clients, resolvers, and nameservers can communicate in our model using asynchronous message passing.
(iii) We construct an abstract, yet sufficiently refined model to capture advanced DNS features, such as resolver subqueries, caching, and QNAME minimization, by utilizing Maude's parameterized data types. Our refined model allows us to discover sophisticated DoS attacks that leverage these features.

3.2.2 Actors, Messages, and Configurations. Following the actors paradigm, we naturally model DNS in an object-oriented style. The DNS state, represented by a configuration in Maude, is formalized as a multiset of objects, including clients, resolvers, nameservers, and messages (e.g., queries) traveling between these objects.

Actors. A client is modeled with two attributes: a list of queries that it intends to send and the address of the resolver it connects to. We model the tree representation for zones (assumed by the nameserver algorithm) using a list that a nameserver stores.[4]

Unlike clients or nameservers, the local state of a recursive resolver is much more complex. We model its attributes by including both static (e.g., safety belt) and dynamic (e.g., caches, per-query state, pending queries, and work budget) states. Note that, in contrast to static states, dynamic states are expected to change as the system state evolves. The following shows two examples which also utilize Maude's *parameterized*, *customized* data types.

Example 3.1 (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. The following shows an SBELT instance modeled in Maude:

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

If the resolver processes a query with a cold cache with no related entries, it will fall back 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 3.2 (Caches). One important part of the dynamic state is the resolver's cache. There are different types of caches: a normal (positive) cache, storing resource records that the resolver received in NOERROR responses, and a negative cache, storing 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 varying levels of trustworthiness (see Section 4.1).

Figure 3 shows a positive cache instance specified in Maude. The first cache entry (line 1) displays a CNAME record received by a resolver, whose owner name is `a.target-ans.com.root` and the 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 cache's second entry (line 2). This `A` value is the address of the `com` nameserver. In both cached entries, the TTL is set to `inf`, meaning that the cached record never expires. This is the case for our non-deterministic, 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.

Configurations. A system state, whether an initial state, an intermediate state, or a final state, is represented in Maude as a *configuration*. Putting together the above two examples, we show an instance (intermediate) configuration in Figure 4 with one client (line 2), one resolver (line 3), three nameservers (lines 4–6), and one active query

[4]This leads to no loss of information as the tree representation can always be reconstructed from the list.

```
1 cacheEntry(< a . target-ans . com . root, cname, inf, b . target-ans . com . root >, 5) :: *** first cache entry
2 cacheEntry(< b . target-ans . com . root, a,     inf, addrNScom >,                      5)    *** second cache entry
```

Figure 3: A positive cache instance specified in Maude.

```
1 (to rAddr from cAddr : query(1, a . target-ans . com . root, a)) *** query waiting to be consumed
2 < cAddr : Client | queries: query(2, c . target-ans . com . root, a), resolver: rAddr, ... > *** client
3 < rAddr : Resolver | cache: the cache instance in Figure 3, sbelt : the SBELT instance in Example 3.1, ... > *** resolver
4 < addrRoot : Nameserver | db: < com . root,       ns, 3600, ns . com . root >
5                               < ns . com . root, a,  3600, addrNScom >, ... > *** root nameserver
6 < addrNScom : Nameserver | ... >    < addrNStargetANS : Nameserver | ... > *** the other two nameservers
```

Figure 4: An example (intermediate) system state, consisting of one client (line 2), one resolver (line 3), three nameservers (lines 4–6), and two messages sent from the client to the resolver (one message is pending, buffered in `queries`). We omit other attributes of an actor, marked by "...".

(line 1). The client actor has type `Client`, address `cAddr`, and the address `rAddr` of the resolver to be queried. The client has already sent the query with identifier 1 (line 1), which is waiting for the resolver to consume. Another client query, with its identifier 2 and the record type A, is currently buffered in `queries` (line 2). A list of records, modeling the tree representation for zones, is stored in the nameserver's database (lines 4–5). Cached entries (like those in Figure 3), resulting from query resolution, are contained in the `cache` of the resolver `rAddr` (line 3).

3.2.3 *DNS Dynamics.* We utilize rewrite rules to specify DNS dynamics which take a (partial) configuration, representing the current system state, and transform it into a configuration, representing the next system state. To exemplify our formalization with refined resolvers and caching, Figure 5 depicts a rewrite rule that corresponds to Step 4, Case A of the resolver algorithm from RFC 6672 [40]:

> 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 `[resolver-recv-answer-for-client]` has 74 LOC with nontrivial auxiliary functions and rule conditions. For the simplicity of our presentation, we only show the most important snippet with respect to positive caching.[5] The 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*. 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 non-deterministic, which is well-suited for detecting timing-independent bugs in DNS, e.g., rewrite blackholing. However, it is ill-suited for (1) capturing the

```
1  crl [resolver-recv-answer-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, ...) /\
9       *** Cache hit in temporary cache
10      RESP := responseFromCache(CACHE_TMP, ...) /\
11      *** Cache the data from the response
12      CACHE' := updateCacheCred(ANS, ...) /\
13      ... *** other assignments or conditions
```

