Declarative DNS – Resolving Recursive Resolution

Felix Linker, *ETH Zurich*
Yue Huang, *HKUST(GZ)*
David Basin, *ETH Zurich*
Huayi Duan, *HKUST(GZ)*

Abstract

DNS zone configuration errors are a frequent source of outages and frustration, particularly in combination with DNSSEC. Research has attempted to address the problem of zone configuration errors using formal methods, but existing proposals (i) only verify the absence of specific, known configuration errors, and (ii) do not consider DNSSEC.

We present a new formal model of the DNS that covers most of DNSSEC, making it one of the most complete DNS models to date. In addition, we present a new tool, DecDNS, that performs a *resolvability change-impact analysis* of zone configuration updates using our model. We show that such a change-impact analysis can identify configuration errors without explicitly searching for them.

We evaluate DecDNS both formally and empirically. We prove that DecDNS is sound and complete and show that its results agree with widely used DNS software. Finally, we explain how our approach can be integrated into operational practice, helping guide zone management in the real world.

Availability Our tool, DecDNS, is available on GitHub:

`https://github.com/felixlinker/decdns`

1 Introduction

The Domain Name System (DNS) is a fundamental Internet technology. So fundamental that the unavailability of the DNS can be practically indistinguishable from the unavailability of the Internet. For example, some governments censor the Internet solely by filtering the DNS [1, 4, 14, 15]. Although "censored" services are technically still reachable, many clients and users cannot reach them anymore.

At first glance, the DNS appears simple. Data is associated with *names* such as `example.com`. All names form a tree, and sets of connected nodes form so-called *zones*. Different zones are served by different *name servers*, and *recursive resolvers* query name servers to obtain the data associated with names. Underlying this simple concept, however, is considerable complexity. The DNS's two original standards were published in 1987 and, since then, it has been extended numerous times. As of fall 2026, the Internet Engineering Task Force (IETF), which maintains the DNS, has published 298 documents, called Requests for Comments (RFCs), on the DNS [3].

One of the most substantial extensions to the DNS are the Domain Name System Security Extensions (DNSSEC). When using DNSSEC, name servers sign the data they serve and publish public key material for verification. DNSSEC drastically increases the complexity of the DNS and thus the likelihood of configuration errors.

It thus comes as no surprise that, given the complexity of the DNS, configuration errors are common and they can have severe consequences. For example, in May 2026, roughly 2/3 of the `.de` top-level domain (TLD) became unavailable to many resolvers [6] due to a DNSSEC configuration error. The `.de` TLD is one of the most frequently queried TLDs [7], and a configuration error in the published public key material used to authenticate data in the `.de` zone led to a 3-hour outage.

There are proposals for detecting configuration errors. For example, some tools test a domain name for certain errors [2, 5]. There is also a large body of work using formal reasoning to detect configuration errors [12, 19, 23, 26, 31]. Such formal approaches generally promise to provide "complete coverage," i.e., modulo abstractions and assumptions of the underlying model and theory, they can provide mathematical certainty. However, existing formal approaches all share two limitations. (i) They suggest verifying the absence of particular, concrete configuration errors. (ii) They do not consider DNSSEC, one of the largest extensions to the DNS.

Note that point (i) above is in stark contrast to formal methods' promise of complete coverage. Since existing approaches only verify the absence of particular classes of configuration errors, they do not capture configuration errors generally and miss every configuration error not encoded directly in the model.

In this work, we address the above two limitations. We

present a new, declarative formal model and a tool called DecDNS (declarative DNS) to reason about DNS configuration errors. Using DecDNS, one can identify configuration errors without explicitly encoding them in the model. Moreover, our model and DecDNS provide support for DNSSEC.

Resolvability Change-Impact Analysis In our approach, we take a new perspective on DNS configurations and consider the property of *resolvability*. We argue that a zone configuration can be characterized by which queries should have a positive response and are thus resolvable. If such a characterization were given, one could detect configuration errors by computing all queries that have a positive response. Any difference to the set of queries that should have a positive response would point to a configuration error.

Such a characterization, though, is hard to obtain. Having the user state all queries that should be resolvable is infeasible and likely a task just as error-prone as configuring the zone itself. We therefore use the old configuration as a baseline and perform a *resolvability change-impact analysis* on a zone configuration update. Given two zone configurations, new and old, a change-impact analysis computes which queries have a positive response in exactly one of the two configurations. Users can then inspect the result of this analysis, compare it with their expectations, and consequently identify configuration errors.

A change-impact analysis, though, will not be able to identify *all* configuration errors. For example, resolvability disregards whether the data associated with a name is correct, e.g., whether the IP address associated with `example.com` indeed routes to `example.com`. Moreover, we only detect configuration errors when they are introduced but not when an update preserves them. Nevertheless, a resolvability change-impact analysis is a beneficial tool as it characterizes a configuration update and is not limited to any particular sort of configuration error.

Additionally, a change-impact analysis is particularly useful for configuration changes that should not impact resolvability, i.e., after which the set of resolvable queries should remain unchanged. Given such a change, a non-empty change-impact analysis immediately highlights a configuration error. Examples of such changes include changing which name servers serve particular zones, changing how the DNS is divided into zones, or changing a zone's verification keys, which is the kind of change that led to the `.de` TLD outage.

Overview of Our Approach Given a zone configuration, DecDNS compiles it to a constraint logic program (CLP). This declarative specification captures which queries have a positive response in that zone configuration. We abstract from details of the underlying transport mechanisms, e.g., packet sizes and headers, and instead model what a recursive resolver would conclude in response to arbitrary queries if it were given the input zone configuration. We capture, though, that recursive resolvers must first discover name servers and obtain their network addresses before they can send them queries. Given two such CLPs, one corresponding to an old zone configuration and the other to a new version of the configuration, DecDNS computes a change-impact analysis.

Modeling the DNS and recursive resolution as a CLP has many benefits. Whereas standard logic programs can only use logical predicates, CLPs can additionally use constraints, which are interpreted by a constraint solver. We use constraints to (i) efficiently represent and reason about very large zones symbolically, and (ii) model cryptographic checks like signature verification. Symbolic representations of large sets and cryptographic checks would be cumbersome to model in ordinary logic programs.

Our formal model only uses standard CLP features and is interoperable with existing logic programming software, for example SWI-Prolog [32], for which we provide an integration. Interoperability with existing software facilitated development initially, but also is independently desirable as it supports the integration of our model or parts thereof into other applications. Moreover, to prove the soundness and completeness of our change-impact analysis algorithm, we can leverage existing soundness and completeness results for CLPs [18].

Contributions We present the following contributions.

- We formally model the DNS and many of its extensions as a CLP, providing one of the most detailed formal models of the DNS to date. In particular, our model covers the DNSSEC standards.

  Our model is an executable, declarative specification of the DNS, which thus promises to be relatively easy to reason about. It could serve as a reference implementation and a basis for further experiments with extensions or variants of the DNS.

- We present an algorithm to compute a change-impact analysis from two CLPs that represent the new and old zone configurations. We provide soundness and completeness results for our algorithm with respect to the well-established semantics of CLPs [18].

- We show that our approach is practical by implementing DecDNS, which can compile a CLP from a zone configuration, following our formal model, and reason about such CLPs. DecDNS provides a mode that computes a change-impact analysis and a querying mode that resolves DNS queries using live DNS infrastructure. The querying mode, in particular, shows that our formal model is not only quite detailed, but that it can also interoperate with live DNS infrastructure. It can be used, relatively straightforwardly, though somewhat inefficiently, to resolve DNS queries on real systems.

- Although we prove the soundness and completeness of our algorithm, no theoretical analysis can establish whether our formal model is faithful to the DNS standards. We thus also empirically evaluate DecDNS and show that its results agree with those of major DNS software. First, we evaluate whether what DecDNS claims to be resolvable is indeed resolvable by real recursive resolver software. Second, we evaluate whether DecDNS's querying mode computes the same responses to queries as other resolvers. We also provide a performance evaluation of DecDNS.
- Finally, we show how to use DecDNS to detect configuration errors. We show which kinds of errors it can detect and under what circumstances.

Structure We begin with a brief example of how DecDNS works and can be used in Section 2. Then, we provide relevant background on the syntax of CLPs and the DNS in Section 3. In Section 4, we present how we compile a zone configuration to a CLP. Then, in Section 5, we present our algorithm to compute a change-impact analysis of two CLPs representing the old and new zone configurations. We present soundness and completeness results as well as relevant background on CLP semantics in Section 6. We present our implementation in Section 7. In Section 8, we evaluate DecDNS's conformance with the DNS and show which configuration errors it can detect. We discuss related work in Section 9 and draw conclusions in Section 10.

2 An Example of DecDNS in Action

We start by providing a brief example of how DecDNS works, how it can be used, and how it can detect potential configuration errors. Consider the following zone, in which the label `lo[...]ong` has 63 characters:

```
*.short. TXT "hello"
*.lo[...]ong. TXT "hello"
```

This zone associates two so-called *wildcard owner names*, those beginning with `*`, with TXT records, i.e., the textual data "hello." The semantics of wildcard owner names are complex, and we will explain them in further detail in Section 3.3, but put simply, queries for TXT records to a subdomain of `short.` or `lo[...]ong.` can be answered with "hello" in this configuration.

Now suppose the zone is updated as follows.

```
short. DNAME lo[...]ong.
*.lo[...]ong. TXT "hello"
```

In this update, one wildcard TXT record was replaced with a DNAME alias to `lo[...]ong.`. DNAME aliases map each subdomain of an aliased name, here `short.`, to the matching subdomain of the alias target. This DNAME record, for example, would map the query `a.b.c.short.` to `a.b.c.lo[...]ong.`. Thus, intuition suggests that this update should preserve the resolvability of subdomains of `short.` since the new alias target has the same record associated with it as the previous wildcard record.

However, one must also take into consideration that DNS names have a maximum length, and DecDNS would highlight exactly this point in its change-impact analysis for the above two zones. Concretely, DecDNS would return:

$$\begin{aligned}&\mathsf{resolvable}(\mathsf{TXT}, [\texttt{short}, L \mid T], \bot) \leftarrow \\ &\quad L \neq *, \\ &\quad \neg\mathsf{isName}([\texttt{lo[...]ong}, L \mid T]), \\ &\quad \mathsf{isName}([\texttt{short}, L \mid T]).\end{aligned}$$

Above is a formula which models when queries for TXT records to any subdomain of `short.` are resolvable in the original configuration but not after the update. The term $[\texttt{short}, L \mid T]$ represents an arbitrary subdomain of `short.`, and is a list of labels. Concretely, L is a variable representing the label directly below `short.`, and T is an arbitrary list suffix, i.e., represents further labels below L`.short.`. The above rule expresses that TXT queries to subdomains of `short.` are not resolvable whenever (a) that subdomain is not at or below `*.short.` ($L \neq *$), (b) the mapped query is too long, i.e., the respective subdomain of `lo[...]ong.` is not a name, and (c) the initial query name is of valid length, i.e., the respective subdomain of `short.` is a name.

The administrator of the respective zone could examine this output and determine whether this loss in resolvability is acceptable and matches their expectations of the update.

While the above example is conceptually simple and small, it highlights how even such small zones can exhibit subtle behavior that can be easy to miss. It also shows the unique benefits of a change-impact analysis. The loss in resolvability in this example, as characterized above, would be hard to formalize in an explicit property that checks for particular configuration errors. A change-impact analysis can highlight the subtle interactions that different parts of the DNS standards have and help zone administrators understand the consequences of their actions.

3 Background

3.1 Mathematical Notation

3.1.1 Functions and Relations

We use $\rightharpoonup$ to denote a partial function, i.e., a function that may be undefined for parts of its domain. Note that every total function is also a partial function. We will sometimes define partial functions explicitly as a set of mappings, e.g., $f = \{1 \mapsto 2\}$ is a partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ such that $f(1) = 2$ and $f(x)$ is undefined for all $x \neq 1$.

Following functional programming conventions, we only use the right-associative symbols $\to$ and $\rightharpoonup$ to define functions, even when they have multiple arguments. For example, $f : A \to B \to C$ is a function with two arguments, mapping to C. This notation permits *partial application*, e.g., $f(a)$ is the function $f' : B \to C$ s.t. $f'(b) = f(a,b)$. We will sometimes treat functions like relations. For example, if $f(a,b) = c$, then $(a,b,c) \in f$.

Let $f : A \rightharpoonup B$ be a partial function. We write $\mathsf{dom}(f)$ for f's domain, i.e., $\mathsf{dom}(f) \subseteq A$ and $a \in \mathsf{dom}(f)$ if $f(a)$ is defined. We write $\mathsf{img}(f)$ for f's image, i.e., $\mathsf{img}(f) \subseteq B$ and $b \in \mathsf{img}(f)$ if there exists $a \in \mathsf{dom}(f)$ such that $f(a) = b$. We write $f|_X$ to denote the restriction of a partial function f's domain to X. That is: $f|_X(x) = f(x)$ whenever $x \in X$ and undefined otherwise.

3.1.2 Terms and Unification

We follow [24] in the definitions below. Let $\mathcal{G}$ be a set of function symbols. We write s/n to denote that a symbol s has arity n. Given a set of function symbols $\mathcal{G}$, variables $\mathcal{V}$, and constants $\mathcal{C}$, we define the set of *terms* $\mathcal{T}$ over those sets inductively as follows:

$$\begin{array}{ll} v \in \mathcal{T} & v \in \mathcal{V} \\ c \in \mathcal{T} & c \in \mathcal{C} \\ f(t_1,\ldots,t_n) \in \mathcal{T} & f/n \in \mathcal{G}, t_1,\ldots,t_n \in \mathcal{T} \end{array}$$

We use $\mathsf{vars}(t)$ to denote the set of variables occurring in a term t. A term t is *ground* if $\mathsf{vars}(t) = \emptyset$. We use uppercase letters for variables in the logic program and lowercase letters for metavariables in our definitions. For example, X is a variable term and v is a variable. We will only refer to constants when they have been explicitly introduced as such. Sometimes, the sets of function symbols, variables, and constants will be part of a signature Σ, in which case we write $\mathcal{T}_\Sigma$ for all terms over that signature.

