How Complex is DNS?

Siva Kesava Reddy Kakarla
UCLA

Ryan Beckett
Microsoft

Todd Millstein
UCLA & Intentionet

George Varghese
UCLA

ABSTRACT

Motivated by recent results that show that Internet protocols can be surprisingly complex and, in particular, that BGP is Turing complete, we ask the same question for the Domain Name System (DNS). DNS is at least as pervasive and essential as BGP in the global Internet infrastructure. Besides the scientific interest, the complexity of DNS can have implications for new applications (that can utilize the unsuspected power of DNS), and for verification (to understand basic complexity limits and suggest new verification algorithms). In this paper, we show that using the power of DNAME record type, DNS can express regular languages and pushdown systems. The first result can be used to build a system for controlling domain access (of which parental control is a special case). The second result shows that verification of DNS zone files is likely to take time that is at least cubic in the number of records.

CCS CONCEPTS

• **Theory of computation** → **Grammars and context-free languages**; **Regular languages**; • **Networks** → **Application layer protocols**;

KEYWORDS

DNS, automata theory, pushdown systems, verification complexity, computational complexity

ACM Reference Format:
Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2021. How Complex is DNS?. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484266.3487369

1 INTRODUCTION

Network protocols often have surprising and accidental complexity. Such complexity can arise when practitioners design new protocol features or extensions targeted towards realistic use cases, without considering the collective impact such extensions can have on the system through their mutual interactions. Perhaps the best example of accidental complexity is in the Border Gateway Protocol (BGP) [24]. Designed to enable routing over the Internet among organizations with different, often conflicting policies, BGP was created to support an extremely rich set of policies. It took many years for theoreticians to "catch up" to practice, demonstrating that the seemingly simple policy mechanisms in BGP can be used to simulate an arbitrary Turing machine [8].

The theoretical complexity of a networked system is important because it has broad ramifications related to the ease with which humans and machines can analyze the system. For instance, even for finite network topologies, simply determining if BGP will converge is NP-Complete [13].

In this paper, we seek to analyze and understand the theoretical complexity of the Domain Name System (DNS) [22, 23], one of the oldest and most widely used distributed networking protocols on the Internet. DNS provides the "glue" that holds the Internet together, by translating user-recognizable domain names (*e.g.,* hotcrp.com) to machine-recognizable IP addresses, text data, mail records, and more [11, 30]. Arguably, DNS is as crucial and widely deployed as BGP, and understanding its complexity has implications on the cost of verification. Further, unlike BGP, DNS's power can be directly used by applications [2, 7].

As part of our investigation, we find that the DNS has surprising complexity. We first show that DNAME rewriting [27], a seemingly simple record type for domain redirection, allows the DNS to recognize arbitrary regular languages encoded in the string labels of the DNS query. Hence users can perform complex validation and lookup logic (e.g., string validation, domain filtering, parental controls, etc.) in the DNS itself as part of the configurable records that are processed at authoritative nameservers. Second, we demonstrate that the expressiveness of the DNS is beyond that of regular languages. Specifically, the combination of DNAME records and nondeterminism due to *nameserver delegation* allows the DNS to encode both deterministic and nondeterministic pushdown systems (PDS) [1, 3, 26] and hence to generate strings of arbitrary context-free grammars (e.g., strings of the form $a^n b^n$). As a consequence of these results, verification for arbitrary DNS zone files is likely to have a time complexity that is at least cubic in the number of DNS records.

2 BACKGROUND

The domain namespace is a tree-like hierarchy, starting with the root node, an empty label under which nodes like com and edu exist. These child nodes can have child nodes recursively. Nodes at any depth of the hierarchy can contain data, which users can request by querying the domain name formed by concatenating the labels from that node to the root. Data is stored as DNS *records* where each record has a domain (owner) name, a type, contents, and other attributes.

To scale to the worldwide Internet, the DNS namespace is divided into smaller manageable portions called *zones*. A zone starts at a domain and extends downward in the tree to the leaf nodes or to the top-level of subdomains where other zones start. Therefore, a zone is a collection of records that share a common end domain name. For example, the hotcrp.com zone has only records ending with hotcrp.com.