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

full TTL semantics and timeouts in DNS, and (2) quantitative analysis, e.g., using statistical model checking. Hence, we transform this model into a timed, probabilistic model by following the systematic methodology proposed by Agha et al. [4]. The idea is to assign to each message a delay sampled from a continuous probability distribution,[6] which determines when the rule that receives the message fires. To schedule the delayed messages (as multiple delayed messages may appear in one 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. More specifically, an incoming message of the form $\{gt, msg\}$ is delivered at the global time gt; an outgoing message of the form $[gt + d,\ msg]$ will be delivered in d time units after gt. The scheduler is specified to advance the global time from gt to $gt + d$ and transform the delayed outgoing message into $\{gt + d,\ msg\}$, which is, by then, ready to be consumed.

This transformation results in a model that is free from unquantified non-determinism [31, 42] in that all transitions are associated with probabilities. The resulting model is therefore suitable for SMC. Section 5.2 shows an example of the transformed rules. Moreover, unlike the untimed model where a cached record never expires (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).

[5]We do not include variable declarations, but follow the Maude convention that variables are written in (all) capital letters. We also omit other attributes of an actor, arguments of functions, and conjunctions in the condition.

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

4 RESOLVING AMBIGUITIES

Formalizing the DNS semantics forces us to iron out any imprecision in the relevant RFCs. Ambiguities in specifications are often at the root of implementation bugs, configuration errors, and security vulnerabilities. To illustrate this, we present three representative ambiguous cases relevant to the new attacks we discovered (Section 6). We show other semantic ambiguities in Appendix B.

4.1 Ambiguities in Resolver Algorithm

4.1.1 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. Such subqueries are internal to a recursive resolver and hidden from clients. Yet, from an authoritative nameserver's perspective, resolver subqueries are no different from other types of queries it may receive. Hence, an important decision for us to make is whether resolver subqueries are treated as special cases during resolution.

One particular question arises regarding whether CNAMEs should be followed in subqueries, which is a situation that occurs when an NS record points to an alias. As defined in RFC 1035 [37], 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 an NS record, or part of the value of an MX record must not be an alias.

However, the specification also advocates a *robustness principle* [36, §3.6.2] to reduce the risk of failure:

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

Major implementations diverge in their treatments of this: BIND immediately aborts a query if the name of a nameserver is found to be an alias, whereas both Unbound and PowerDNS 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 would remain hidden.

4.1.2 Data Credibility. A DNS answer can contain varying records with different levels of relevance and credibility. There must be proper policies in place for resolvers to decide which records to accept and cache, as well as 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, it is difficult to interpret and implement these rules precisely and comprehensively. To complicate matters further, in many situations a resolver must 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` (see 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 on 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.

This 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 unspecified: 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 in our semantic model that only the record describing an alias is authoritative. Nonetheless, with the default value (i.e., 2) for `rsvMinCacheCredClient` and `rsvMinCacheCredResolver`, our model becomes more forgiving and can capture the other case as well. This is an example of our framework's configurability and flexibility.

4.1.3 QMIN Limits. QNAME minimization (QMIN) [11] is an enhancement to improve privacy for DNS authoritative queries. 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. This is explicitly documented [11, §2.3]:

> 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.

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

Resolvers supporting QNAME minimisation MUST implement a mechanism to limit the number of outgoing queries per user request.

Nonetheless, among other ambiguities (see Appendix B), 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 the resolver software that is considered in our measurements.

4.2 Discussion

DNS semantics is complex. Completely capturing all of the corner cases is highly non-trivial, especially because hundreds of RFCs, documented informally, contain ambiguities as well as unspecified, even conflicting statements. Moreover, there is a lack of canonical reference semantics. All together, this leads to a significant divergence between actual implementations in practice.

To handle these difficulties, our formalization closely follows the RFCs' documentation and, whenever possible, uses principles for disambiguation (e.g., the robustness principle in Section 4.1.1). Moreover, our semantics is executable, so we can test it incrementally. This allows us to eliminate ambiguities and to enhance our understanding of the semantics' corner cases. In case of ambiguities, our executable model also enables us to quickly explore different interpretations, analyzing the consequences of each of them (e.g., data credibility in Section 4.1.2). Finally, when using our model to analyze possible real-world attacks, we are more interested in the interpretations made by real-world implementations. Therefore, whenever possible, we follow their common interpretations; see, e.g., Section 4.1.3 for enforcing the QMIN limits.

Validating Our Semantics. The RFCs provide very few test cases for wildcard [29, §2.2.1] and QMIN [11, §2.3]. We have checked that our semantic model passes all these tests.