A *substitution* is a partial function $\theta : \mathcal{V} \rightharpoonup \mathcal{T}$, which maps variables to terms. We homomorphically extend substitutions to also apply to sets of terms. A substitution is a *valuation* if it is total and every term in its image is ground. For two substitutions θ_1, θ_2, we write $\theta_1\theta_2$ to denote their *composition*, defined as:

$$\theta_1\theta_2 = \theta_2|_{\mathsf{dom}(\theta_2)\setminus\mathsf{dom}(\theta_1)} \cup (\theta_2 \circ \theta_1)$$

For example, for two constants a and b:

$$\theta_1 = \{X \mapsto f(Y), Y \mapsto Z\} \quad \theta_2 = \{X \mapsto a, Y \mapsto b, Z \mapsto Y\}$$
$$\theta_1\theta_2 = \{X \mapsto f(b), Y \mapsto Y, Z \mapsto Y\}$$

For a set of terms T, a substitution θ is a *unifier* if $\theta(T)$ is a singleton set, i.e., all pairs of terms in T are equal after applying θ. A unifier θ is the *most general unifier* for T if for all unifiers θ' there exists a substitution σ such that $\theta' = \theta\sigma$. It is well-known that for any two terms t_1 and t_2, a most general unifier exists if and only if t_1 and t_2 unify [24], which we denote by $\mathsf{MGU}(t_1, t_2)$.

3.2 Constraint Logic Programs

We will model the DNS using constraint logic programs (CLPs). Logic programs are sets of Horn clauses, which in turn are first-order logic formulas of the form $\forall\vec{x}.b_1 \wedge \cdots \wedge b_n \implies h$. In a Horn clause, every variable is universally quantified. Moreover, a Horn clause has a single predicate h on the implication's right-hand side, the *head*, and a finite (possibly empty) set of predicates on the implication's left-hand side, the *body*. In this section, we will introduce the syntax of CLPs following [18].

3.2.1 Logic Programs

Logic programs are defined with respect to a signature $\Sigma = (\mathcal{V}, \mathcal{C}, \mathcal{G}, \mathcal{P})$ for a set of variables $\mathcal{V}$, constants $\mathcal{C}$, function symbols $\mathcal{G}$, as before, and a set of predicate symbols $\mathcal{P}$. The set of formulas $\mathcal{F}_\Sigma$ over Σ is inductively defined as:

$$\begin{array}{ll} p(t_1,\ldots,t_n) \in \mathcal{F}_\Sigma & p/n \in \mathcal{P}, t_1,\ldots,t_n \in \mathcal{T}_\Sigma \\ \phi_1 \vee \phi_2 \in \mathcal{F}_\Sigma & \phi_1, \phi_2 \in \mathcal{F}_\Sigma \\ \phi_1 \wedge \phi_2 \in \mathcal{F}_\Sigma & \phi_1, \phi_2 \in \mathcal{F}_\Sigma \\ \neg\phi \in \mathcal{F}_\Sigma & \phi \in \mathcal{F}_\Sigma \\ \forall v.\phi \in \mathcal{F}_\Sigma & v \in \mathcal{V}, \phi \in \mathcal{F}_\Sigma \\ \exists v.\phi \in \mathcal{F}_\Sigma & v \in \mathcal{V}, \phi \in \mathcal{F}_\Sigma \end{array}$$

A formula is an *atom* if it has a predicate as a top-level symbol. For a formula φ, we overload $\mathsf{vars}(\varphi)$ to denote all free variables occurring in φ. A formula φ is *closed* if $\mathsf{vars}(\varphi) = \emptyset$. A formula is a *Horn clause* if it is closed and for $0 \le n, 1 \le m$ and atoms $\phi_1, \ldots, \phi_m$ it is of the form:

$$\forall v_1 \ldots \forall v_n . \phi_1 \vee \neg\phi_2 \vee \cdots \vee \neg\phi_m.$$

Following standard logic programming conventions, we will write Horn clauses as inference rules. The above Horn clause can be written equivalently as:

$$\phi_1 \leftarrow \phi_2, \ldots, \phi_m.$$

If $m = 1$, we omit $\leftarrow$ and simply write ϕ_1. We call such rules *facts*. Note that free variables are implicitly universally quantified. A *logic program* is a set of Horn clauses $\Gamma \subseteq \mathcal{F}_\Sigma$.

We will treat sequences of conjuncts and sets of conjuncts interchangeably and use $\top$ to denote the empty set of conjuncts. For example:

$$\top = \emptyset, \quad \phi \leftarrow \top = \phi,$$
$$\phi_1 \leftarrow \phi_2, \ldots, \phi_n = \phi_1 \leftarrow \{\phi_2, \ldots, \phi_n\}.$$

Notation To limit the naming of variables, we will write _ to denote a fresh, universally quantified variable that occurs nowhere else. For example, $\forall x.\forall y.p(x,y)$ and $p(_,_)$ are equal up to renaming of quantified variables. In this paper, we will use the function symbols $\cdot/2, []/0, 1/0, 0/0$ where $\cdot$ prepends an element to a list, [] denotes the empty list, 1 denotes the bit 1, and 0 denotes the bit 0. Following Prolog syntax, $[x_1,\ldots,x_n]$ denotes a list with n elements and $[x_1,\ldots,x_n \mid y]$ denotes a list with tail y. This means:

$$[x_1,\ldots,x_n] = x_1\cdot(x_2\cdot(\ldots\cdot(x_n\cdot[])))$$
$$[x_1,\ldots,x_n \mid y] = x_1\cdot(x_2\cdot(\ldots\cdot(x_n\cdot y))).$$

3.2.2 Resolution

Logic programs are typically used by querying them with a *goal*. A goal is an atom potentially containing free variables, and querying the logic program should determine whether the goal is entailed by the program. For example, consider the following program P, which formalizes when a Peano number is even and when all elements of a list are even:

$$\mathsf{even}(0), \mathsf{even}(\mathsf{s}(\mathsf{s}(N))) \leftarrow \mathsf{even}(N),$$
$$\mathsf{all_even}([]), \mathsf{all_even}(A\cdot B) \leftarrow \mathsf{even}(A), \mathsf{all_even}(B).$$

One could query P either with the goal $\mathsf{all_even}(X)$ to find some X such that all elements of X are even or with the goal $\mathsf{all_even}([\mathsf{s}(\mathsf{s}(0))])$ to check whether all elements of the given list are even.

Reasoning about such queries can be implemented using the *resolution rule*, an inference rule with appealing soundness and completeness properties [27]. The resolution rule allows one to replace a rule's body atom by another rule's body, provided that rule's head unifies with the respective body atom. Formally, for a substitution θ:

$$\frac{h \leftarrow p_1,\ldots,p_i,\ldots,p_m \quad h' \leftarrow q_1,\ldots,q_n \quad \theta(h') = \theta(p_i)}{\theta(h \leftarrow p_1,\ldots,p_{i-1},q_1,\ldots,q_n,p_{i+1},\ldots,p_m)}$$

One can use the resolution rule to solve a goal G by contradiction. For example, considering $G = \mathsf{all_even}([\mathsf{s}(\mathsf{s}(0))])$ from the example above, one would negate the goal G to obtain the query $Q = \bot \leftarrow G$, and attempt to derive a contradiction using the resolution rule within the Horn clause theory $P \cup \{Q\}$. A proof for this goal is depicted in Fig. 1.

Powerful programming languages have been built using the resolution rule or variants thereof, for example, Prolog [24] and Datalog [10]. Concrete implementations of programming languages typically do not solve queries by searching for proofs of goals such as in Fig. 1 but use methods like SLD resolution [24]. This typically leads to programs having multiple semantics. Logical semantics interpret logic programs in first-order logic whereas procedural semantics interpret them by reference to the answer search that solvers employ.

3.2.3 Constraint Domains

In this work, we will use a more general version of logic programs than described above, namely constraint logic programs (CLPs). In CLPs, one distinguishes between predicates that have an interpretation in the logic program, i.e., they are defined by the program's Horn clauses, and predicates that have a pre-defined interpretation in a *constraint domain* $\mathfrak{C}$. The latter type of predicates are *constraints*, and a *solver* solves sets of constraints. CLPs support incomplete solvers, i.e., for a set of constraints a solver may return $\top$ (true), $\bot$ (false), or undefined.

CLPs provide two advantages over ordinary logic programs. First, for certain problem domains, e.g., the real numbers, a solver can find solutions much more efficiently than a logic programming engine that just uses the resolution rule. For example, we will use constraints to model signature verification and hash digest comparisons. Second, CLPs allow logic programs to capture infinitely many solutions for a goal, described by a constraint set.

The constraint domain consists of (a) a constraint signature $\Sigma_\mathfrak{C}$, which at least must include $(=)$ as a constraint, (b) a class of constraints $\mathcal{F}_{\Sigma_\mathfrak{C}}$, a set of formulas over $\Sigma_\mathfrak{C}$, (c) a domain of computation $D_\mathfrak{C}$ interpreting the constraints, (d) a constraint theory $T_\mathfrak{C}$ specifying the semantics of the constraints, and (e) a solver $\mathsf{solve}_\mathfrak{C} : 2^{\mathcal{F}_{\Sigma_\mathfrak{C}}} \rightharpoonup \{\top,\bot\}$. We require that the domain of computation is a model of the constraint theory, and that a constraint domain's solver is *sound* in that it returns $\bot$ only if the set of constraints indeed is contradictory, i.e., if $\mathsf{solve}_\mathfrak{C}(c) = \bot$ then $T_\mathfrak{C} \models \neg\exists\vec{v}\bigwedge c$ where $\vec{v}$ binds all free variables occurring in c. Practically, a solver should also be sound in that it returns $\top$ only if the constraints are satisfiable, but this is irrelevant to the semantics of CLPs, as will become clear later.

We call those atoms interpreted in the logic program simply atoms and call atoms that are part of the constraint domain *constraints*.

In CLPs, proofs are encoded as *derivations*. A derivation is a sequence of atoms and constraints $\langle G_1 \mid c_1\rangle \Rightarrow \langle G_2 \mid c_2\rangle \Rightarrow \ldots$. The first element of a derivation step, G_i, is the current list of target atoms, or the *goal*, and the second element c_i is a set of constraints, understood as the conjunction of its elements. To update a derivation step $\langle G_i \mid c_i\rangle$ to $\langle G_{i+1} \mid c_{i+1}\rangle$, one can either:

- Apply the resolution rule and select an atom p_j in G_i and a rule $h \leftarrow b_1,\ldots,b_m$ in the logic program, and then define $c_{i+1} = c_i$ and:

  $$G_{i+1} = (p_1,\ldots,p_{j-1},p_j = h,b_1,\ldots,b_m,p_{j+1},\ldots,p_n).$$

  Note that unification is not performed directly, but is represented by the constraint $p_j = h$.

- Select a constraint p_j in G_i, for which $\mathsf{solve}_\mathfrak{C}(c_i \cup \{p_j\}) \neq \bot$, in which case the derivation's next element

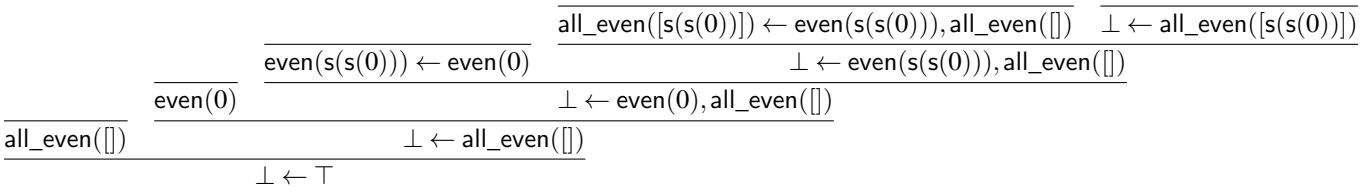


Figure 1: Example proof by contradiction using the resolution rule. We do not explicitly state the respective unifiers but apply them directly. We terminate proof search whenever we apply a rule that is included in the logic program P modulo unification.

is:

$$\langle p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n \mid c_i \cup \{p_j\} \rangle.$$

- Select a constraint p_j in G_i, but $\mathsf{solve}_{\mathfrak{C}}(c_i \cup \{p_j\}) = \bot$. In this case, the derivation's next element is $\langle \emptyset \mid \bot \rangle$.

A derivation is *from* a goal G if its first element is $\langle G \mid \emptyset \rangle$. A derivation is *successful* if its last element is $\langle \emptyset \mid c \rangle$ and $c \neq \bot$. We extend the definition of facts to CLPs in that a fact is any rule of the form $h \leftarrow c_1, \ldots, c_n$ such that $c_1, \ldots, c_n$ are constraints. A set of constraints c is an *answer* for a goal G in a CLP P if there exists a successful derivation from G that ends in $\langle \emptyset \mid c \rangle$.

3.3 The DNS

Overview The DNS is a distributed database that associates typed data with hierarchical names. Names are sequences of one or more labels separated by dots and terminated with the empty root label, e.g., `example.com.` (note the trailing dot separating the label "com" from the empty root label). The name space forms a tree, which has the empty label at its root. Note that in practice, the root label is often omitted and names like `google.com` are understood as equal to `google.com.`.

Data is represented by resource records (RRs) of different types, and the name that is associated with some RR is that RR's *owner name*. A name can have multiple RRs of the same type, and RRs with the same owner name and of the same type form a so-called *RRset*.

The DNS name space is divided into *zones*, which are subtrees of the DNS name space. *Name servers* serve the contents of one or more zones. Name servers use *zone cuts* to delegate a sub-tree of the name space to a different name server. For example, the root zone, the root of which is the name ".", may cut the name space at each TLD, e.g., `com.`, `net.`, etc. The name at the root of a zone is that zone's *apex*.

Clients typically use *recursive resolvers* to query the DNS, which in turn query name servers known to them. When a recursive resolver queries a name server, that name server may respond either (a) with records that answer the query directly, (b) with structural records that inform further recursive resolution, or (c) with an error. In the first case, a recursive resolver will forward the data to the requesting client, and in the second and third cases, the recursive resolver may send follow-up queries to other name servers.