A distributed collection of organizations manage the zones and provide the translation service through publicly accessible DNS servers, called *nameservers*. Each nameserver serves one or more zones. Multiple servers also serve the same zone to ensure redundancy and availability. Each nameserver can provide the data requested for a domain name directly or point to other nameservers. A *resolver* is the client-side software that goes back and forth among different nameservers to fetch the data requested by the client.

DNS supports many record types, including records for IP addresses, text records, domain aliases, delegation records, and more. Table 1 shows a few example records. When a query (a pair of domain name and a type) arrives at a nameserver, it first checks the available zones to select the best matching zone and then uses the best matching records from that zone to answer the query. If the selected best records are of type A or AAAA or TXT, then the resolver gets the intended response. If the nameserver responds with an NS record then the resolver must contact another nameserver. If the best records are of type CNAME or DNAME then the original query is rewritten.

Consider the CNAME and DNAME records in Table 1. For the CNAME to apply, the input query domain name has to be the same as the CNAME record name (c.uni.edu.), and the query name is completely replaced by the content of the record (w.uni.edu.). For a DNAME to apply on the other hand, the query domain name only has to be a subdomain (for example, x.y.b.uni.edu.) of the name in the DNAME record (b.uni.edu.). The new query will preserve the subdomain and replace the suffix that matches the DNAME record name (producing the new query name x.y.cs.edu.). The DNAME record's ability to rewrite a suffix of the query, regardless of what comes before it, turns out to be surprisingly powerful.

Example Record			Description
`a.uni.edu.`	`A`	`1.1.1.1`	IPv4 record
`*.uni.edu.`	`TXT`	"Awesome"	Wildcard Text record
`s.uni.edu.`	`NS`	`ns.dns.com.`	Delegation record
`b.uni.edu.`	`DNAME`	`cs.edu.`	Domain redirection
`c.uni.edu.`	`CNAME`	`w.uni.edu.`	Canonical name

Table 1: Examples of common DNS record types.

3 DETERMINISTIC FINITE AUTOMATA

We first define a deterministic finite automaton (DFA) and then show how a DFA can be encoded in DNS. We then give an example and conclude with potential applications.

A DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the input string [15]. DFAs recognize exactly the set of regular languages – languages that use regular expressions [15]. Formally, a deterministic finite automaton $\mathcal{M}$ is a quintuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ such that Q is a finite set of states, Σ is a finite set of input symbols called the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is an initial or start state, and $F \subseteq Q$ is a set of final or accept states. The *language* of $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$, is the set of strings whose processing by $\mathcal{M}$ ends in a final state.

While a DFA is a mathematical concept, it is often implemented in hardware and software for solving specific problems such as lexical analysis in compilers and pattern matching. For example, a DFA can model software that decides whether or not online user input such as email addresses are syntactically valid.

3.1 Encoding an arbitrary DFA in DNS

Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be any DFA. Let $Q = \{q_0, q_1, \cdots, q_n\}$, $\Sigma = \{a_0, \cdots, a_m\}$. We show that $\mathcal{M}$ can be encoded in DNS using a single zone. Let the zone file be for the domain dfa.com.. Intuitively, we use the DNS query to encode both the remaining input string and the current state. We then use DNAME records to encode the transition relation and use TXT records to encode the final accept/reject status.

The steps to encode a DFA $\mathcal{M}$ as a zone file z are:

- **Start:** For each symbol a_i in the alphabet, add a DNAME record of the form "$\texttt{a}_\texttt{i}$ DNAME $\texttt{a}_\texttt{i}.\texttt{q}_0$", where q_0 is the start state.[1] These records add the start state to the beginning of the query without consuming any input.

[1]For exposition purposes we use *relative* domains here, which lack the trailing ".": implicitly the zone domain dfa.com. is appended to form the complete domain.