In the absence of a reference semantics and rich benchmarks, we decide to validate our formalization of our semantic model by heavily testing it. The test cases are designed to check that we have formalized all the main and corner cases correctly and that they have satisfied expected properties. In the model, the rewrite rules use around 150 auxiliary functions to specify resolver operations (cache lookup and manipulation, the creation of subqueries, etc.); recursive equations are used to define nameserver operations (DNAME substitution, wildcard processing, etc.). To ensure their correctness, we have 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 (Section 6), we have generated millions of randomized configurations. Our tool always checks if an execution ends in a well-formed state, e.g., a resolver has no pending/blocked queries. In addition, as we will see, the model-side estimations are consistent with the implementation-side evaluation results. All together, this leads to a high assurance of our semantics' correctness.

5 FORMAL ANALYSIS

This section describes how to utilize our framework to perform formal analysis using simulation, model checking, and statistical verification. The analysis tools take as input our semantic model augmented by a monitoring mechanism (Section 5.2), an initial state (Section 5.1), and a property of interest (Section 5.3). Depending on the type of analysis, the tool's output can be a final system state with useful information for subsequent analysis, a validated qualitative property or counterexample, or a quantitative result with statistical confidence.

5.1 Initial-state Generation

We implement a *configurable* initial-state generator to facilitate automated analysis. Our generator takes as input a range of parameter values and outputs a Maude module that can be used to invoke one of our analysis tools. We consider two classes of parameters: (1) those for configuring the behaviors of actors, especially the resolver, and (2) those for controlling the generation of zone files and how they are assigned to nameservers. Table 2 provides a partial list of the parameters in class (1) pertinent to our discussion in Section 6. The parameters in class (2) include, for example, the number of zones, the record types, and the depth of names. Our random zone generation algorithm is given in Appendix C.

5.2 Monitoring

We define properties over a global *monitor* object, which is attached to the rewrite rules and records statistics on system executions. The monitor is specified just like the nameservers and resolvers with two main attributes: one for keeping track of the queries sent by clients (`clQueryLog`) and the other for recording the responses received by them (`clRespLog`). Note that we use custom data structures

```
1 rl [client-recv-resp-send-next] :
2    < M : Monitor | clQueryLog: L, clRespLog: L', ... >
3    {T, to CL from RSV : RESP} *** arriving at time T
4    < CL : Client | queries: Q QS, ... >
5 =>
6    < M : Monitor | clQueryLog: < CL, Q > L,
7                   clRespLog: < CL, RESP > L', ... >
8    < CL : Client | queries: QS, ... >
9    [T + delay, to RSV from CL : Q] .  *** delayed msg
```

Figure 6: Example rule for the monitoring mechanism

to store the information, in this case lists of <*address*, *query*> or <*address*, *response*> pairs.

Figure 6 illustrates the monitoring mechanism with an example rewrite rule where the client receives a response from its resolver and there are more queries to be issued. Along with the client's reaction, the monitor also updates its local state: the response that the client receives is added to `clRespLog` (line 7), and the new query sent to the resolver is added to `clQueryLog` (line 6).

Note that this example also illustrates the idea of transforming an untimed, non-deterministic rule into a timed, probabilistic rule (Section 3.3). The resolver response arrives at the global time `T` while the client's next query is issued with a `delay`.

Our monitoring mechanism supports a strict separation of concerns between the semantics (i.e., the formalization of DNS resolution, which is the actual client reaction in this case) 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 (e.g., the client's local state marked by "..."). Finally, we can easily *extend* the analysis part without touching the semantics; in particular, we only need to define new attributes and update them in the monitor of relevant rules. See Figure 8, Section 5.3 for an example where the monitor uses other attributes to record the number of answered/failed client queries.

5.3 Property Library and Formal Analysis

Our framework supports the analysis of two types of properties:

(1) *qualitative* properties like those considered by GRoot [26] and beyond, such as repeated queries, domain overflow at nameservers, and inconsistent RRsets. These are used to check the correctness of our DNS model via Maude's LTL model checker [16].
(2) *quantitative* properties such as the amplification factor that a query induces at the resolver, the probability that a client request is successfully answered, and average query latency. These are used for performance estimations via the PVeStA statistical verifier [5].

We specify (1) using LTL formulas [15] and (2) in QuaTEx [4], a quantitative temporal logic that extends *probabilistic computation tree logic* [24] with real-valued expressions. In the following, we briefly discuss an example for each type of properties. A complete list of our predefined properties, along with their descriptions, can be found in Appendix D.

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 a rewrite blackhole. We specify this property over the response log maintained by the monitor, checking whether there is a response with a non-empty answer section and NXDOMAIN code.

As shown in Figure 7, the state predicate `hasRewriteBlackhole` is defined as 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` which then checks all entries for a non-empty answer section and a NXDOMAIN RCODE (lines 6–8). The atomic proposition in LTL is then specified in Maude syntax (line 11), where `|=` defines the satisfaction relation with respect to the state predicate `hasRewriteBlackhole`. Finally, we define the LTL formula: `[] ~ propRewriteBlackhole`,[7] meaning that the rewrite blackhole property is never satisfied. Our model checker can take this property and verify whether it is violated for given zones and queries specified in an initial state (line 13). See Appendix E for an example.

Query Success Ratio. An important performance metric in DNS is the fraction of *successful* queries, which are 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 typically generate many SERVFAIL responses.