The DNS is described in numerous RFCs and has evolved considerably over the recent decades. One of the most substantial extensions is DNSSEC, which supports the authentication of zone contents by specifying how name servers can sign the contents of the zones they administer. At zone cuts, name servers can indicate which key material may be used to authenticate the child zone. This allows a resolver to validate a zone's authenticity as long as it can authenticate one of the root zone's verification keys and as long as every zone cut indicates a key for the child zone.

RR Types We distinguish between *data RR types* and *structural RR types*. There are many data RR types, e.g., A and AAAA records for IP addresses, TXT for arbitrary text data, and MX records for mail servers. More interesting, though, are structural records. The NS record signals a zone cut, indicating which name server is responsible for the child zone, and the CNAME and DNAME records indicate aliases. While CNAME records indicate direct aliases, e.g., `a.com.` may alias to `b.com.`, a DNAME record maps all names below a name to the corresponding name below the target. For example, a DNAME record for `a.com.` with the value `b.com.` would map a query for `x.y.z.a.com.` to `x.y.z.b.com.`.

DNSSEC and its further extensions added more structural RR types. RRSIG records provide signatures over data, DNSKEY records provide the signature verification keys, and DS records contain digests of child-zone verification keys. DNSSEC also supports the authenticated denial of existence of records using the NSEC and NSEC3 records.

The Wildcard Label Another source of complexity for the DNS is the *wildcard label* "$*$." Conceptually, the wildcard label is simple. When a record owner name uses a wildcard, e.g., `*.example.com.`, responses are synthesized for all names that match this name, e.g., `sub.sub.example.com.`. The wildcard label semantics, however, are complex and subtle and an entire RFC was dedicated to clarifying them [22]. A *wildcard owner name* is a name that has the wildcard as its leftmost label. Name servers will synthesize responses for records with a wildcard owner name whenever (a) the query

name *matches* the wildcard record owner name, and (b) no synthesized part of the query name *exists*.

Matching is defined as expected. For a wildcard owner name "`*.parent`", any subdomain of "`parent`" matches that wildcard owner name. A name is said to exist if it has records associated with it, or it has a child that has records associated with it. For example, when a zone contains records for `x.y.example.`, the names `x.y.example.`, `y.example.`, `example.`, and `.` all exist. Considering `*.example.` and `x.y.example.`, no response would be synthesized for a name containing `y.example.`, as that name exists, even though no records are associated with `y.example.`.

4 Formal Model

We now describe how we formally model the DNS to reason about the resolvability of zones, in particular to determine *whether* a recursive resolver could find an answer in a given set of zone files. Thus, we abstract from malicious name server behavior and transport issues and assume that as long as a recursive resolver can reach a name server, the name server will share all data relevant to a query.

To model the DNS, we compile a zone configuration, a set of initial name servers, and a (possibly empty) set of trusted DS records that bootstrap DNSSEC validation to a CLP, as described in Section 3.2. At a high level, our compilation procedure creates a "base case" fact for every RRset in a given zone, modeling that if a recursive resolver queried the name servers serving this RRset for exactly the RRset's type and owner name, the name servers would respond with that RRset. For the structural record types NS, CNAME, DNAME, and RRSIG, we additionally compile "step case" rules that model how these record types would lead to follow-up queries by a recursive resolver.

In the base case compilation, we treat the RRsets that are part of the zone configuration as unauthenticated. Our compiler also adds base case rules for the trusted DS records and treats them as authentic, thereby bootstrapping DNSSEC validation.

We compile the structural record types NS, CNAME, DNAME, and RRSIG to step case rules as follows. The rules for NS records model the propagation of responses from a name server serving a delegated child zone to the name server serving the parent zone. The rules for CNAME and DNAME records model the propagation of responses from queries for the target name to queries for the alias. Finally, the rules for RRSIG records can turn unauthenticated responses into authenticated ones.

We describe this compilation procedure in the following paragraphs. We start by describing how we represent zone data itself. The representation that we choose can be easily obtained from a zone's textual representation. Next, we describe how we encode names and data as ground terms. Finally, we describe how we compile RRsets to Horn clauses.

4.1 The DNS

We represent a set of zones as follows. We consider a finite set of labels $\mathcal{L}$ containing the empty label ε and the wildcard label "$*$". A *name* over $\mathcal{L}$ is a non-empty sequence of labels, separated by dots "." and terminated by ε. We use $\cdot$ to denote the concatenation of names, which adds a "." between consecutive labels. For example, `example.com.` $=$ `example`$\cdot$`com`$\cdot\varepsilon$.

Definition 4.1. The set of *names* is the smallest set $\mathcal{N}$ such that:

- $\varepsilon \in \mathcal{N}$, and
- if $n \in \mathcal{N}$ and $l \in \mathcal{L} \setminus \{\varepsilon\}$, then $l \cdot n \in \mathcal{N}$.

The DNS associates names with typed data. We use the set of symbols $\mathbb{T}$ for the relevant types:

$$\mathbb{T} = \{\mathsf{A}, \mathsf{AAAA}, \mathsf{NS}, \mathsf{CNAME}, \mathsf{DNAME}, \mathsf{RRSIG}, \mathsf{DNSKEY}, \mathsf{DS}, \mathsf{NSEC}, \mathsf{NSEC3}, \dots\}.$$

Here, we only list types that inform recursive resolution. There are more types, e.g., MX for email communication, which we do not list here.

We represent the data itself as complex terms where we use the corresponding symbol in $\mathbb{T}$ as a function symbol and the function arguments represent the individual fields of the data. For example, an NS record delegates the resolution of its owner name to a server s and has one field. The data of this NS record would be represented as $\mathsf{NS}(s)$. An RRSIG record stores the signature for a set of other resource records, and thus has many fields, including a tag for the signing key, a reference to a signing algorithm, the signature itself, and more. We use $\mathcal{D}$ to refer to the set of all data.

We identify zones by their apex in $\mathcal{N}$ and represent them as a partial function $z : \mathcal{N} \rightharpoonup \mathbb{T} \rightharpoonup 2^{\mathcal{D}}$, which associates names and types with the respective RRsets. Zones are served by different name servers. A *zone configuration* c specifies which name servers serve which zones, and c is a partial function $c : 2^{\mathcal{N}} \rightharpoonup \mathcal{N} \rightharpoonup \mathcal{N} \rightharpoonup \mathbb{T} \rightharpoonup 2^{\mathcal{D}}$ where $c(s,n) = z$ means that name server set s serves zone z, which has the apex n.

4.1.1 Zone Well-Formedness

We assume that zone files are well-formed in that they adhere to basic RFC guidelines, which DecDNS checks before compiling them. For example, RFC 1034 allows at most one CNAME record associated with an owner name [25, Sec. 3.6.2]. Critically, the well-formedness checks require no advanced reasoning and are always local to each zone. See Appendix C for their full description.

4.2 Modeling Names and Data

4.2.1 Representing Names

We represent names as lists of labels, and labels in $\mathcal{L}$ as constants. Effectively, we assume that $\mathcal{L} \subseteq \mathcal{C}$. We define the translation function $\mathsf{name} : \mathcal{N} \cup \mathcal{N}^* \to \mathcal{T}_\Sigma$, which translates names into lists of labels, i.e., terms over Σ. We omit the empty label as it is implicit.[1] The root name "." is thus represented by the empty list.

$$\mathsf{name}(l_1 \cdot \ldots \cdot l_n \cdot \varepsilon) = \begin{cases} [l_n, \ldots, l_2, *] & l_1 \in \mathcal{L}^* \\ [l_n, \ldots, l_1] & \text{otherwise.} \end{cases}$$

We homomorphically extend name to sets of names, representing the set as a list, ordered by the canonical ordering of names as specified in RFC 4034 [29].

Representing names as lists allows unification to be used for synthesis, DNAME rewrites, and checking whether one domain is a subdomain of another. For example, checking whether the name `a.example.com.` is a subdomain of `example.com.` can be easily realized by checking whether the lists $[\texttt{com}, \texttt{example} \mid _]$ and $[\texttt{com}, \texttt{example}, \texttt{a}]$ unify. Note that the latter list corresponds to $\mathsf{name}(\texttt{a.example.com.})$, but the former one does not correspond to $\mathsf{name}(\texttt{example.com.})$.

Semantics of Wildcard Labels Due to wildcard labels, DNS zones can effectively be very large. In a naïve model, one could represent the semantics of wildcard owner names by the set of all ground names for which wildcard synthesis would occur, though such sets would be unwieldily large.

To represent the semantics of wildcard owner names more efficiently, we use sets of *wildcard constraints*, which represent labels for which wildcard synthesis does *not* occur (see Section 3.3). These sets replace wildcard labels and thus form their own kind of labels.

Definition 4.2. The set of *wildcard-constraint labels* $\mathcal{L}^*$ is defined as:

$$\mathcal{L}^* = \{S \cup \{*\} \mid S \subseteq \mathcal{L}\}.$$

Observe that every wildcard-constraint label contains the wildcard label $*$ itself. This is because the literal wildcard owner name, by definition, always exists. Thus, no name server would perform synthesis for a name containing the wildcard owner name.

We extend the set of names $\mathcal{N}$ to $\mathcal{N}^*$, which will use labels from $\mathcal{L}^*$ in their leftmost label.

Definition 4.3. The set of *wildcard-constraint names* is the smallest set $\mathcal{N}^*$ such that:

- Let $n = l_1 \cdot \ldots \cdot l_k \in \mathcal{N}$. If $l_1 \neq *$, then $n \in \mathcal{N}^*$.
- Let $* \cdot l_2 \cdot \ldots \cdot l_k \in \mathcal{N}$ and $l \in \mathcal{L}^*$. $l \cdot l_2 \cdot \ldots \cdot l_k \in \mathcal{N}^*$.

For example, assuming that only the name `y.example.` exists below `example`, the name `*.example.` would be represented as $\{\texttt{y}, *\}\texttt{.example.}$.

We lift zone configurations c from $2^{\mathcal{N}} \rightharpoonup \mathcal{N} \rightharpoonup \mathcal{N} \rightharpoonup \mathbb{T} \rightharpoonup 2^{\mathcal{D}}$ to $2^{\mathcal{N}} \rightharpoonup \mathcal{N} \rightharpoonup \mathcal{N}^* \rightharpoonup \mathbb{T} \rightharpoonup 2^{\mathcal{D}}$ by replacing every name in the third argument, representing an RRset owner name, with the corresponding name in $\mathcal{N}^*$, following Definition 4.3. The concrete constraints for a wildcard owner name $* \cdot n$ can easily be computed within c.

Whenever the wildcard synthesis of responses must be considered in our formal model, we represent wildcard owner names, i.e., names where $l_1 \in \mathcal{L}^*$, by lists in which we represent the wildcard label with a variable $v \in \mathcal{V}$. Whenever we use such lists in CLP rules, we represent the wildcard constraints $l \in l_1$ by inequality constraints $v \neq l$ in the rule body. For example, consider again the example wildcard-constraint name $\{\texttt{y}, *\}\texttt{.example.}$. In a CLP rule, we would model when wildcard synthesis occurs by a rule of the following structure:

$$p([\texttt{example}, V \mid _]) \leftarrow V \neq \texttt{y}, V \neq *.$$

More formally, we define the function $\mathsf{name}_{v,w} : \mathcal{N} \cup \mathcal{N}^* \to \mathcal{T}_\Sigma$ that replaces wildcard-constraint labels with a variable $v \in \mathcal{V}$ and uses a variable $w \in \mathcal{V}$ for the list tail:

$$\mathsf{name}_{v,w}(l_1 \cdot \ldots \cdot l_n \cdot \varepsilon) = \begin{cases} [l_n, \ldots, l_2, v \mid w] & l_1 \in \mathcal{L}^* \\ [l_n, \ldots, l_1] & \text{otherwise.} \end{cases}$$

The function $\mathsf{C}_v : \mathcal{N} \cup \mathcal{N}^* \to 2^{\mathcal{F}}$ associates a name with a set of constraints for v like those occurring in the rule body above. The constraints represent all labels for which no synthesis will occur:

$$\mathsf{C}_v(l_1 \cdot n) = \begin{cases} \{v \neq l \mid l \in l_1\} & l_1 \in \mathcal{L}^* \\ \emptyset & \text{otherwise.} \end{cases}$$

4.2.2 Representing Data

To represent data, we introduce the function $\mathsf{data} : 2^{\mathcal{D}} \to \mathcal{T}_\Sigma$. For an RRset $\{f(x_1, \ldots, x_n), \ldots\} \in 2^{\mathcal{D}}$, data maps that set to a list, representing each of the RRset's elements in no particular order, using the respective function term $f/n \in \mathcal{G}$, and representing the concrete data item x_i either as an atom or as $\mathsf{name}(x_i)$, if x_i is a name.

4.3 A Formal Model of Recursive Resolution

In this section, we describe how we translate a zone configuration into a logic program that allows one to reason about the resolvable queries for that configuration.

[1] By Definition 4.1, every name in $\mathcal{N}$ is terminated by the empty label.

4.3.1 Logic Program Domain

Our logic program uses the following function symbols $\mathcal{G}$:

- $\cdot/2$ and $[]/0$ for lists.
- $\bot/0$ and $\top/0$ for a Boolean.
- $0/0$ and $1/0$ for a bit.
- $\mathsf{referral}/0$ to model when a name server serves non-authoritative data for referrals, e.g., NS or glue records.
- All symbols in $\mathbb{T}$, with the respective arity of the data, to represent data.

We will use $x \notin \{a,b,c,\dots\}$ as syntactic sugar for $x \neq a, x \neq b, x \neq c, \dots$. Furthermore, our logic program will use the following predicate symbols $\mathcal{P}$:

- $\mathsf{member}/2$ for list inclusion,
- $\mathsf{min}/3$ for the conjunction of two AD bits,
- $\mathsf{resolves}/8$ to model recursive resolution considering response data, and
- $\mathsf{resolvable}/3$ to model which queries are resolvable without considering the data.