- **Transition:** For each transition of the form $q_i \times a_j \rightarrow q_k$ in δ, add a DNAME record "`ai.qj DNAME qk`". These records consume an input symbol a_i in a state q_j and move to the state q_k. By the DNS semantics, these DNAME records only apply to a query if it is a strict subdomain of `ai.qj`, so these records have the effect of transitioning the system from the start state to the penultimate state, with one input symbol remaining.
- **Decision:** For each transition of the form $q_i \times a_j \rightarrow q_k$, add a TXT record - "`aj.qi TXT "accept"`" if $q_k \in F$; otherwise add "`aj.qi TXT "reject"`". These records are the final step in the transition system, where a_j is the last input symbol and the system is in state q_i.

To test whether a given string $S \in \Sigma^*$ is accepted by the DFA $\mathcal{M}$, we encode $S = s_0 s_1 \cdots s_n$ as the domain name `sn.···.s1.s0.dfa.com.`. This query is then sent by the resolver to the nameserver that contains the zone file z. The text record response will contain "`accept`" if and only if the string $S \in \mathcal{L}(\mathcal{M})$.

3.2 Example

In this subsection, we show an example DFA and its encoding in DNS using the three steps mentioned above. Consider the DFA $\mathcal{M}_\mathcal{O}$ over alphabet $\{a, b\}$ shown in Figure 1, which accepts all strings that contain an odd number of a's.

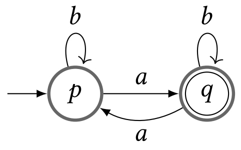

Figure 1: An example DFA $\mathcal{M}_\mathcal{O}$ that accepts strings only if they contain an odd number of a's

Table 2 shows the encoding of DFA $\mathcal{M}_\mathcal{O}$ shown in Figure 1 in DNS as a zone file z. To make it a valid zone, there must also exist an SOA and NS record for `dfa.com.`, which are omitted for brevity. To test whether the string $abaa$ is accepted by the $\mathcal{M}_\mathcal{O}$, we send the query ⟨`a.a.b.a.dfa.com.`, `TXT`⟩ to the nameserver serving z. The steps followed by the nameserver to resolve the query are shown below.

$$\begin{tikzcd}
\langle \texttt{a.a.b.a, TXT} \rangle \arrow[r, "\textcircled{1}"] & \langle \texttt{a.a.b.a.p, TXT} \rangle \arrow[r, "\textcircled{3}"] & \langle \texttt{a.a.b.q, TXT} \rangle \arrow[d, "\textcircled{6}"] \\
\textcircled{7} & \langle \texttt{a.p, TXT} \rangle \arrow[l] & \langle \texttt{a.a.q, TXT} \rangle \arrow[l, "\textcircled{5}"']
\end{tikzcd}$$

The nameserver returns the entire trace along with the TXT (⑦) record. Since the TXT record received contains "`accept`" in its content, the string is accepted by $\mathcal{M}_\mathcal{O}$.

In practice, there are zone files with millions of records; therefore, complex DFAs with many states and transitions can easily be encoded in DNS. We wrote a small script to

Start	①	a	IN	DNAME	a.p
	②	b	IN	DNAME	b.p
Transition	③	a.p	IN	DNAME	q
	④	b.p	IN	DNAME	p
	⑤	a.q	IN	DNAME	p
	⑥	b.q	IN	DNAME	q
Descision	⑦	a.p	IN	TXT	"accept"
	⑧	b.p	IN	TXT	"reject"
	⑨	a.q	IN	TXT	"reject"
	⑩	b.q	IN	TXT	"accept"

Table 2: Zone file z showing the enconding of DFA $\mathcal{M}_\mathcal{O}$ shown in Figure 1.

encode a DFA in DNS and successfully tested it with two popular DNS implementations, Bind [9] and Nsd [20]. In DNS, the domain name has certain length restrictions; specifically, the domain name cannot be longer than 255 characters, and each label cannot be more than 63 characters. The nameserver can also limit the number of rewrites that it will perform on a query. However, various techniques can be used to overcome such limitations. For example, we can map pairs of alphabet symbols from the DFA to single labels in DNS and then change the encoding of the transition relation to consume multiple symbols at a time, thereby processing longer DFA input strings.

3.3 Applications

Regexes are frequently used to validate user input for well-formedness. For example, the regex "`^[a-zA-Z0-9+_.-]+@[a-zA-Z0-9.-]+`" is a simple validator for email addresses. Since a regex can be represented as a DFA [29], using the construction detailed in the previous subsection we can validate if user input is a proper email address or not.