Figure 8 shows how we specify this property, alongside the QuaTEx formula[8] for statistical model checking with PVeStA. The specification is straightforward with the numbers of answered and failed queries stored in the monitor (lines 3–4). We define a temporal operator `succRatio()` and PVeStA returns the ratio (computed by the Maude function to which `rval(1)` refers; see line 7) if all client queries are finished (checked by the predicate `sat(0)` omitted in Figure 8) in the current state `s` (line 10). Otherwise, it evaluates `succRatio()` on the *next* state denoted by `#` (line 11). Finally, the *expected* ratio is returned (line 12).

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,[9] PVeStA returns 0.71, i.e., roughly 71% of legitimate client queries are still answered successfully. With a stronger attacker 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 attacker settings. 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. This suggests that the MAXFETCH(K) mitigation is indeed effective against the NXNS attack.

6 AUTOMATED ATTACK ANALYSIS

In this section, we demonstrate the predictive power of our framework with an 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

[7] `[]` and `~` are Maude syntax for the "always" operator $\square$ and the "not" operator $\neg$ in the traditional LTL notation [15].
[8] A QuaTEx formula is a query of the expected value of a path expression interpreted over an execution path with state expressions interpreted over states [4].
[9] An SMC analysis returns the expected value $\bar{v}$ of a QuaTEx query with respect to two parameters α and δ: $\bar{v}$ is obtained such that, with $(1-\alpha)$ statistical confidence, it lies in the interval $[\bar{v} - \frac{\delta}{2}, \bar{v} + \frac{\delta}{2}]$.

```
*** extract partial information, i.e., clRespLog, from the monitor, thus from the entire system state
op hasRewriteBlackhole : Configuration -> Bool .
eq hasRewriteBlackhole(< M : Monitor | clRespLog: LOG, ... > ...) = rewriteBlackhole(LOG) .

*** define rewrite blackholing
op rewriteBlackhole : addrRespList -> Bool .  eq rewriteBlackhole(nilLog) = false . *** all entries checked
eq rewriteBlackhole(< ADDR, response(ID, NAME, ANS, AUTH, ADD, RCODE) > LOG)
 = if ANS =/= nil and RCODE == 3 then true else rewriteBlackhole(LOG) fi . *** 3 refers to NXDOMAIN

*** atomic proposition in LTL
op propRewriteBlackhole : -> Prop .      eq CONFIG |= propRewriteBlackhole = hasRewriteBlackhole(CONFIG) .

red modelCheck(initState, [] ~ propRewriteBlackhole) . *** invoke LTL model checker
```

Figure 7: LTL model checking of the rewrite blackholing property.

```
*** define query success ratio
op qrySuccRatio : Configuration -> Float .
eq qrySuccRatio(< M : Monitor | qryAnswered: N,
      qryFailed: N', ... > ...) = (N - N') / N .

*** QuaTEx interaction with Maude
eq val(1,CONFIG) = qrySuccRatio(CONFIG) .

*** QuaTEx formula for SMC
succRatio() = if { s.sat(0) } *** all queries done
      then { s.rval(1) } else # succRatio() fi ;
eval E[succRatio()] ; *** expected ratio
```

Figure 8: QuaTEX formula for query success ratio. This formula is used for statistical model checking with PVeStA.

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 of 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 the resolution of the given client query (or queries) produces an undesirable effect like a 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 *and* queries. Unfortunately, this is intractable due to the huge 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), and possible record types (several dozens); meanwhile, zones can be arbitrarily assigned to nameservers.

However, even if we fix the nameserver configurations, the number of possible client queries is still huge as 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. GRoot defines such a notion of query *equivalence classes* (ECs) [26]. While this approach is manually proved sound and complete with respect to its simplified DNS semantics, it is unsound with respect to our more realistic model (explained in Appendix F).

Our Approach. We develop a highly effective attack analysis approach, along with an automated tool, 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's ECs are unsound with respect to our model, they still provide an effective heuristic for the exploration of the query space. Lastly, whether an initial state leads to an attack rarely depends on the particular non-deterministic choices made during the system execution, such as the interleaving between messages. More often, it suffices to analyze *one* possible execution to determine if the configuration leads to an attack.

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 utilizes 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, our tool then computes GRoot's ECs and creates a separate initial state for each EC, with a client sending a representative query from the EC. Subsequently, our tool invokes Maude's simulator to execute our model with each initial state, computing the metric of interest. The resulting value is then compared to a user-provided threshold: if it exceeds the threshold, the execution is considered to be a successful attack and the initial state is stored for a later inspection.

6.2 Rediscovering 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, and 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`), which are all hosted by separate nameservers. This simulates a minimal realistic setup of the DNS namespace. We configure our tool to record a suspicious run 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 (which is considered in Section 6.3).