$\mathsf{resolves}(s,t,z,n,d,r,v,a)$ models that querying name server set s with query type t for query name n results in the data d. Furthermore, z indicates the zone through which the answer was inferred, $r \in \{\bot,\top\}$ indicates whether an alias was followed, $v \in \{\bot,\top\}$ indicates whether the resolver is performing DNSSEC validation, and $a \in \{0,1\}$ indicates whether the query answer could be validated as authentic. Note that $v = \bot$ implies that $a = 0$; one cannot validate an answer as authentic without performing validation. When $v = \top$, a can be either 0 or 1. If $a = 1$, the resolver validated the answer as authentic. If $a = 0$, the resolver accepted the answer as insecure because it validated a delegation as insecure. Note that $\mathsf{resolvable}(t,n,v)$ effectively is a projection of $\mathsf{resolves}(s,t,z,n,d,r,v,a)$, as will become clear later.

4.3.2 Constraint Domain

To model the DNS, we also use the following constraints:

- $(\neq)/2$ for inequality,
- $\mathsf{isName}/1$ for whether a term corresponds to a valid name, e.g., checking that its length remains within the bounds of the DNS,
- $\mathsf{digestValid}/3$ and $\mathsf{signatureValid}/4$ for digest and signature verification, as well as
- $\mathsf{nsecUnsigned}/2$, $\mathsf{closestEncloser}/5$, and $\mathsf{nsec3Unsigned}/3$ for verifying proofs of insecure delegations.

We model $(\neq)$ and isName as constraints to represent large zones efficiently. For example, some DNAME and wildcard records can lead to a vast number of queries being resolvable, which we represent symbolically using the above two predicates.

We encode DNSSEC-related checks using constraints because (a) we solve them only when they are instantiated on ground terms, and expressing them as CLP constraints provides an efficient way to achieve that, and (b) implementing, e.g., signature verification is much more straightforward in an ordinary programming language than in Horn clauses.

Formally, our logic program will use the constraint signature $\Sigma_{\mathfrak{C}} = (\mathcal{V}, \mathcal{C}, \mathcal{G}, P)$, where P contains the above predicates. Our constraint domain's domain of computation $D_{\mathfrak{C}}$ uses the universe of all ground terms. It interprets $(\neq)$ as the complement of the identity relation, and it interprets all other constraints directly as agreeing with the solver implementation, which we will describe in Section 7. To obtain the constraint theory $T_{\mathfrak{C}}$, we follow the straightforward construction from [18], and obtain it as the theory of all first-order formulae true in $D_{\mathfrak{C}}$.

4.3.3 The Formal Model

To reason about the resolvability of a zone configuration $c : 2^{\mathcal{N}} \rightharpoonup \mathcal{N} \rightharpoonup \mathcal{N}^* \rightharpoonup \mathbb{T} \rightharpoonup 2^{\mathcal{D}}$, we compile the configuration's RRsets to a CLP $\Gamma(s_0, k_0, c)$, where $s_0 \subseteq \mathcal{N}$ is a set of initial name servers and $k_0 \subseteq \mathcal{D}$ is a set of trusted DS records that bootstraps DNSSEC validation. We assume that k_0 is for the apex of c's topmost zone $z_0 \in \mathcal{N}$. We compile every RRset into a "base case" rule, which models that if a recursive resolver queries a name server s for some name n and query type t, it will respond with the data that it has stored for that name and type, and that data will be unauthenticated. Moreover, we compile every RRset with a structural record type into "step case" rules, which model how a recursive resolver interprets the respective structural record, e.g., leading to recursive queries.

Let $s \subseteq \mathcal{N}$ be a set of name servers, $z \in \mathcal{N}$ a zone name, $n \in \mathcal{N}^*$ an owner name, and $t \in \mathbb{T}$ a record type, such that $(s,z,n,t,d) \in c$. In the following, we will define a function $\gamma_{s_0} : 2^{\mathcal{N}} \to \mathcal{N} \to \mathcal{N}^* \to \mathbb{T} \to 2^{\mathcal{D}} \to 2^{\mathcal{F}}$ that compiles (s,z,n,t,d) to a set of Horn clauses. The Horn clauses $\Gamma(s_0,k_0,c)$ associated with a zone configuration c then include a rule $\mathsf{R}_{\mathsf{resolvable}}$ that projects resolves to resolvable atoms, rules $A(s_0,k_0,z_0)$ for the trusted DS records, and the images of γ_{s_0} over c:

$$\Gamma(s_0,k_0,c) = \{\mathsf{R}_{\mathsf{resolvable}}\} \cup A(s_0,k_0,z_0) \cup \bigcup_{(s,z,n,t,d)\in c} \gamma_{s_0}(s,z,n,t,d).$$

We project resolves atoms to resolvable atoms, provided that the response was obtained from the initial name servers,

not referral data, and that the query type was for a non-structural record type, i.e., not in the set:

$$\mathbb{T}_S = \{\mathsf{SOA}, \mathsf{NS}, \mathsf{CNAME}, \mathsf{DNAME}, \mathsf{DS}, \mathsf{DNSKEY}, \mathsf{NSEC}, \mathsf{NSEC3}, \mathsf{RRSIG}, \mathsf{NSEC3PARAM}\}.$$

We opted to exclude reasoning about the resolvability of structural record types because their purpose is not to be resolvable by themselves but to enable the (authenticated) resolvability of other records.

$$\mathsf{R}_{\mathsf{resolvable}} = \mathsf{resolvable}(Q,N,V) \leftarrow Q \notin \mathbb{T}_S, \mathsf{resolves}(\mathsf{name}(s_0), Q, Z, N, _, _, V, _), Z \neq \mathsf{referral}.$$

In the following, we will define γ_{s_0} by a case distinction. In each case, we will constrain which formulas must be in γ_{s_0}'s image. The image of γ_{s_0} is then understood to be the smallest set satisfying all constraints. In some cases, parts of a rule will be similar. To highlight them, we will underline those parts that differ.

Base Case Rules For every RRset in c's image, we add a rule to represent that when we query a set of name servers for a query name and query type, they will respond with the RRset matching the query exactly. We choose $zone \in \{\mathsf{referral}, \mathsf{name}(z)\}$ according to whether the owner name n is indeed part of zone z or a referral. As the name servers may perform synthesis, we add one rule covering a query for the literal wildcard owner name, and one rule considering synthesis. Note that if n is not a wildcard owner name, both rules will be equal.

$$\mathsf{resolves}(\mathsf{name}(s), t, zone, \mathsf{name}(n), \mathsf{data}(d), \bot, \bot, 0). \in \gamma_{s_0}(s,z,n,t,d)$$

$$\mathsf{resolves}(\mathsf{name}(s), t, zone, \mathsf{name}_{L,N}(n), \mathsf{data}(d), \bot, \bot, 0) \leftarrow \mathsf{C}_L(n), \mathsf{isName}(\mathsf{name}_{L,N}(n)). \in \gamma_{s_0}(s,z,n,t,d)$$

We compile the trusted DS records similarly to the base case rules above. If $k_0 = \emptyset$, then $A(s_0,k_0,z_0) = \emptyset$. Otherwise, we compile the following rule to bootstrap DNSSEC validation:

$$A(s_0,k_0,z_0) = \{\mathsf{resolves}(\mathsf{name}(s_0), \mathsf{DS}, \mathsf{name}(z_0), \mathsf{name}(z_0), \mathsf{data}(k_0), \bot, \top, 1)\}.$$

Step Case Rules The compilation of structural records into "step case" rules depends on the record type.

NS Let $t = \mathsf{NS}$ and $d = \{\mathsf{NS}(\vec{s}_1), \ldots, \mathsf{NS}(\vec{s}_k)\}$, let $\vec{s} = \{\vec{s}_1, \ldots, \vec{s}_k\}$ be the set of name servers responsible for the child, and let $\mathsf{name}(n) = [l_n, \ldots, l_1]$ be the delegated name.

We compile NS records into multiple rules whenever $n \neq z$, i.e., the NS records are not at the zone's apex, and these rules can be grouped into two categories. First, we compile rules that model the propagation of data from a child to the parent. Intuitively, this models that one can use the NS records from the parent zone to obtain data from the child zone, provided one can also obtain one of the respective servers' IP addresses. We compile the following rule for each child name server $\vec{s}_i$, address record type $t' \in \{\mathsf{A}, \mathsf{AAAA}\}$, and address lookup *glue*, with one restriction, which we explain below.

$$glue \in \{\mathsf{resolves}(\underline{\mathsf{name}(s)}, t', \underline{\mathsf{referral}}, \mathsf{name}(\vec{s}_i), _, \bot, \bot, 0), \mathsf{resolves}(\underline{\mathsf{name}(s_0)}, t', _, \mathsf{name}(\vec{s}_i), _, \bot, \bot, 0)\}$$

$$\mathsf{resolves}(\underline{\mathsf{name}(s)}, T, \underline{\mathsf{name}(z)}, [l_n, \ldots, l_1 \mid N], D, R, V, A) \leftarrow \mathsf{isName}([l_n, \ldots, l_1 \mid N]), glue, \mathsf{resolves}(\underline{\mathsf{name}(\vec{s})}, T, \underline{\mathsf{name}(n)}, [l_n, \ldots, l_1 \mid N], D, R, V, A). \in \gamma_{s_0}(s,z,n,\mathsf{NS},d)$$

Note that the name server address query *glue* does not follow aliases, which is required by RFC 2181 [13].

To model how resolvers obtain the name server's IP address, we distinguish whether the name server $\vec{s}_i$ sits below the delegated name n. If $\vec{s}_i$ is at or below n, the recursive resolver expects the parent zone's name servers s to serve glue records. Thus, we only emit the rule that queries the name server address via the initial name servers s_0 if $\vec{s}_i$ is not at or below n.

Second, we compile rules that model the propagation of unauthenticated answers through insecure delegations while performing validation. A resolver can establish that a delegation is insecure by validating an NSEC or NSEC3 proof. NSEC and NSEC3 records provide authenticated denial of existence, and they indicate which types exist for a given owner name. When they indicate that no DS records exist at a zone cut, a resolver cannot obtain authentic child zone verification keys through the configured trust anchors, and can thus accept insecure responses [8, 28, 29].

The difference between NSEC and NSEC3 records is that NSEC records allow one to enumerate all names that occur in a zone, which can be prevented when using NSEC3. This is achieved by hashing the original owner names, which, though, changes how resolvers can authenticate that there is no DS record at a zone cut.

There are three ways in which a resolver can learn that a delegation is insecure, and thus we compile the rule below for each choice of $insecure \in \{nsec, nsec3_a, nsec3_b\}$. The rule models that if a resolver can validate a delegation to be insecure (modeled by *insecure*) they will propagate insecure

data as validated to be insecure. Note that the validation flag changes from $\bot$ in the body to $\top$ in the head.

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), [l_n, \ldots, l_1 \mid N], D, \bot, \top, 0) \leftarrow \\ &\quad T \neq \mathsf{DS}, \\ &\quad \mathsf{isName}([l_n, \ldots, l_1 \mid N]), \\ &\quad \mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), [l_n, \ldots, l_1 \mid N], D, \bot, \bot, 0), \\ &\quad \mathit{insecure}. \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{NS}, d)\end{aligned}$$

In the model for *insecure*, the resolves predicates obtain the records used to prove that the delegation is insecure, and the constraints nsecUnsigned, nsec3Unsigned, and closestEncloser model the necessary checks on those records.

$$\begin{aligned}&\mathit{nsec} = \\ &\mathsf{resolves}(\mathsf{name}(s), \mathsf{NSEC}, \mathsf{name}(z), \mathsf{name}(n), [D_{\mathsf{NSEC}}], \bot, \top, 1), \\ &\qquad\qquad \mathsf{nsecUnsigned}(\mathsf{name}(n), D_{\mathsf{NSEC}})\end{aligned}$$

NSEC3 records have particular owner names, which hide the records' original owner names. Namely, their owner names are the name of the zone with a hash prepended, for example, `S83...5D8.example.com.`. For a parent zone with apex $z = l_1 \cdot \cdots \cdot l_n$, we represent such names with the two lists below, each containing a variable for the hashed label:

$$n_1 = [l_n, \ldots, l_1, L_1] \quad n_2 = [l_n, \ldots, l_1, L_2]$$

$$\begin{aligned}&\mathit{nsec3}_a = \\ &\mathsf{resolves}(\mathsf{name}(s), \mathsf{NSEC3}, \mathsf{name}(z), n_1, [D_{\mathsf{NSEC3}}], \bot, \top, 1), \\ &\qquad\qquad \mathsf{nsec3Unsigned}(\mathsf{name}(n), n_1, D_{\mathsf{NSEC3}})\end{aligned}$$

$$\begin{aligned}&\mathit{nsec3}_b = \\ &\mathsf{resolves}(\mathsf{name}(s), \mathsf{NSEC3}, \mathsf{name}(z), n_1, [D_{\mathsf{NSEC3}}], \bot, \top, 1), \\ &\mathsf{resolves}(\mathsf{name}(s), \mathsf{NSEC3}, \mathsf{name}(z), n_2, [D_{\mathsf{NSEC3}}'], \bot, \top, 1), \\ &\qquad \mathsf{closestEncloser}(\mathsf{name}(n), n_1, D_{\mathsf{NSEC3}}, n_2, D_{\mathsf{NSEC3}}')\end{aligned}$$

As per [22], the semantics of NS records with a wildcard owner name are undefined, and we thus do not consider the wildcard synthesis of responses for NS records.

CNAME Let $t = \mathsf{CNAME}$ and $d = \{\mathsf{CNAME}(\vec{n})\}$. For CNAME records, we model that if we can resolve the target name $\vec{n}$ starting from the initial name servers, we can also resolve the alias at the name servers s serving the CNAME record. Note that there must be only one CNAME record per owner name, which is why we only consider d to contain one element above. We forbid the resolution of CNAME queries using this rule ($T \neq \mathsf{CNAME}$) as those queries would be answered by the corresponding CNAME record directly, and forbid the resolution of DNSSEC-related records $\mathbb{T}_{\mathsf{SEC}}$, as their resolution does not follow aliases:

$$\mathbb{T}_{\mathsf{SEC}} = \{\mathsf{DS}, \mathsf{DNSKEY}, \mathsf{NSEC}, \mathsf{NSEC3}, \mathsf{RRSIG}\}.$$

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), \mathsf{name}(n), D, \top, \bot, 0) \leftarrow \\ &\quad T \notin \{\mathsf{CNAME}\} \cup \mathbb{T}_{\mathsf{SEC}}, \\ &\quad \mathsf{isName}(\mathsf{name}(n)), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), T, Z, \mathsf{name}(\vec{n}), D, _, \bot, 0), \\ &\quad Z \neq \mathsf{referral}. \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{CNAME}, d)\end{aligned}$$

Similarly to our model of NS records, we model that CNAME records propagate the authentication status of aliases. A response for a query to an aliased name can be authenticated if both the query for the CNAME record and for the alias target can be authenticated. If either query has an insecure response, the combined response is insecure as well, which we model using the variable A and the min predicate below.