While the idea of using the DNS to check input well-formedness may seem far-fetched, we believe that it could have some natural use cases. For example, organizations generally want to control what domains their employees can visit while using their office devices, due to security and various other reasons. If the allowed domains can be represented as a regular expression, then this validation can be done in the DNS, *as part of the DNS lookup for the domain.* Office devices are generally configured to use specific DNS resolvers. Therefore, the resolver could first use our approach, with a local DNS nameserver implementing the policy DFA, to check that the user's DNS query is to an allowed domain, and only then send it to the outside world in order to resolve it to an IP address. A similar setup could be used for parental control in the home setting. Doing this directly in the DNS gives a single, global, always-available vantage from which to enforce policies.

4 PUSHDOWN SYSTEM

While the DNS can encode finite automata, its expressiveness goes beyond that of regular languages. In this section, we show that the DNS can express nondeterministic pushdown systems and by extension can generate strings in arbitrary context-free languages.

Definition 4.1. A pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ is a quadruple, where P and Γ are finite sets called the control locations and the stack alphabet, respectively. A configuration of $\mathcal{P}$ is a pair $\langle p, w\rangle$, where $p \in P$ and $w \in \Gamma^*$, and c_0 is the initial configuration. The set of all configurations is denoted by $Conf(\mathcal{P})$. Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$, which consists rules of the form $\langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', w\rangle$, where $p, p' \in P, \gamma \in \Gamma$, and $w \in \Gamma^*$. These rules define the transition relation $\Rightarrow$ between configurations of $\mathcal{P}$ as follows: *If* $\langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', w\rangle$, *then* $\langle p, \gamma w'\rangle \Rightarrow \langle p', ww'\rangle$ *for all* $w' \in \Gamma^*$. As shown above, each step depends only on the control location (p) and the topmost element (γ) of the stack ($\gamma w'$). The rest of the stack (w') is unchanged and has no influence on the possible next actions. A pushdown system may have infinitely many reachable states. An important use of pushdown system is in representing sequential programs with (possibly recursive) functions [25]. These programs in general cannot be modeled using FSMs as there is no limit on the depth of the call stack for function calls [28].

4.1 Encoding a PDS in DNS

With the help of an example, we show how a PDS can be encoded in DNS. Similar to how we encoded a DFA, we will encode both the stack and the current state in the query and use DNAME records to implement the transition relation. We employ multiple zone files and nameservers and *delegate* among them to encode any nondeterminism in the transition relation.

Consider a PDS $\mathcal{P}$ with $P = \{\mathrm{p}, \mathrm{q}\}$, $\Gamma = \{a, b, c, d\}$, $c_0 = \langle \mathrm{p}, c\rangle$ and Δ given by:

$$\begin{aligned} r_1 &= \langle \mathrm{p}, a\rangle \hookrightarrow \langle \mathrm{q}, b\rangle & r_2 &= \langle \mathrm{p}, a\rangle \hookrightarrow \langle \mathrm{p}, c\rangle \\ r_3 &= \langle \mathrm{q}, b\rangle \hookrightarrow \langle \mathrm{p}, d\rangle & r_4 &= \langle \mathrm{p}, c\rangle \hookrightarrow \langle \mathrm{p}, ad\rangle \\ r_5 &= \langle \mathrm{p}, d\rangle \hookrightarrow \langle \mathrm{p}, \epsilon\rangle \end{aligned}$$

We show some transitions between different configurations of $\mathcal{P}$ starting with c_0 and with the rules given by Δ.

$$\begin{tikzcd}
 & & \langle \mathrm{q},\ bd\rangle \arrow[r] & \langle \mathrm{p},\ dd\rangle \arrow[r] & \langle \mathrm{p},\ d\rangle \arrow[r] & \langle \mathrm{p},\ \epsilon\rangle & \text{(i)} \\
\langle \mathrm{p},\ c\rangle \arrow[r] & \langle \mathrm{p},\ ad\rangle \arrow[ur] \arrow[dr] & & & \langle \mathrm{q},\ bdd\rangle \arrow[r] & {} & \text{(ii)} \\
 & & \langle \mathrm{p},\ cd\rangle \arrow[r] & \langle \mathrm{p},\ add\rangle \arrow[ur] \arrow[dr] & & & \\
 & & & & \langle \mathrm{p},\ cdd\rangle \arrow[r] & {} & \text{(iii)}
\end{tikzcd}$$