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

NXNS Attack. While considering a single malicious zone, our tool already reports a potential attack after a few runs with respect to 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. >
```

This zone contains an unusual number of NS records for `r.atk1.com` pointing to non-existent nameservers. The amplification is caused by the subqueries that the resolver generated for each of the 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's domain. As there is no explicit victim in our setup, different names from the benign zones are used to construct the nameserver names.

TsuNAME Attack. The following is reported as a potential attack configuration, when initial states are generated with *two* random malicious zones.

```
--- zone atk1.com
< rr.j.atk1.com., NS, nxdomain0-1.o.atk2.net. >
--- 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 is significantly 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 behaviors [38] with repeated queries sent to the `atk1.com` zone until the resolver's `workBudget` (Table 2) is eventually exceeded.

Unchained Attack. It is possible 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, one must generate a large number of zones before a long CNAME chain is generated. 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 the random zone generation and additionally restrict the *values* of records to contain only names in other malicious zones, our tool can report suspicious Unchained behaviors.

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, which 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*.

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

To validate the attacks, we create a testbed using popular DNS implementations which mirrors the configuration of our model. Table 3 summaries the default parameters of these implementations relevant to our discussion.

Subqueries + Unchained. Our analysis of Unchained focuses on CNAMES in the random zone generation. By additionally allowing NS RRsets for leaf zones, our tool reports attack cases with a 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 therefore *multiplied.*

Figure 9a 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. Additionally, 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 predictions almost exactly. BIND is not vulnerable to this attack as it does not follow CNAME chains starting from nameserver names. PowerDNS limits the maximum CNAME chain length that 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 go to the victim. Unbound generates even more queries than predicted by our model. Our inspection of the query logs shows 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 does not trust an entire CNAME chain received in a single response and continues to query the canonical name of the first record. This is unofficially referred to as *CNAME scrubbing* [39] in the DNS development community. We can enable this behavior in our model by increasing `rsvMinCredClient` and `rsvMinCredResolver` to 5. With these minor changes, our tool can report potential malicious zones as for 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 is expected to achieve twice as high an amplification factor as the previous attack. This is confirmed by our measurement results depicted in Figure 9b. Again, both PowerDNS and Unbound are vulnerable to the attack while BIND is not. Despite being more powerful than Unchained, to the best of our knowledge, the exploitation of CNAME scrubbing for amplification has not been reported before.

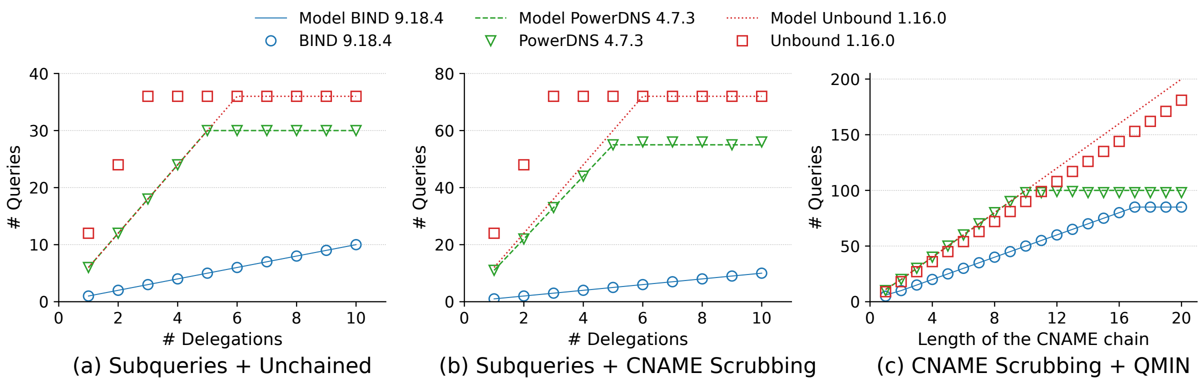

Figure 9: Measurement results of our new attacks that produce large amplification by combining vulnerabilities.

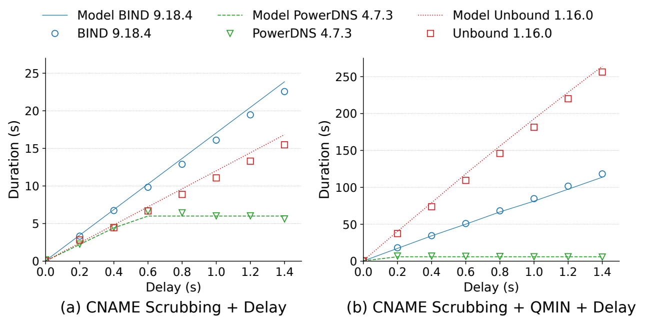

Figure 10: Measurement results of our new slow DoS attacks that prolong query processing time at a resolver.

CNAME Scrubbing + QMIN. In our analysis of all the previous attacks, the QMIN feature is disabled. We now enable it by setting a non-zero value for `maxMinimiseCount`. Other settings for the 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 9c. Depending on the implementation of the QMIN algorithm, the attack can incur up to 10 queries per element on a long chain. BIND and PowerDNS hit a maximum number of queries of 85 and 100, respectively, as they impose overall resource budgets for each client request. Interestingly, Unbound keeps generating queries in some of our test cases even when it exceeds its limit on CNAME chains. After investigation, we additionally discover 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. Our tool can also support more advanced analysis 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.