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), \mathsf{name}(n), D, \top, \top, A) \leftarrow \\ &\quad T \notin \{\mathsf{CNAME}\} \cup \mathbb{T}_{\mathsf{SEC}}, \\ &\quad \mathsf{isName}(\mathsf{name}(n)), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), \mathsf{CNAME}, _, \mathsf{name}(n), \mathsf{data}(d), \bot, \top, A_1), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), T, Z, \mathsf{name}(\vec{n}), D, _, \top, A_2), \\ &\quad \min(A_1, A_2, A), Z \neq \mathsf{referral}. \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{CNAME}, d)\end{aligned}$$

CNAME records may have wildcard owner names. We thus also add rules to $\gamma_{s_0}(s, z, n, \mathsf{CNAME}, d)$ that consider the synthesis of responses. We compile these rules just as before but with the difference that we add the constraints $\mathsf{C}_L(n)$ to the rule body and replace $\mathsf{name}(n)$ with $\mathsf{name}_{L,N}(n)$ outside the query for the validated CNAME record.

DNAME Let $t = \mathsf{DNAME}$, $d = \{\mathsf{DNAME}(\vec{n})\}$, $\mathsf{name}(n) = [l_n, \ldots, l_1]$, and $\mathsf{name}(\vec{n}) = [\vec{l}_m, \ldots, \vec{l}_1]$. We compile DNAME records analogously to CNAME records, with the only difference being that names are rewritten according to the DNAME rules [30] and that we allow for resolving DNAME queries. We model the rewriting of DNAME records using the variables L and N, which capture the non-empty list suffix $[L \mid N]$, corresponding to the rewritten name prefix. We add the following two rules, the latter of which again models the propagation of the alias's authentication status.

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), [l_n, \ldots, l_1, L \mid N], D, \top, \bot, 0) \leftarrow \\ &\quad T \notin \mathbb{T}_{\mathsf{SEC}}, \\ &\quad \mathsf{isName}([l_n, \ldots, l_1, L \mid N]), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), T, Z, [\vec{l}_m, \ldots, \vec{l}_1, L \mid N], D, _, \bot, 0), \\ &\quad Z \neq \mathsf{referral}. \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{DNAME}, d)\end{aligned}$$

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), T, \mathsf{name}(z), [l_n, \ldots, l_1, L \mid N], D, \top, \top, A) \leftarrow \\ &\quad T \notin \mathbb{T}_{\mathsf{SEC}}, \\ &\quad \mathsf{isName}([l_n, \ldots, l_1, L \mid N]), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), \mathsf{DNAME}, _, \mathsf{name}(n), \mathsf{data}(d), \bot, \top, A_1), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), T, Z, [\vec{l}_m, \ldots, \vec{l}_1, L \mid N], D, _, \top, A_2), \\ &\quad \min(A_1, A_2, A), Z \neq \mathsf{referral}. \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{DNAME}, d)\end{aligned}$$

We reject DNAME records with wildcard owner names, following the recommendation in [22], and therefore need not consider them in our compilation.

RRSIG Finally, we model the validation of responses via DNSSEC, using RRSIG records. Let $t = \mathsf{RRSIG}$ and $d = \{\mathrm{sig}, \ldots\}$ where sig is RRSIG data covering type t_{COV} and signed by the operator of zone n_{SIG}. We distinguish between validation of DNSKEY and other record types as well as between the validation of records with wildcard owner names and other owner names. The difference is that DNSKEY RRsets contain their own verification key, while one must first obtain an authentic verification key for other record types. If $t_{\mathrm{COV}} \neq \mathsf{DNSKEY}$ and the record does not have a wildcard owner name, we add a rule for each data item sig within d that (a) looks up the data to be validated, (b) looks up an authenticated verification key RRset, (c) non-deterministically chooses one of the keys in the RRset with the member predicate, and (d) verifies the signature with that key with the signatureValid constraint.

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), t_{\mathrm{COV}}, \mathsf{name}(z), \mathsf{name}(n), D, \bot, \underline{\top}, 1) \leftarrow \\ &\quad \mathsf{resolves}(\mathsf{name}(s), t_{\mathrm{COV}}, \mathsf{name}(z), \mathsf{name}(n), D, \bot, \underline{\bot}, 0), \\ &\quad \mathsf{resolves}(\mathsf{name}(s), \mathsf{DNSKEY}, \mathsf{name}(z), \mathsf{name}(n_{\mathrm{SIG}}), D_{\mathrm{KEY}}, \bot, \top, 1), \\ &\quad \mathsf{member}(K, D_{\mathrm{KEY}}), \\ &\quad \mathsf{signatureValid}(\mathsf{name}(n), D, \mathrm{sig}, K). \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{RRSIG}, d)\end{aligned}$$

If the RRSIG record has a wildcard owner name, we generate the same rule as above but replace all occurrences of $\mathsf{name}(n)$ outside the signatureValid constraint with $\mathsf{name}_{L,N}(n)$ and add the corresponding constraints $\mathsf{C}_L(n)$ to the rule body. While the respective RFCs contain explicit mechanisms for signature verification for synthesized responses, we perform the signature check directly on the record owner name instead of the synthesized name.

Note that the validation of responses synthesized from a wildcard owner name also requires validating that no non-synthesized answer to the query exists, which we omit in our model for performance reasons. If a direct answer to the query existed, the name server would not synthesize the response. Thus, if a resolver omits the check of non-existence, an attacker could use the signature over a record with a wildcard owner name to "overwrite" the name server's actual response.

As DNSKEY RRsets contain their own verification key, we cannot authenticate that key by performing another DNSKEY lookup. DNSKEY verification keys are authenticated by authentic DS RRsets, which replace the DNSKEY lookup from the rule above in our model for DNSKEY RRset validation. Thus, if $t_{\mathrm{COV}} = \mathsf{DNSKEY}$, then we add the rule below for each data item sig within d.

$$\begin{aligned}&\mathsf{resolves}(\mathsf{name}(s), \mathsf{DNSKEY}, \mathsf{name}(z), \mathsf{name}(n), D, \bot, \underline{\top}, 1) \leftarrow \\ &\quad \mathsf{resolves}(\mathsf{name}(s), \mathsf{DNSKEY}, \mathsf{name}(z), \mathsf{name}(n), D, \bot, \underline{\bot}, 0), \\ &\quad \mathsf{resolves}(\mathsf{name}(s_0), \mathsf{DS}, _, \mathsf{name}(n), D_{\mathrm{DS}}, \bot, \top, 1), \\ &\quad \mathsf{member}(K, D), \\ &\quad \mathsf{member}(K_{\mathrm{DIG}}, D_{\mathrm{DS}}), \\ &\quad \mathsf{digestValid}(\mathsf{name}(n), K, K_{\mathrm{DIG}}), \\ &\quad \mathsf{signatureValid}(\mathsf{name}(n), D, \mathrm{sig}, K). \\ &\qquad\qquad \in \gamma_{s_0}(s, z, n, \mathsf{RRSIG}, d)\end{aligned}$$

We do not consider the case that a DNSKEY RRset is synthesized from an RRset with a wildcard owner name. This is because DNSKEY RRsets must only occur at the apex of a zone, which requires delegation by an NS record with the same owner and thus wildcard owner name, which has undefined semantics [22].

5 Change-Impact Analysis

In this section, we present an algorithm to compute a change-impact analysis for two CLPs. This algorithm proceeds in two steps. First, it computes all facts that are a *consequence* of the CLP, i.e., everything that is resolvable in a given zone configuration. Second, it computes the difference between the two sets of resolvable queries. Note that this difference cannot be computed by a simple set difference as the computed facts may contain residual constraints, for example, representing wildcard synthesis. We will prove soundness and completeness of our algorithm in Section 6.

5.1 Resolvable Queries

For a CLP P, a fact $a \leftarrow c$ is an *immediate consequence* [18] of a set of facts F using a rule $R = h \leftarrow b_1, \ldots, b_n \in P$ if $a = h$ and there is a derivation $\langle b_1, \ldots, b_n \mid \emptyset \rangle \Rightarrow \cdots \Rightarrow \langle \emptyset \mid c \rangle$ in F. Intuitively, a rule has an immediate consequence with respect to a set of facts F if all predicates in the rule body but not the constraints can be unified consistently with some fact in F. $S_R(F)$ denotes the set of F's procedural immediate consequences using R, and $S_P(F)$ denotes a program's procedural immediate consequences, defined as:

$$S_P(F) = \bigcup_{R \in P} S_R(F).$$

For a set of initial name servers s_0 and trusted DS records k_0, all resolvable queries $Q_{\mathsf{resolvable}}(s_0, k_0, c)$ of zone configuration c can then be computed as the least fixpoint lfp of the

S_P operator, i.e., as the $\subseteq$-smallest set of facts F for which $S_P(F) = F$.

$$Q_{\mathsf{resolvable}}(s_0, k_0, c) = \{a \leftarrow \Psi \in \mathsf{lfp}(S_{\Gamma(s_0,k_0,c)}) \mid a \text{ is a } \mathsf{resolvable} \text{ atom}\}.$$

5.2 Change-Impact Analysis

The change-impact analysis for a zone configuration update can be computed from $Q_{\mathsf{resolvable}}$. Given a zone configuration c and its updated version c', the goal is to compute the symmetric difference of all ground instances entailed by $Q_{\mathsf{resolvable}}(s_0, k_0, c)$ and $Q_{\mathsf{resolvable}}(s_0', k_0', c')$, using the initial name servers and trust anchors for each version. Note that we cannot directly compute the symmetric difference of $Q_{\mathsf{resolvable}}$'s images as they contain facts with potentially different constraints.

We define the operator $(\setminus) : \mathcal{F} \rightharpoonup \mathcal{F} \rightharpoonup 2^{\mathcal{F}}$. $\phi_1 \setminus \phi_2$ is a set of formulas entailing all ground instances of ϕ_1 without the ground instances of ϕ_2, provided that ϕ_1 and ϕ_2 are both facts.

In general, $(\setminus)$ returns a set of formulas, not a set of facts, and these formulas may not even be CLP rules anymore. Consider, for example, $p(X) \setminus p(f(Y))$. The entailed ground terms are all terms that do not use f as a top-level symbol in p's first argument, i.e., $p(X) \leftarrow \neg\exists Y.p(X) = p(f(Y))$, which can be rewritten to $p(X) \leftarrow \forall Y.p(X) \neq p(f(Y))$.

We thus model subtraction using universally quantified inequality constraints. As $\forall$ is part of the constraint domain's constraint language, and $\neq$ is a constraint, our implementation will treat conjuncts as constraints, and simplify them following some simple heuristics.

When computing $\phi_1 \setminus \phi_2$ there are two cases. Either a ground term is in ϕ_1 but not in ϕ_2 because it does not unify with ϕ_2's head, or, if it does unify with ϕ_2's head, because it violates one of the constraints in ϕ_2. In general, the result is a disjunction. We define $(\setminus)$ in Algorithm 1.

Algorithm 1 Algorithm to compute $\phi_1 \setminus \phi_2$. We assume that ϕ_1 and ϕ_2 share no variables.

1: Let $\phi_1 = h_1 \leftarrow \Psi_1$, $\phi_2 = h_2 \leftarrow \Psi_2$, $\theta = \mathsf{MGU}(h_1, h_2)$.
2: **if** no such θ exists **then**
3: **return** $\{\phi_1\}$
4: **else**
5: Let $\hat{u} = \{h_1 \leftarrow \{\forall \mathsf{vars}(h_2).h_1 \neq h_2\} \cup \Psi_1\}$.
6: Let $u = \{\theta(h_1) \leftarrow \theta(\Psi_1) \cup \{\theta(\neg c)\} \mid c \in \Psi_2\}$.
7: **return** $u \cup \hat{u}$
8: **end if**

First, to capture that the heads do not unify, Algorithm 1 states that they must not unify as a universally quantified inequality constraint in line 5. Second, to capture when the heads unify but one of the constraints is false, we unify the heads and negate each constraint of ϕ_2 in line 6.

We extend $(\setminus)$ to finite sets of formulas $\Gamma_1, \Gamma_2 \subseteq \mathcal{F}$ as follows:

$$\Gamma_1 \setminus \psi := \bigcup_{\phi \in \Gamma_1} \phi \setminus \psi$$

$$\Gamma_1 \setminus \Gamma_2 := ((\Gamma_1 \setminus \psi_1) \setminus \ldots) \setminus \psi_n, \quad \Gamma_2 = \{\psi_1, \ldots, \psi_n\}$$

Thus, finally, we define the symmetric difference of ground instances entailed by two sets of facts as follows:

$$\Gamma_1 \,\Delta\, \Gamma_2 := (\Gamma_1 \setminus \Gamma_2) \,\dot\cup\, (\Gamma_2 \setminus \Gamma_1)$$

6 Soundness and Completeness

In this section, we provide soundness and completeness results for the algorithms that we presented in the previous section. We begin by showing that computing the set of resolvable queries as the least fixpoint of the immediate consequence operator indeed computes all queries that are resolvable within the CLP. To establish this, we can straightforwardly reuse results established in prior work [18]. Afterwards, we provide soundness and completeness results for our change-impact analysis.

6.1 Resolvable Queries

In this section, we prove that $Q_{\mathsf{resolvable}}(s_0, k_0, c)$ actually computes all resolvable facts derivable from the CLP $\Gamma(s_0, k_0, c)$. We do this by relating the least fixpoint of the S_P operator to a logical interpretation of a CLP, following prior work [18].