As with the DFA encoding, we assume we control the pds.com. domain and all its subdomains. We create the pds.com. zone file as shown below and place it in the server1.pds.com. nameserver. The resolver is bootstrapped with the IP address of this nameserver.

nameserver: server1.pds.com.

```
$ORIGIN pds.com.
b.q      IN  DNAME  d.p       1
c.p      IN  DNAME  d.a.p     2
d.p      IN  DNAME  p         3
a.p      IN  NS     server2   4
a.p      IN  NS     server3   5
server2  IN  A      2.2.2.2   6
server3  IN  A      3.3.3.3   7
```

In the pds.com. zone, we first encode all the deterministic rules, r_3, r_4, r_5 with DNAME records, $\boxed{1}$, $\boxed{2}$, and $\boxed{3}$. We then use DNS delegation (referral to a new nameserver) for each nondeterministic set of rules. This apporach leverages the nondeterminism inherent in DNS delegation – given multiple NS records, DNS implementations will chose one nondeterministically. Here we have only one set of nondeterministic rules, namely $\langle \mathrm{p}, a\rangle$ with two rules. For each rule, we create an NS record ($\boxed{4}$ and $\boxed{5}$) and assign it a nameserver not previously assigned. For each NS record, we also add a glue record ($\boxed{6}$ and $\boxed{7}$) to provide the IP address of the nameserver.

We then create a zone file for a.p.pds.com. at each of the delegated nameservers and place a DNAME record for each nondeterministic rule at a unique nameserver. In our example we end up with two nameservers and zone files:

nameserver: server2.pds.com.

```
$ORIGIN a.p.pds.com.
a.p.pds.com.  IN  DNAME  b.q.pds.com.   8
```

nameserver: server3.pds.com.

```
$ORIGIN a.p.pds.com.
a.p.pds.com.  IN  DNAME  c.p.pds.com.   9
```

To execute the PDS from the initial configuration $\langle \mathrm{p}, c\rangle$, we ask the DNS query $\langle \beta.\mathrm{c.p},\ \mathtt{TXT}\rangle$. As with the DFA, the query encodes the current stack followed by current state. Additionally, we start the query with a dummy subdomain β. This is necessary since DNAME records only apply to strict subdomains; doing so ensures that the DNAME records apply even when the stack contains only a single element.

One possible execution starting from the query $\langle \beta.\mathrm{c.p},\ \mathtt{TXT}\rangle$ at the resolver is as follows:

(1) **Resolver:** Queries the default server Server1 with the query $\langle \beta.\mathrm{c.p},\ \mathtt{TXT}\rangle$.
(2) **Server1:** The server first rewrites the query, and the best records for the new query are NS records. The server returns the rewrite, the delegation records, and the corresponding glue records to the resolver.

(a) R_1 - $\langle\beta$.c.p, TXT$\rangle \xrightarrow{\boxed{2}} \langle\beta$.d.a.p, TXT$\rangle$

(b) Delegation - $\boxed{4}$, $\boxed{5}$, $\boxed{6}$, and $\boxed{7}$

(3) **Resolver:** The resolver now has a choice to contact either Server2 ($\boxed{4}$) or Server3 ($\boxed{5}$). We show the sequence of steps if the resolver sends the query $\langle\beta$.d.a.p, TXT$\rangle$ to Server2.

(4) **Server2:** Rewrites the query and returns it.

R_2 - $\langle\beta$.d.a.p, TXT$\rangle \xrightarrow{\boxed{8}} \langle\beta$.d.b.q, TXT$\rangle$

(5) **Resolver:** Queries Server1 again.

(6) **Server1:** Rewrites the query three times and returns the final rewritten query ($\langle\beta$.p, TXT$\rangle$) to the resolver.