We now describe how we produce such slow queries to prolong the resolution of a malicious client query. Specifically, 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 an attack alarm. Moreover, since the resolver's query duration depends on probabilistic message delays, we use SMC, instead of a single simulation, to estimate the duration.

With this setup, our tool automatically reports slow DoS attacks for malicious zone files. These attacks force the resolver to, in the resolution of one single 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. Note that 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 as 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 10 depicts our measurement results with varying artificial delay for the attacks enabled (a) by CNAME scrubbing and (b) additionally by QMIN, respectively. The maximum artificial delay we show is 1.4 seconds, which is the largest value for which we 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 shown by the plots, the behavior of real resolvers closely follows 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 is due to the different limits enforced by the implementations. Comparing the two plots, we see that QMIN significantly intensifies the slow DoS attack: a single malicious client query can last for over 100 seconds in two resolver implementations. 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.

These amplification vulnerabilities can be exploited by attackers to launch highly effective DoS attacks at low costs. We anticipate that DNS developers and operators can leverage our framework to proactively discover and fix such security vulnerabilities.

6.4 Discussion

Improving Automation. While our approach produces zones and queries that exhibit undesirable system behavior, some manual work is still necessary to triage and classify the results. We have seen that discovering certain attacks requires changing the focus of the zone generation to, e.g., specific record types.

Moreover, the generated zones mainly expose problematic interactions of DNS features, whereas constructing realistic attack scenarios would be even more helpful. For example, our tool easily discovers that large delegations to non-existent nameservers introduce a large amplification, which is, however, not yet directed at a specific victim. We would like to improve our tool to automate the simulation of real-world attacks, e.g., automatically placing the non-existent nameservers under a victim zone to maximize amplification as in a full-fledged NXNS attack.

Reducing the Attack Search Space. While we have demonstrated that random zone generation with a heuristic coverage of the query space is quite effective in discovering problematic interactions of DNS features, an exhaustive (albeit still bounded) exploration of the search space would be preferable. GRoot's ECs provide a good starting point, but they are unsound with respect to our more realistic semantic model, i.e., two queries in the same EC can cause a resolver to behave differently.

As future work, we plan to refine the definition of ECs for the specific analysis of amplification attacks. GRoot does not place multiple queries for different existing names in the same equivalence class; only non-existent names may be collapsed into one EC. However, different existing names in a zone are often resolved in exactly the same way, e.g., the resolution of a type A query for ns1.example.com and ns2.example.com are likely identical, apart from the contents of the final response. Similarly, query rewriting at the nameserver is irrelevant for amplification (assuming the resolver does not validate the CNAME chain). A promising direction would therefore be to define a "resolver-centric" notion of query equivalence, where two queries are considered equivalent if they lead to the same actions at the resolver.

7 RELATED WORK

Formalizing DNS Semantics. GRoot [26] provides the first formalization of a subset of DNS semantics. We have already discussed its limitations in Section 1. Relying on parts of the GRoot semantics, the authors also develop SCALE [27], a test case generator to find RFC compliance errors in nameserver implementations. While SCALE identifies 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. RHINE is a DNS-based naming system co-designed with a formal model [20], which, unlike our formal semantics, focuses on an end-to-end authentication architecture that is agnostic to the detailed resolution process.

Verification of DNS Implementations. Son et al. [44] formalize the cache implementations of popular open-source resolvers, modeling aspects such as data credibility requirements and bailiwick rules. In comparison, our model is more comprehensive as it specifies 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 being provably secure against single-packet DoS attacks, it may still be vulnerable to application-layer attacks at the DNS protocol level, which are 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 [19], often used for cryptographic protocol analysis, is too strong for this purpose. Meadows [34] introducs a cost-based framework for the analysis of DoS attacks where an attacker's power can be metered and limited. Urquiza et al. [48] refine this intruder model by explicitly considering timing aspects, which can 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. [18] model the classic bandwidth amplification attack as a continuous-time Markov chain to analyze different countermeasures. However, their model is abstract and 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 as well as provide a basis for tools to automatically discover known and new attacks. In addition to pursuing the research directions highlighted in Section 6.4, we also plan to extend our framework to cover DNSSEC and DoH/DoT, and support multiple zones at a single nameserver.

Ethics. We have followed common practices for responsible disclosure of the discovered DoS vulnerabilities. The affected entities have acknowledged the validity of our reported vulnerabilities, and are investigating their potential impact and mitigation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We gratefully acknowledge support from ETH Zürich, ZISC, and from SNSF for project RHINE (200021_215318).

REFERENCES

[1] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. 2020. NXNSAttack: Recursive DNS Inefficiencies and Vulnerabilities. In *USENIX Security 2020*. 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 *CALCO 2011 (LNCS, Vol. 6859)*. 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 (LNCS, Vol. 11050)*. 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 *GLOBECOM 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.* LNCS, Vol. 4350. Springer.