For a CLP P, an atom a is a *logical immediate consequence* of a set of atoms A if there is a rule $r = h \leftarrow b_1, \ldots, b_n \in P$ and valuation θ such that $a = \theta(h)$ and both (a) for all atoms b_i occurring in r's body it holds that $\theta(b_i) \in A$, and (b) for the conjunction c of all constraint formulas occurring in r's body it holds that $D_{\mathfrak{C}} \models \theta(c)$. The logical immediate consequence function $T_P^{\mathfrak{C}}(I)$ maps I to all its logical immediate consequences for P.

One can relate a program's logical immediate consequences to a program's immediate consequences as defined by S_P. To do so, though, we must first interpret the constraints that remain when applying S_P, which we do with the operator $[\,]_{\mathfrak{C}}$. $[F]_{\mathfrak{C}}$ maps a fact F to its set of ground consequences in the constraint domain. Formally:

$$[h \leftarrow c_1, \ldots, c_n]_{\mathfrak{C}} = \{\sigma(h) \mid D_{\mathfrak{C}} \models_\sigma c_1 \wedge \cdots \wedge c_n\}.$$

We extend $[\,]_{\mathfrak{C}}$ to sets of facts as expected:

$$[S]_{\mathfrak{C}} = \bigcup_{F \in S} [F]_{\mathfrak{C}}.$$

It has been shown in [18] that the least fixpoint of S_P, when its constraints are interpreted, is equal to the least fixpoint of $T_P^{\mathfrak{C}}$, which interprets constraints directly in the constraint domain. The *least fixpoint* of a function f, denoted $\mathsf{lfp}(f)$, is the smallest set M such that $M = f(M)$.

Theorem 1 (Theorem 4.8 in [18]). *Let P be a CLP. Then:*

$$[\mathsf{lfp}(S_P)]_{\mathfrak{C}} = \mathsf{lfp}(T_P^{\mathfrak{C}}).$$

Theorem 1 applies directly to $Q_{\mathsf{resolvable}}$, which only restricts $\mathsf{lfp}(S_{\Gamma(s_0,k_0,c)})$ to facts with particular heads. It shows that the ground instances of the facts in $Q_{\mathsf{resolvable}}(s_0,k_0,c)$ are exactly the resolvable atoms in the logical least fixpoint of $\Gamma(s_0,k_0,c)$. Thus, our algorithm is sound ($\subseteq$) and complete ($\supseteq$).

6.2 Change-Impact Analysis

We show soundness and completeness of our change-impact analysis again by reference to $T_P^{\mathfrak{C}}$. Concretely, we show that the immediate consequences of $\phi_1 \setminus \phi_2$ correspond to the immediate consequences of ϕ_1 without the immediate consequences of ϕ_2. In the following, we assume that ϕ_1 and ϕ_2 do not share any variables, which we call *variable-disjoint*, and that all free variables occurring in the formula body occur in the formula head, which we call *constraint-closed*.

Theorem 2. *Let ϕ_1, ϕ_2 be two variable-disjoint and constraint-closed rules, whose bodies only contain constraint formulas. Then:*

$$\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) = \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}) \setminus \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2})$$

We prove Theorem 2 in Appendix A. The proof follows the construction of $\phi_1 \setminus \phi_2$ as per Algorithm 1.

We do not establish that the results of $Q_{\mathsf{resolvable}}$ will be both variable-disjoint and constraint-closed. However, we enforce both in our implementation. Variable-disjointness can be easily achieved by renaming facts before computing ($\setminus$), and constraint-closedness is achieved by the design of rules in $\Gamma(s_0,k_0,c)$. We check constraint-closedness nevertheless.

7 Implementation

We provide a complete implementation of both our formal model and reasoning, as described in Section 4 and Section 5. DecDNS is written in Haskell and provides multiple modes. It can (a) reason about the resolvability of zone files, (b) compute a change-impact analysis for two sets of zone files, (c) use our formal model to recursively resolve a DNS query with real DNS name servers, and (d) compile a set of zone files to a CLP that can be loaded into SWI-Prolog [32].

Our implementation shows that our formal model and reasoning are practical. Moreover, the mode for querying live DNS servers to resolve DNS queries shows that our formal model is so accurate and detailed that it can interoperate with real DNS servers.

DecDNS has three parts: (a) a compiler from DNS zone data to a CLP, (b) a reasoning component for computing a least fixpoint, performing resolution, and computing a change-impact analysis, and (c) a constraint solver supporting the constraints described in Section 4.3.2. The implementation of the compiler and the reasoning is straightforward. The constraint solver requires more attention, though, and we thus explain it in more detail below.

7.1 Constraint Solver Implementation

The constraint solver provides support for the inequality constraint ($\neq$) (also universally quantified), which we describe in detail in Appendix B, the constraint isName, and the DNSSEC-related constraints. DNSSEC-related constraints are only solved when all their arguments are ground. To avoid repeated, expensive cryptographic checks such as signature verification, our constraint solver caches the results of previously solved ground DNSSEC constraints. Checking DNSSEC-related constraints currently does not consider the records' validity periods.

The isName constraint checks the minimum length in bytes of a name potentially containing variables and evaluates to $\bot$ when a label exceeds the maximum length of 63 bytes or the name exceeds the maximum length of 255 bytes. It evaluates to $\top$ only for ground inputs of valid length. We assign a variable name X a minimum length of 1 as it may be the root name which is encoded as a zero byte. We assign a name represented by the list $[l_1, \ldots, l_n \mid \ldots]$ a minimum length of 1 (for the root label) plus the sum of the lengths of its elements $l_1, \ldots, l_n$, which represent labels. We assign no length to a variable list tail as it may be empty. Labels represented by atoms are of length 1 (for the length byte) plus the atom's length. Labels represented by variables have a minimum length of 2 (the length byte plus at least one character).

7.2 DNS Integration

DecDNS provides a mode in which it resolves a query using real DNS name servers, for which it only needs a list of root name server addresses, root apex NS records, and a set of trusted DS records. This mode uses the resolution rule as described in Section 3.2.2 and works as follows.

We compile an initial CLP from the root hints, which identify a set s_0 of root name servers, and the supplied trust anchors k_0. Then, we attempt to solve the DNS query for query type qt and query name qn by solving the CLP query $\mathsf{resolves}(\mathsf{name}(s_0), \mathsf{qt}, _, \mathsf{name}(\mathsf{qn}), D, _, \top, _)$, i.e., we check whether we can obtain some data D at the root name servers for the given query type and name while performing validation, but disregarding the particular zone, whether an alias was followed, and the final authentication status, thus allowing both authenticated answers and answers accepted as insecure.

During resolution in the CLP, we record all failed goals, i.e., those for which there is no solution. These failed goals encode potential follow-up queries. Concretely, for every

failed goal, we extract the goal's name server set, query name, and query type, which encodes a query to one of the name servers for that query name and type. Initially, the goal $\mathsf{resolves}(\mathsf{name}(s_0), \mathsf{qt}, _, \mathsf{name}(\mathsf{qn}), D, _, \top, _)$ will fail, which triggers the first query to a root name server.

Once we obtain all failed goals, we sort them according to a simple heuristic and send the respective queries to the respective name servers. For the example above, this would initially be a query to one of the root name servers, which would likely return referral data. We compile this new data to further CLP rules, add those to our CLP, and attempt again to solve the initial CLP query. We iterate until we find an answer to the query.

We used an existing library for DNS query encoding and parsing, but all recursive resolution logic is handled by the CLP and the query sorting heuristic.

7.3 SWI-Prolog Integration

SWI-Prolog is a popular Prolog interpreter [32], which also supports CLPs. We provide an integration for SWI-Prolog so that it can reason about CLPs compiled by DecDNS. SWI-Prolog already supports inequality constraints natively. For all other constraints, we provide a Haskell foreign-function interface, which allows Haskell code to be called from C code. For full SWI-Prolog integration, we provide a small C module that provides support for isName and DNSSEC-related constraints by linking their interpretation to our Haskell code.

8 Evaluation

In this section, we present our evaluation of DecDNS. First, we evaluate whether DecDNS's reasoning is faithful to the DNS standards, both when reasoning offline about zone files in Section 8.1 and when querying the live DNS infrastructure in Section 8.2. Second, we show which kinds of configuration errors DecDNS can detect (Section 8.3). We discuss limitations of DecDNS in this regard later in Section 10. Finally, we measure DecDNS's performance in Section 8.4.

8.1 Static Evaluation

We use the Ferret dataset [20] to evaluate whether our model of recursive resolution is faithful to the DNS standards. The Ferret dataset was created to test name server RFC compliance and contains zones and queries to those zones that exhibit corner cases of the DNS. It does not, however, contain signed zones. For each zone in the dataset, we analyze which queries are resolvable according to DecDNS and compare the results with those of BIND [17] and Knot [11].

Concretely, for a given zone and set of queries Q in the Ferret dataset, we test the following combinations and queries:

- We consider the zone once unsigned and once signed. The Ferret dataset also includes some invalid zones, which we only consider unsigned.
- We load the zone into BIND or Knot as the authoritative name server.
- We resolve each query in Q with BIND or Knot as the recursive resolver. Resolver queries are directly forwarded to the name server serving the respective zone.
- We also generate random queries and resolve them with either BIND or Knot as the recursive resolver.

It can occur that a name server rejects loading a zone because it considers it malformed. This can also occur for DecDNS whenever well-formedness checks fail. When a zone is loaded by both the respective name server and DecDNS, we check whether $Q_{\mathsf{resolvable}}$ contains a matching query if and only if the resolver returns a positive response, and we additionally compare DNSSEC validation results and response data. This comparison subsumes different reasons why a resolver may return no data, including an error such as SERVFAIL or NXDOMAIN and a successful response with no data. Overall, we evaluated 140,572 queries over 26,246 zones in four configurations (BIND/Knot name server $\times$ BIND/Knot resolver).

Results We find that DecDNS's output generally agrees with the real resolver software. We only find six types of discrepancies between DecDNS and the name servers or resolvers, which we accept as inconsistencies. During the evaluation, we found a number of errors in DecDNS, e.g., inconsistent serialization of record data for signature verification, incorrectly treating referral data as an answer, and incorrect handling of NS and DNAME records with the same owner name. We fixed these errors and iterated the evaluation until we decided that the remaining inconsistencies were acceptable.

Table 1 provides the details of the accepted inconsistencies. The largest source of inconsistencies is that the BIND name server rejects zones with missing name server addresses, but DecDNS does not. DecDNS assumes that the addresses of the name servers serving the common root of the input zones are available to the resolver. Only if a parent zone and delegation were considered would the lack of name server addresses become visible in DecDNS's output.

DecDNS rejects zones that contain non-referral data at or below a delegation, which Knot and BIND permit (see Appendix C). RFC 2181 [13, Section 6.1] only requires name servers to ignore such data at or below a zone cut. We choose to be stricter than the RFCs as we still argue that such data indicates a configuration error. DecDNS also does not synthesize CNAME records for queries that match DNAME records, but instead encodes their semantics directly in the model of recursive resolution.

Mismatch attribution	Queries	Zones
BIND name server rejects NS address configurations	2,700	38
DecDNS rejects data at or below delegation	–	1,556
DecDNS omits synthesized CNAME records	776	45
Resolver alias rewrite limits	259	7
Knot resolver loops on DS lookups	9	9
Knot name server omits wildcard NSEC proofs	3	3
Timeout	–	1

Table 1: Categories of observed differences. For query counts, categories may overlap.

In some cases, DecDNS reported queries as being resolvable but the resolver did not provide a positive response. Mostly, this was due to rewrite limits of the respective resolver. For nine queries, though, Knot returned an error after it entered a loop in an attempt to resolve DS follow-up queries.

We also identified what we believe to be a Knot name server bug. For a few queries, Knot omitted the relevant NSEC proofs for a signed response synthesized from a wildcard owner name, leading to a failure in resolution by BIND and Knot while DecDNS reported the queries as resolvable.

Finally, one zone configuration from the evaluation led to a timeout. This zone allows for the resolution of approximately 2^{40} queries due to DNAME rewrites, which cannot be represented efficiently with constraints.

8.2 Online Evaluation

We also evaluated DecDNS against real DNS infrastructure using a "100,000 queries" file, which the BIND [17] implementation provides and uses for A/B testing new releases. From this file, we randomly sampled 2,775 queries for record types supported by DecDNS. We resolved each query with both the `dig` utility and DecDNS in querying mode (see Section 7.2), configured with a 30 s timeout and a limit of 100 queries to DNS name servers.

We show our results in Table 2. DecDNS timed out for approximately 67% of the queries. For 21% of the queries, DecDNS matched the result from `dig`. The results differed for around 12% of the queries.

We attribute 20 of these mismatches to a name server returning different data to `dig` and DecDNS. This is common practice, e.g., for services using content-delivery networks. Both `dig` and DecDNS log all records used for resolution. After studying these logs, we concluded that both interpreted the data correctly but arrived at different results.

In all other cases, we attribute the mismatch to DecDNS raising a benign error when `dig` succeeds in resolving the query. In some cases, DecDNS received records that violated well-formedness invariants. For example, it crashed upon receiving `. NS localhost.`. In other cases, there was a network error in our testing environment affecting only DecDNS, or DecDNS exhausted its query limit.

Category	Reason	Count
Success	–	579
Timeout	–	1854
Mismatch	Different response	20
	Well-formedness errors	8
	Network error	290
	Query limit exhausted	24

Table 2: Results of the online evaluation.

We note, though, these results could be subject to a survivorship bias, as more complex queries are more likely to lead to a timeout but also more likely to exhibit diverging behavior between `dig` and DecDNS. Nevertheless, this evaluation shows that our approach provides some interoperability with real DNS infrastructure.

8.3 Detecting Configuration Errors

We evaluated whether DecDNS can detect configuration errors by introducing them into a number of example zones. Because DecDNS performs a change-impact analysis, it can detect configuration errors whenever they manifest as a change in resolvability. This, though, means that detecting configuration errors may require interpreting the results of the change-impact analysis.