(a) R_3 - $\langle\beta$.d.b.q, TXT$\rangle \xrightarrow{\boxed{1}} \langle\beta$.d.d.p, TXT$\rangle$

(b) R_4 - $\langle\beta$.d.d.p, TXT$\rangle \xrightarrow{\boxed{3}} \langle\beta$.d.p, TXT$\rangle$

(c) R_5 - $\langle\beta$.d.p, TXT$\rangle \xrightarrow{\boxed{3}} \langle\beta$.p, TXT$\rangle$

When the resolver gets the response from the Server1 in step 6, it is clear that the stack is empty as the domain name has only the control symbol and the dummy subdomain. If we put together all the rewrites (R_1, R_2, R_3, R_4, and R_5) starting from the first rewrite then we have the trace corresponding to the top trace (i) shown earlier in transitions between different configurations in §3.2. A different set of configurations can be explored if the resolver instead chooses Server3 at step 3.

In this way we can use the DNS to explore the reachable configurations of a PDS. Generally records returned to the resolver have a time to live (TTL) field for caching. The resolver will use the local cache when a matching query comes, thus slowing down the exploration of other configurations. We can avoid this by setting the TTL of the DNAME records to be small, even to 0. Another issue is that nameservers often have a limit on the number of rewrites they will perform, at which point they stop processing the query further and return it. To overcome this limitation and explore more configurations, the resolver can then send a fresh query from that last configuration.

So far, we have seen how to explore reachable configurations of a PDS using the DNS. In the next subsection we will describe how we can use this capability to *generate* strings from any context-free language.

4.2 Context-free Language Generator

A formal grammar is a set of production rules that describe all possible strings in a given formal language. A context-free grammar (CFG) is a formal grammar whose production rules are of the form "$A \longrightarrow \alpha$", with A being a single nonterminal symbol, and α a string of terminals and/or nonterminals (α can be empty). Context-free grammars generate context-free languages, which are strictly more expressive than regular expressions. Context-free languages have many applications in programming languages; in particular, most programming language syntaxes are specified by context-free grammars.

Formally, a context-free grammar $\mathcal{G}$ is defined as a 4-tuple $\mathcal{G} = (N, \Sigma, P, S)$, where N is a finite set of non-terminal symbols, and Σ is a finite set of terminal symbols disjoint from N. The set of terminals is the alphabet of the language defined by the grammar $\mathcal{G}$. P is the set of production rules and is a finite relation in $N \times (N \cup \Sigma)^*$. S is the start symbol and is one of the non-terminal symbols in N.

We derive strings in the language of a CFG by starting with the start symbol and repeatedly replacing some non-terminal by the right side of one of its production rules. Consider the context-free language $L = \{a^n b^n : n \geq 1\}$. The grammar of this language, with the start symbol S is:

$$S \longrightarrow aSb \tag{1}$$

$$S \longrightarrow ab \tag{2}$$

The string a^3b^3 in this language is generated by applying rule (1) twice followed by (2): $S \longrightarrow aSb \longrightarrow aaSbb \longrightarrow aaabbb$.

We first describe a program variant of the above grammar and show how that program can be represented using a pushdown system. Then based on the encoding described in §4.1 we can implement this in the DNS.

A program that generates the strings in L is:

```
    procedure S1:        procedure S2:         procedure S:
ℓ1   output a        ℓ5   output a        ℓ8   if ?
ℓ2   call S          ℓ6   output b        ℓ9    call S1
ℓ3   output b        ℓ7   return          ℓ10  else call S2
ℓ4   return                               ℓ11  return
```

Here ℓ_1, ℓ_2, and other such symbols are used to denote each program location (line of code) uniquely, which will be later used as the stack alphabet Γ in our PDS. Since S is the start symbol in the grammar, the `procedure S` would be called to start the program. The symbol "?" in ℓ_8 represents nondeterministic choice, reflecting the nondeterminism in the original grammar.