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

[18] 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 *HASE 2011*. IEEE Computer Society, 360–367. https://doi.org/10.1109/HASE.2011.57

[19] 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

[20] Huayi Duan, Rubén Fischer, Jie Lou, Si Liu, David Basin, and Adrian Perrig. 2023. RHINE: Robust and High-performance Internet Naming with E2E Authenticity. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[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.

[24] Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects Comput.* 6, 5 (1994), 512–535.

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

[26] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd D. Millstein, and George Varghese. 2020. GRooT: Proactive Verification of DNS Configurations. In *SIGCOMM '20*. ACM, 310–328. https://doi.org/10.1145/3387514.3405871

[27] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2022. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *NSDI '22*. USENIX Association, Renton, WA, 307–323. https://www.usenix.org/conference/nsdi22/presentation/kakarla

[28] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11 (LNCS, Vol. 6806)*. Springer, 585–591.

[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, 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

[32] 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.

[33] 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.

[34] Catherine A. Meadows. 2001. A Cost-Based Framework for Analysis of Denial of Service Networks. *J. Comput. Secur.* 9, 1/2 (2001), 143–164.

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

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

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

[38] 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, 398–418. https://doi.org/10.1145/3487552.3487824

[39] Olivier Poitrey. Accessed 2023-02-07. Consider disabling CNAME scrubbing for forwarded queries #132. https://github.com/NLnetLabs/unbound/issues/132.

[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 *NDSS 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 *ESORICS 2009 (LNCS, Vol. 5789)*. Springer, 390–405.

[44] Sooel Son and Vitaly Shmatikov. 2010. The Hitchhiker's Guide to DNS Cache Poisoning. In *ICST 2010 (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 50)*. 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] Sam Thielman and Chris Johnston. Accessed 2023-02-07. Major cyber attack disrupts internet service across Europe and US. https://www.theguardian.com/technology/2016/oct/21/ddos-attack-dyn-internet-denial-service.

[47] Uppaal. 2023. Uppaal 4.0.15. http://www.uppaal.org.

[48] 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 *CSF 2019*. IEEE, 382–396. https://doi.org/10.1109/CSF.2019.00033

[49] 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

[50] 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 *INFOCOM 2014*. IEEE, 637–645. https://doi.org/10.1109/INFOCOM.2014.6847989

[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] Zack Whittaker. Accessed 2023-02-07. A DNS outage just took down a large chunk of the internet. https://techcrunch.com/2021/07/22/a-dns-outage-just-took-down-a-good-chunk-of-the-internet/.

Appendices are supporting material that has not been peer-reviewed.

A ADVANCED FEATURES OF DNS

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.

A similar feature are DNAME records [40]. However, 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. 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 with the owner name `nx.example.com`, but the value taken from the wildcard record.

QNAME Minimization. One recent DNS feature is QNAME minimization (QMIN) [11], an enhancement to improve privacy for DNS authoritative queries. In normal DNS resolution, the recursive resolver sends the full QNAME to each nameserver, which reveals more information than necessary. For example, 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. Note that the delegations are independent of the QTYPE, and thus the resolver can use a different type to obfuscate the original one.

Example A.1. Figure 11 shows the resolution of the same query as in Example 2.1, which uses QMIN this time, 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 ②, and similarly abbreviates the QNAME sent to the com nameservers in ④. Also note that *two* queries are sent to the `example.com` nameservers for the original QNAME: The first one (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 (⑧).

QMIN is complicated as 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. With QMIN, 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 as 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 QMIN happen to agree.[10]

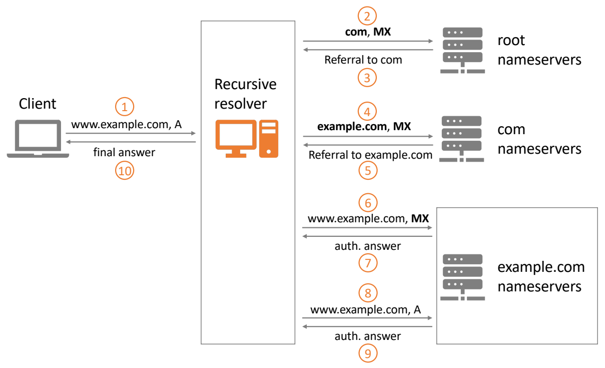

Figure 11: Resolving the type A query for `www.example.com` using QMIN with type MX to obfuscate the original type.

B RESOLVING OTHER AMBIGUITIES

B.1 Clarifying the Resolver Algorithm

In the absence of QMIN, the resolver algorithm given in RFC 6672 [40, §3.4.1] is the most refined one, as shown in listing 1. At a high level, it works as follows: when the recursive resolver receives a client query, it first tries to answer the query from its 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 the resolver 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.

Additionally, there are a number of error cases that must be handled: the nameserver may not respond at all (e.g., as the message was dropped), refuse to answer the query (e.g., as 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. The resolver works on many client queries concurrently, and thus needs to maintain state for each of them.