Table 3 summarizes example configuration errors that we can detect. We can most easily detect errors introduced by changes that should preserve the resolvability of existing queries. This includes, for example, introducing a new zone cut or a DNSSEC key rollover, the kind of change that led to the DENIC outage [6]. For many other configuration errors, detection depends on whether the change introducing them deletes records. For example, when decommissioning name servers and moving the zone to new ones, the old name server addresses would be expected to become unresolvable, and thus there would be an expected loss of resolvability.

8.4 Performance

8.4.1 Static Evaluation

For the static evaluation as described in Section 8.1, we measured DecDNS's runtime per zone, including process startup and output collection, but excluding zone signing. We exclude configurations that fail well-formedness checks or time out. The average runtime is 38.0 ms for unsigned and 92.2 ms for signed zones, with medians of 31.0 ms and 76.0 ms, respectively. The 95th-percentile runtimes for original and signed

Configuration error	Error description	Interpretation?
Cyclic zone dependencies	Name server address resolution forms a cycle.	•
Delegation inconsistency	The parent's delegation and the child's authoritative name server information disagree.	✓
DENIC outage	During a rollover, multiple signing keys are used but only a subset is published as DNSKEY records, as in DENIC's May 2026 outage [6].	✗
Incomplete zone cut	A new zone cut is introduced but some records are omitted.	✗
Lame delegation	A parent delegates a child zone to a name server that does not serve the zone.	•
Missing glue	A parent delegates a zone to a name server but does not include required glue records.	•
Rewrite blackholing	Alias rewriting terminates with a non-existent name.	•
Rewrite loop	Alias rewriting forms a cycle.	•
Wildcard shadowing	A newly added name blocks wildcard synthesis.	✗

Table 3: Configuration errors that a resolvability regression analysis can detect. "Interpretation?" marks whether detecting the configuration error requires interpretation of the output. This is not the case when a change introducing that error is generally not expected to change the resolvability of data queries. "•" indicates that interpretation may only sometimes be required.

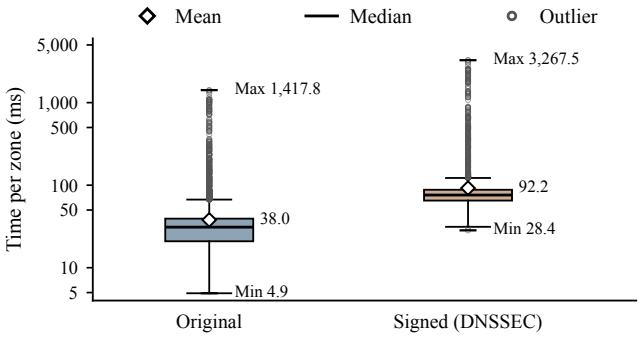


Figure 2: Runtime distribution per valid zone on a logarithmic scale. Boxes span the 25th–75th percentiles, and whiskers extend to minima and maxima excluding outliers. Outliers are shown as dots.

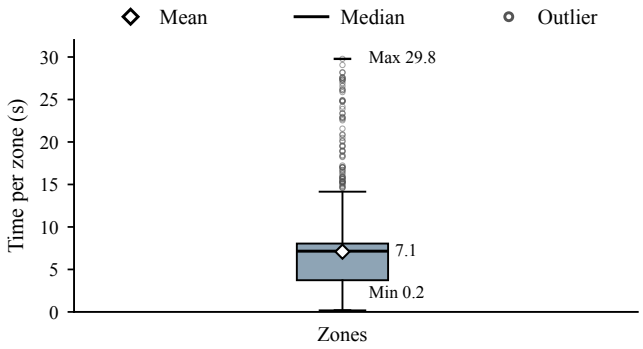


Figure 3: Runtime distribution for successful online queries, in seconds. Boxes span the 25th–75th percentiles, and whiskers extend to minima and maxima excluding outliers. Outliers are shown as dots.

configurations are 64.8 ms and 107.8 ms, respectively. To investigate outliers, we inspected the ten slowest signed and unsigned configurations, all of which contained DNAME loops, producing hundreds of derived facts from only four or five records. Note that the Ferret dataset is relatively homogeneous in structure, and that no zone configuration contains more than five records.

8.4.2 Online Evaluation

We also timed resolution by DecDNS during the online evaluation described in Section 8.2. Figure 3 shows the distribution of runtimes for queries that did not time out and did not raise an error. The mean and median runtimes were both 7.1 s.

9 Related Work

Many works tackled the issue of establishing properties of DNS zone configurations [12, 19, 23, 26, 31]. All these works, though, lack support for DNSSEC and suggest verifying the absence of particular types of configuration errors. Many of these works provide support for verifying other properties, so it would theoretically be possible to verify different, more constructive properties, but no such properties are suggested. In contrast, DecDNS performs a change-impact analysis, which allows for detecting configuration errors in a much more general way, and provides support for DNSSEC.

There is also prior work that suggested the use of logic programming for reasoning about the DNS [16]. Their model, however, lacks many features that we cover. For example, they

do not cover DNSSEC, the following of referrals, obtaining name server addresses, and CNAME records with a wildcard owner name. The authors also leave it open what their model could be used for and handle wildcard synthesis and DNAME rewrites explicitly rather than symbolically as we do.

There are other works that studied DNSSEC formally [9, 33], but these works studied the security of the DNSSEC standards rather than the correctness of zones. For example, both works analyzed whether an attacker can trick a resolver into accepting attacker-supplied data as valid. Thus, these models are to some extent incomparable to ours. Both works consider only a fixed example zone for their security analyses and do not consider aliases, and both analyses include a model of recursive resolvers. The latter difference most clearly contrasts these works to ours. We do not model recursive resolvers since our model *is* the resolver.

Finally, a related but orthogonal work is "Topaz" [21]. Topaz is a formally verified name server architecture that correctly serves data associated with domain names, and was designed for use within content-delivery networks. Such networks often serve different IP addresses to clients, e.g., optimizing the server location with respect to the client location. In Topaz, such optimizations can be expressed in a verified policy language which administrators can use to configure which DNS data should be served to which clients.

10 Discussion & Conclusions

In this paper, we presented a new formal model for reasoning about recursive resolution in the DNS, as well as DecDNS, an implementation of that model. DecDNS was designed to perform a change-impact analysis of zone configuration updates, which can help detect zone configuration errors. Moreover, our formal model is so accurate and detailed that DecDNS can resolve DNS queries with the real DNS infrastructure.

Limitations Nevertheless, our work is not without limitations. First, DecDNS's change-impact analysis does not detect all potential configuration errors. (a) DecDNS disregards the correctness of the data associated with names and thus cannot detect whether, e.g., the IP address associated with a name server is incorrect. (b) DecDNS cannot detect when name servers serve inconsistent data. We assume that all name servers configured at a zone's apex indeed serve the respective zone. (c) DecDNS does not detect when the name servers serving a configuration's topmost zone have no addresses configured. We assume resolvers are configured with those addresses. (d) DecDNS reasons about whether a resolution path exists, which means that we may not detect that a particular resolution path, potentially taken by many resolvers, does not return a response.

Second, our model and algorithm have some practical limitations. (a) It could occur that our change-impact analysis returns formulas containing contradictory residual constraints that our constraint solver fails to recognize as such. We, however, never encountered such a case. (b) DecDNS's performance can be further improved; for example, it timed out when we attempted to reason about the root zone. (c) Our model currently omits some aspects of NSEC and NSEC3 records, particularly when validating responses synthesized from signed wildcard records.

Third and finally, our evaluation does not cover DNSSEC and multi-zone corner cases well. The Ferret dataset (see Section 8.1) includes only corner cases for individual, unsigned zones, and there is no characterization as to which configurations are covered by our online evaluation (Section 8.2).

Future Work We plan to address many of the above limitations in future work. Concretely, we plan to improve performance, broaden the evaluation, and fully model NSEC and NSEC3. Moreover, we believe that the change-impact analysis computed by DecDNS can also be generalized to not only consider old and new zone configurations but also different compilations of zone configurations. For example, the same zone could be compiled for different versions of the DNS standards or the compilation could model that all resolution paths must succeed instead of modeling the existence of a resolution path.

DecDNS's change-impact analysis currently requires complete knowledge of a zone and all its subzones. One could investigate treating unresolved resolves goals for unknown zones as residual constraints, which could allow reasoning with partial knowledge.

Finally, it would be worthwhile to analyze DecDNS's security, particularly in querying mode. Such an analysis would strengthen DecDNS's value as a potential reference implementation.

Acknowledgments

We thank Steve Crocker, Petr Špaček, and Peter Thomassen for their interest in this project and their valuable feedback.

References

[1] The AFNIC Scientific Council shares its report on DNS-based Internet filtering. URL: https://www.afnic.fr/en/observatory-and-resources/news/the-afnic-scientific-council-shares-its-report-on-dns-based-internet-filtering/.

[2] DNSViz. URL: https://github.com/dnsviz/dnsviz.

[3] RFCs about DNS | RFC Editor. URL: https://www.rfc-editor.org/subjects/dns/.

[4] SAC056 | Executive Summary for SSAC Advisory on Impacts of Content Blocking via the Domain Name System. URL: https://www.icann.org/en/ssac/publications/documents/sac056-executive-summary-for-ssac-advisory-on-impacts-of-content-blocking-via-the-domain-name-system-09-10-2012-en.

[5] The Zonemaster Project. URL: https://github.com/zonemaster/zonemaster.

[6] Final Report: DNS Outage of 5 May 2026, June 2026. URL: https://blog.denic.de/en/final-report-dns-outage-of-5-may-2026/.

[7] When DNSSEC goes wrong: How we responded to the .de TLD outage, May 2026. URL: https://blog.cloudflare.com/de-tld-outage-dnssec/.

[8] Roy Arends, Geoffrey Sisson, David Blacka, and Ben Laurie. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. Request for Comments RFC 5155, Internet Engineering Task Force, March 2008. doi:10.17487/RFC5155.

[9] Jason Bau and John C. Mitchell. A Security Evaluation of DNSSEC with NSEC3. In *Network and Distributed System Security (NDSS)*, 2010. URL: https://www.ndss-symposium.org/ndss2010/security-evaluation-dnssec-nsec3/.

[10] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), March 1989. doi:10.1109/69.43410.

[11] CZ.NIC. Knot Resolver. Accessed: 2026-10-31. URL: https://www.knot-resolver.cz/.

[12] Haizhou Du, Kaiqiang Hu, Yao Wang, and Zhihao Guo. VeriDNS: Incremental distributed verification of DNS configurations. *Computer Networks*, 275, February 2026. doi:10.1016/j.comnet.2025.111929.

[13] Robert Elz and Randy Bush. Clarifications to the DNS Specification. Request for Comments RFC 2181, Internet Engineering Task Force, July 1997. doi:10.17487/RFC2181.

[14] Oliver Farnan, Alexander Darer, and Joss Wright. Poisoning the Well: Exploring the Great Firewall's Poisoned DNS Responses. In *Workshop on Privacy in the Electronic Society*, October 2016. doi:10.1145/2994620.2994636.

[15] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. How Great is the Great Firewall? Measuring China's DNS Censorship. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/hoang.

[16] Kaiqiang Hu, Haizhou Du, Kexin Yu, and Xiaoliang Wang. SemDNS: A Declarative Semantics for DNS Resolution. In *Workshop on Formal Methods Aided Network Operation (FMANO '26)*, August 2026. doi:10.1145/3789240.3822358.

[17] Internet Systems Consortium. BIND 9. Accessed: 2026-10-31. URL: https://www.isc.org/bind/.

[18] Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The semantics of constraint logic programs. *The Journal of Logic Programming*, 37(1), October 1998. doi:10.1016/S0743-1066(98)10002-X.

[19] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. GRooT: Proactive Verification of DNS Configurations. In *SIGCOMM*, July 2020. doi:10.1145/3387514.3405871.

[20] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/kakarla.

[21] James Larisch, Timothy Alberdingk Thijm, Suleman Ahmad, Peter Wu, Tom Arnfeld, and Marwan Fayed. Topaz: Declarative and Verifiable Authoritative DNS at CDN-Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, August 2024. doi:10.1145/3651890.3672240.

[22] Edward P. Lewis. The Role of Wildcards in the Domain Name System. Request for Comments RFC 4592, Internet Engineering Task Force, July 2006. doi:10.17487/RFC4592.

[23] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A Formal Framework for End-to-End DNS Resolution. In *SIGCOMM*, September 2023. doi:10.1145/3603269.3604870.

[24] J. W. Lloyd. *Foundations of Logic Programming*. 1984.

[25] P. Mockapetris. Domain names - concepts and facilities. Request for Comments RFC 1034, Internet Engineering Task Force, November 1987. doi:10.17487/RFC1034.

[26] Dhruv Nevatia, Si Liu, and David Basin. Reachability Analysis of the Domain Name System. *Proc. ACM*

Program. Lang., 9(POPL), January 2025. doi:10.1145/3704898.

[27] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM (JACM)*, January 1965. doi:10.1145/321250.321253.

[28] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. Protocol Modifications for the DNS Security Extensions. Request for Comments RFC 4035, Internet Engineering Task Force, March 2005. doi:10.17487/RFC4035.

[29] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. Resource Records for the DNS Security Extensions. Request for Comments RFC 4034, Internet Engineering Task Force, March 2005. doi:10.17487/RFC4034.

[30] Scott Rose and Wouter Wijngaards. DNAME Redirection in the DNS. Request for Comments RFC 6672, Internet Engineering Task Force, June 2012. doi:10.17487/RFC6672.

[31] Yao Wang, Kexin Yu, Ziyi Wang, Kaiqiang Hu, Haizhou Du, Qiao Xiang, Xing Fang, Geng Li, Ruiting Zhou, Linghe Kong, and Jiwu Shu. Rethinking DNS Configuration Verification with a Distributed Architecture. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, August 2024. doi:10.1145/3663408.3663412.

[32] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2), January 2012. doi:10.1017/S1471068411000494.

[33] Qifan Zhang, Zilin Shen, Imtiaz Karim, Elisa Bertino, and Zhou Li. Proving DNSSEC Correctness: A Formal Approach to Secure Domain Name Resolution, December 2025. arXiv:2512.11431, doi:10.48550/arXiv.2512.11431.