An important notion is that of *resolver subqueries*, which are 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 indicates 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

[10]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.

```
 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.

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 must be resolved:

(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 the resolver 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 largely ignores 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).

Moreover, 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 as 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 QMIN algorithm, which substantially changes a resolver's behavior with respect to both the QNAME and QTYPE sent to authoritative nameservers. RFC 9156 [11] contains an extended resolver algorithm using QMIN, as shown 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 numerous 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 using the most common QTYPE for obfuscation, which saves the extra query 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 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 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.

Whether or not the cache is checked in Step (3) also has consequences for the handling of CNAME responses. As indicated in Case (6c), a CNAME received for an incomplete QNAME must not be followed. 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

```
(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 nameservers. 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.

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 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 (yet 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 QMIN can thus be much larger than MAX_MINIMISE_COUNT, an attack vector that has been overlooked so far and which we exploit in different ways in Section 6.

Finding Zone Cuts. Finally, we discuss Step (5) of the algorithm. This step retrieves information on the absence of zone cuts from the cache, which would allow the resolver to safely add additional labels without revealing unnecessary information. However, there are a number of logical flaws in the specification:

[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.

(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 neither mandates nor suggests 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) In contrast, entries in the *negative* cache 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 searching 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 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 associated SOA record's
     owner name is ANCESTOR, go back to Step (3).
```

C 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. We do not currently generate wildcard labels, although this could be easily 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 associated RRsets. Similarly, records can point to non-existent names, which are created by adding a number of non-existent labels to an existing name. Note that this existing name can be the root label, thus any non-existent name is possible. 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.

D PREDEFINED PROPERTIES

Table 4 lists our predefined properties, alongside their description.

E ZONE FILES FOR REWRITE BLACKHOLING

Using model checking, we can detect misconfigurations in zone files that lead to a rewrite blackhole. Consider a setup with a client, a recursive resolver, and various authoritative nameservers. In particular, there are two nameservers for the `example.com` zone, namely `ns1.example.com` and `ns2.example.com`. However, these nameservers are misconfigured with inconsistent zone files for the `example.com` zone. The zone file used by `ns1.example.com` is shown below:

```
--- 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. >
```

The zone contains the mandatory SOA and NS records, alongside the address records for the nameservers. In addition, there is a type A record for `www.example.com` and a CNAME record at `alias.example.com`, pointing to `www.example.com`.

The other zone file, used by `ns2.example.com`, is almost identical, except that the CNAME record points to the (non-existent) domain `nxdomain.example.com`.

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

Whether a client that sends a query for `alias.example.com` encounters an instance of *rewrite blackholing* thus depends on which nameserver the resolver asks, which is a non-deterministic choice in DNS.

F UNSOUNDNESS OF GROOT'S ECS

We show that the GRoot's equivalence classes (ECs) are *unsound* with respect to our semantic model. This means even if two queries are resolved in the same way under GRoot's simplified semantics, they may resolve to different answers under our more realistic semantic model.

To illustrate this unsoundness, we show two queries that are in the same GRoot's EC, but have different resolution behaviors. 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 >
```

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 >
```

Let us now consider two queries for the same type with QNAMES `a.example.com` and `b.example.com`. Neither of the names appear in the zone files. Hence, both queries are contained in the equivalence class α`.example.com` for that type.

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 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,

Table 4: Our library of predefined properties. Properties 1–10 are also considered by GRoot [26].

#	Property	Kind	Description	Comment
1	Delegation Inconsistency	Qualitative	The NS or A records in a referral response differ from the records in an authoritative response.	
2	Lame Delegation	Qualitative	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	Qualitative	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	Qualitative	The query is answered with NXDOMAIN.	
5	Cyclic Zone Dependency	Qualitative	The delegations involved in the query's resolution are cyclically dependent.	
6	Rewrite Loop	Qualitative	The query is rewritten to itself.	
7	Domain Overflow	Qualitative	The query name at some point exceeds the maximum domain length.	This can happen due to DNAME substitution.
8	Answer Inconsistency	Qualitative	The query can produce different answers.	This can happen due to inconsistent zone files.
9	Zero TTL	Qualitative	Resolution of the query involves records with a TTL of 0.	
10	Rewrite Blackholing	Qualitative	The query is rewritten to a name that does not exist.	
11	Repeated Query	Qualitative	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	Qualitative	The nameserver sends a YXDOMAIN response due to 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	Qualitative	Different nameservers have inconsistent RRsets for the same name and type.	This static property can be checked without considering resolution dynamics.
14	Amplification Factor	Quantitative	The number of messages received by victim divided by the nuber of messages sent by attacker	
15	Query Duration	Quantitative	How long the resolution of a client query takes	
16	Query Success Ratio	Quantitative	The probability for a client query to be answered	

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, the DNAME response (indicating `a.dname.example.net` should be rewritten to `a.example.com`) does *not* yet allow the resolver to detect the rewrite loop as 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 EC. Hence, we can conclude that the GRoot's ECs are unsound with respect to our semantic model.