A Proofs for Change-Impact Analysis

We begin our proof of Theorem 2 by proving a lemma that allows us to rewrite our proof goal in simpler terms.

Lemma 1. *Let F be a set of rules only containing constraint formulas in their body. Then* $\mathsf{lfp}(T_F^{\mathfrak{C}}) = T_F^{\mathfrak{C}}(\emptyset)$.

Proof. Let F be a set of rules only containing constraint formulas in their body. Observe that for such rules the definition of $T_P^{\mathfrak{C}}$ is independent of its argument, that is:

$$\forall x, y. T_F^{\mathfrak{C}}(x) = T_F^{\mathfrak{C}}(y)$$

Thus, we obtain $\mathsf{lfp}(T_F^{\mathfrak{C}}) = T_F^{\mathfrak{C}}(\emptyset)$. □

We proceed to introduce two lemmas that relate $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2})$ to $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1})$ and $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2})$, which will help prove that:

$$\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) \subseteq \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}) \setminus \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2}).$$

Lemma 2. *Let ϕ_1, ϕ_2 be two variable-disjoint facts. Then:*

$$\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) \subseteq \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}).$$

Proof. Observe that in every case of Algorithm 1, we only strengthen ϕ_1, i.e., either add more conjuncts to its body or additionally apply a substitution to its head. This directly proves Lemma 2. □

Lemma 3. *Let ϕ_1, ϕ_2 be two variable-disjoint and constraint-closed facts. Then:*

$$\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) \cap \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2}) = \emptyset.$$

We require that ϕ_1 and ϕ_2 are constraint-closed because Algorithm 1 might otherwise return incorrect results for cases corresponding to line 6. To realize that, consider the two facts $\phi_1 = p(X)$ and $\phi_2 = p(f(Y)) \leftarrow c(Z)$ for some constraint $c/1$. Observe that ϕ_2 is not constraint-closed. For those formulas, we obtain:

$$\phi_1 \setminus \phi_2 = \{p(X) \leftarrow \forall Y. p(X) \neq p(f(Y)),\ \psi\},$$
$$\psi = p(f(Y)) \leftarrow \neg c(Z).$$

As the heads of ϕ_2 and ψ, which is obtained from Algorithm 1 line 6, are independent of the constraint $c(Z)$ or $\neg c(Z)$ respectively, we can derive $p(f(Y))$ for any ground instance of Y from both $\phi_1 \setminus \phi_2$ and ϕ_2 provided that c is neither tautological nor contradictory. Clearly, this is not intended for $\phi_1 \setminus \phi_2$. We thus limit the application of Algorithm 1 to constraint-closed facts, which are the norm, and for which this problem does not arise, as we show below.

Proof. Let ϕ_1, ϕ_2 be two variable-disjoint and constraint-closed facts such that

$$\phi_1 = h_1 \leftarrow \Psi_1 \quad \phi_2 = h_2 \leftarrow \Psi_2$$

Let $a \in \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2})$ and let σ be a valuation such that $a = \sigma(h_2)$ and $D_{\mathfrak{C}} \models \sigma(\Psi_2)$. Presume, for a contradiction, that $a \in \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2})$. Since $a \in \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2})$, we have that $a \in T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$ by Lemma 1, and thus, there must be a rule $\psi = h \leftarrow \Psi$ in $\phi_1 \setminus \phi_2$ such that $a \in T^{\mathfrak{C}}_{\psi}(\emptyset)$, i.e., there must be a valuation σ' such that $a = \sigma'(h)$ and $D_{\mathfrak{C}} \models \sigma'(\Psi)$. We proceed with a case distinction.

- Let ψ have the following shape, i.e., come from Algorithm 1 line 5:

$$h_1 \leftarrow \{\forall \mathsf{vars}(h_2). h_1 \neq h_2\} \cup \Psi_1.$$

Observe that:

$$\sigma'(\forall \mathsf{vars}(h_2).h_1 \neq h_2) = \forall \mathsf{vars}(h_2).a \neq h_2.$$

However, we have that $D_{\mathfrak{C}} \not\models \forall \mathsf{vars}(h_2).a \neq h_2$ as $a = \sigma(h_2)$. Thus, ψ cannot be of the above shape.

- Let ψ have the following shape for $\theta = \mathsf{MGU}(h_1, h_2)$ and $c \in \Psi_2$, i.e., come from Algorithm 1 line 6:

$$\theta(h_1) \leftarrow \theta(\Psi_1) \cup \{\theta(\neg c)\}$$

Observe that by assumption we have:

$$D_{\mathfrak{C}} \models \sigma'(\Psi) = \sigma'(\theta(\Psi_1) \cup \{\theta(\neg c)\}).$$

Thus and in particular, we have that $D_{\mathfrak{C}} \models \sigma'(\theta(\neg c))$ or equivalently $D_{\mathfrak{C}} \not\models \sigma'(\theta(c))$.

Observe that since $\theta(h_1) = \theta(h_2)$, we have that $\mathsf{vars}(\theta(h_1)) = \mathsf{vars}(\theta(h_2))$. Consequently, we have that ψ is constraint-closed, as ϕ_2 is constraint-closed and thus $\mathsf{vars}(\theta(c)) \subseteq \mathsf{vars}(\theta(h_2))$.

As $a = \sigma(h_2)$ and $a = \sigma'(\theta(h_1))$, and because both ϕ_2 and ψ are constraint-closed, we can deduce that $\sigma(c) = \sigma'(\theta(c))$. This means, though, that all of the following statements are true, which is a contradiction:

$$D_{\mathfrak{C}} \models \sigma(c), \quad D_{\mathfrak{C}} \not\models \sigma'(\theta(c)), \quad \sigma(c) = \sigma'(\theta(c)).$$

In consequence of the above case distinction, we can deduce that $a \notin T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$, which contradicts our initial assumption. □

Before we proceed to our proof of Theorem 2, we prove a helpful property about most general unifiers and their relationship to valuations.

Lemma 4. *Let a_1 and a_2 be two variable-disjoint atoms and $\theta = \mathsf{MGU}(a_1, a_2)$. If there exist valuations σ_1, σ_2 such that $\sigma_1(a_1) = \sigma_2(a_2)$, then there also exists a valuation σ such that $\sigma_1(a_1) = \theta\sigma(a_1) = \theta\sigma(a_2) = \sigma_2(a_2)$.*

Proof. Let $a_1, a_2, \sigma_1, \sigma_2$ and θ as per Lemma 4. Let θ' be a valuation such that:

$$\theta'|_{\mathsf{vars}(a_1) \cup \mathsf{vars}(a_2)} = \sigma_1|_{\mathsf{vars}(a_1)} \cup \sigma_2|_{\mathsf{vars}(a_2)}.$$

Observe that θ' is a unifier for a_1 and a_2. Thus, there must exist a valuation σ such that $\theta' = \theta\sigma$, as θ is a most general unifier. □

Proof of Theorem 2. Let ϕ_1, ϕ_2 be two variable-disjoint and constraint-closed facts such that:

$$\phi_1 = h_1 \leftarrow \Psi_1, \quad \phi_2 = h_2 \leftarrow \Psi_2.$$

Presume that the heads of ϕ_1 and ϕ_2 do not unify. Then:

$$\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) = \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}) = \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}) \setminus \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2}).$$

The first equality follows from Algorithm 1, line 3. The second follows because the heads of ϕ_1 and ϕ_2 do not unify. As they do not unify, no atom in $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2})$ can be contained in $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1})$.

Now presume that the heads of ϕ_1 and ϕ_2 do unify and let $\theta = \mathsf{MGU}(h_1, h_2)$. We proceed to prove Theorem 2 by contradiction. Suppose that $\mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}) \neq \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_1}) \setminus \mathsf{lfp}(T^{\mathfrak{C}}_{\phi_2})$. By Lemma 1, we can rewrite this inequality to:

$$T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset) \neq T^{\mathfrak{C}}_{\phi_1}(\emptyset) \setminus T^{\mathfrak{C}}_{\phi_2}(\emptyset).$$

Thus, there must exist a ground atom a such that:

$$a \in T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset) \iff a \notin T^{\mathfrak{C}}_{\phi_1}(\emptyset) \setminus T^{\mathfrak{C}}_{\phi_2}(\emptyset).$$

We proceed by case distinction.

- Presume that $a \in T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$. Then, by the above statement, we have that either $a \notin T^{\mathfrak{C}}_{\phi_1}(\emptyset)$ or $a \in T^{\mathfrak{C}}_{\phi_2}(\emptyset)$.
  - Presume that $a \notin T^{\mathfrak{C}}_{\phi_1}(\emptyset)$. This is impossible by Lemma 2. Contradiction.
  - Presume that $a \in T^{\mathfrak{C}}_{\phi_2}(\emptyset)$. This is impossible by Lemma 3. Contradiction.
- Now presume that $a \in T^{\mathfrak{C}}_{\phi_1}(\emptyset) \setminus T^{\mathfrak{C}}_{\phi_2}(\emptyset)$. Then, symmetrically to the case before, we must have that $a \notin T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$.
  - Presume that a and h_2 unify. Then there must exist a valuation σ such that $a = \theta\sigma(h_1) = \theta\sigma(h_2)$ by Lemma 4.

    Since $a \notin T^{\mathfrak{C}}_{\phi_2}(\emptyset)$, there must be a constraint $c \in \Psi_2$ such that $D_{\mathfrak{C}} \not\models \theta\sigma(c)$ or equivalently $D_{\mathfrak{C}} \models \theta\sigma(\neg c)$. Also observe that since $a \in T^{\mathfrak{C}}_{\phi_1}(\emptyset)$, we have $D_{\mathfrak{C}} \models \theta\sigma(\Psi_1)$. Thus:

$$D_{\mathfrak{C}} \models \theta\sigma(\Psi_1 \cup \{\neg c\}).$$

    Then, for $\psi = \theta(h_1) \leftarrow \theta(\Psi_1 \cup \{\neg c\})$, we have that $a \in T^{\mathfrak{C}}_{\psi}(\emptyset)$ under the valuation σ. The formula ψ, though, is in $\phi_1 \setminus \phi_2$ (see Algorithm 1 line 6), and thus $a \in T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$. Contradiction.
  - Now suppose that a and h_2 do not unify. Let σ be a valuation such that $a = \sigma(h_1)$ and $D_{\mathfrak{C}} \models \sigma(\Psi_1)$. As a and h_2 do not unify, observe that for all valuations σ' we have $a \neq \sigma'(h_2)$. Hence $D_{\mathfrak{C}} \models \forall \mathsf{vars}(h_2).a \neq h_2$. Thus, σ is a valuation modeling $\psi = h_1 \leftarrow \{\forall \mathsf{vars}(h_2).h_1 \neq h_2\} \cup \Psi_1$. The formula ψ, though, is included in $\phi_1 \setminus \phi_2$ (see Algorithm 1 line 5), and therefore $a \in T^{\mathfrak{C}}_{\phi_1 \setminus \phi_2}(\emptyset)$. Contradiction.

□

B Simplifying Inequalities

Our constraint solver implementation simplifies inequality constraints as follows. It discards inequalities that are true, and it reduces inequalities on two of the same functors to a disjunction of inequalities between corresponding arguments. For example, it would reduce $f(a,X,Y) \neq f(a,Z,Z)$ to the disjunction $X \neq Z \vee Y \neq Z$ as when either inequality holds, the original inequality holds as well. Our constraint solver thereby maintains the invariant that inequalities always have a variable on their left-hand side.

Our constraint solver also supports simplifying universally quantified inequality constraints, which result from our change-impact analysis (see Section 6.2). Observe that Algorithm 1 only emits universally quantified constraints of the shape $\forall \mathsf{vars}(t_r).t_l \neq t_r$, i.e., all variables on the inequality's left-hand side are free, and all variables on the inequality's right-hand side are quantified. Our solver simplifies these constraints in four cases and the constraint remains unchanged in any other case:

- If t_l is ground, the constraint is solved by unification.
- If t_r is a variable, the constraint evaluates to $\bot$.
- If t_l and t_r do not unify, the constraint evaluates to $\top$.
- If t_l and t_r use the same function as the top-most symbol, we simplify their inequality to a disjunction of inequalities of their arguments. We then distribute the universal quantifier over this disjunction as much as possible. Afterwards, we recursively simplify each disjunct. For example, our solver simplifies the following inequality as follows:

$$\forall X,Y.f(a,b,A,B) \neq f(a,X,Y,Y)$$
$$\text{becomes } a \neq a \vee \forall X.b \neq X \vee \forall Y.(A \neq Y \vee B \neq Y)$$
$$\text{becomes } \forall Y.(A \neq Y \vee B \neq Y).$$

C Well-Formedness Checks

DecDNS checks that zone files are well-formed before compiling them. Note that we could enforce stricter well-formedness requirements. For example, RFC 5155, Sec. 7.2, [8] requires that NSEC3 records in a given response must share all relevant parameters. We, though, decided to only enforce invariants that are required for the correctness of our formal model.

- A zone must contain exactly one SOA record, which must occur at the apex.
- A zone must contain at least one NS at the zone apex.
- A zone must not contain DS records at the apex.
- If an owner name has a CNAME record, only CNAME, NSEC, and RRSIG records may occur at that owner name [25, Sec. 3.6.2].
- If a non-apex owner name has an NS record, only NS, DS, NSEC, RRSIG, A, and AAAA records may occur at that owner name, and only A and AAAA records may occur below it.
- No owner name may have more than one CNAME, DNAME, or NSEC record.
- If an owner name has a DNAME record, no records may occur below it [30, Sec. 2.4].
- DNAME, DS, and NS records cannot have a wildcard owner name [22].
- An RRSIG record's signer name must be the owner name or one of its ancestors. If the signature covers an SOA record, the signer's name must be equal to the owner name. If the signature covers a DS record, the signer's name must be a strict ancestor.
- An RRSIG record's label count must be correct with respect to [29, Sec. 3.1.3].
- DNSKEY records may only occur at the apex, except for non-DNSSEC keys.

DecDNS also generates warnings when records in an RRset have different TTLs (excluding RRSIG records from this check), or when an RRSIG record has a different TTL from any record in the RRset it covers.