The technique used to convert the above grammar into a program can be generalized as follows. Let $\mathcal{G}$ be a CFG. First, create a uniquely named procedure (disjoint from $\Sigma \cup N$) for each production rule in P, as in S1 and S2 in the above example. The body of the procedure then encodes the right side of the corresponding rule. Specifically, there is an "`output t`" line for each terminal symbol t in the right side and a "`call A`" line for each non-terminal A in the right side of the rule, in order of their appearance in the rule. Finally, for every non-terminal A in N create a "`procedure A`" and use an `if` statement to nondeterministically call one of the procedures created in the previous step whose corresponding rule has A on the left side.

We can create a PDS that encodes all possible executions of such a program [28]. The PDS has a single control location and uses the program labels as the stack alphabet. For example, a PDS $\mathcal{P}_L$ for the program shown above has $P = \{\mathrm{p}\}$, $\Gamma = \{\ell_1, \cdots, \ell_{11}\}$, and $c_0 = \langle \mathrm{p},\ \ell_8 \rangle$. Δ is given by:

$$\begin{array}{lll}
\langle \mathrm{p},\ \ell_1\rangle \hookrightarrow \langle \mathrm{p},\ \ell_2\rangle & \langle \mathrm{p},\ \ell_5\rangle \hookrightarrow \langle \mathrm{p},\ \ell_6\rangle & \langle \mathrm{p},\ \ell_8\rangle \hookrightarrow \langle \mathrm{p},\ \ell_9\rangle \\
\langle \mathrm{p},\ \ell_2\rangle \hookrightarrow \langle \mathrm{p},\ \ell_8\ell_3\rangle & \langle \mathrm{p},\ \ell_6\rangle \hookrightarrow \langle \mathrm{p},\ \ell_7\rangle & \langle \mathrm{p},\ \ell_8\rangle \hookrightarrow \langle \mathrm{p},\ \ell_{10}\rangle \\
\langle \mathrm{p},\ \ell_3\rangle \hookrightarrow \langle \mathrm{p},\ \ell_4\rangle & \langle \mathrm{p},\ \ell_7\rangle \hookrightarrow \langle \mathrm{p},\ \epsilon\rangle & \langle \mathrm{p},\ \ell_9\rangle \hookrightarrow \langle \mathrm{p},\ \ell_1\ell_{11}\rangle \\
\langle \mathrm{p},\ \ell_4\rangle \hookrightarrow \langle \mathrm{p},\ \epsilon\rangle & & \langle \mathrm{p},\ \ell_{10}\rangle \hookrightarrow \langle \mathrm{p},\ \ell_5\ell_{11}\rangle \\
 & & \langle \mathrm{p},\ \ell_{11}\rangle \hookrightarrow \langle \mathrm{p},\ \epsilon\rangle
\end{array}$$

Intuitively, Δ encodes the control flow of the program. For statements where control passes from one line to the next (here just `output`), we add rules of the form $\langle \mathrm{p},\ \ell_1\rangle \hookrightarrow \langle \mathrm{p},\ \ell_2\rangle$. A procedure call (for example, at ℓ_2) is encoded by pushing the return point (ℓ_3) followed by the called procedure's (ℓ_8) starting statement. A `return` statement is encoded as a stack pop (ϵ).

Finally, we can use this encoding to generate strings in our original CFG L. Define a *full trace* in $\mathcal{P}_L$ as the sequence of configurations starting with c_0 and ending with an empty stack. Given a full trace, consider the top symbol of the stack in each configuration, and retain only those symbols that correspond to an `output` code line. If we concatenate the output of those lines, then we obtain a string. The set of such strings is exactly the set of strings defined by the CFG L.

For example, the full trace that would generate a^3b^3 is shown in Figure 2. In the trace, among all the top stack elements only six symbols, shown with circles and squares, represent `output` code lines. If we concatenate them in the order given by the full trace then we obtain the string *aaabbb*.

In summary, we have shown how to encode a PDS in DNS in §4.1, and here we have shown how to encode a generator for a CFG as a PDS. Hence we can use the DNS system to generate strings in the language of a given CFG.

$$\begin{tikzcd}
\langle \mathrm{p},\ \ell_8\rangle \arrow[r] & \langle \mathrm{p},\ \ell_9\rangle \arrow[r] & \langle \mathrm{p}, \textcircled{ℓ_1}\ell_{11}\rangle \arrow[r] & \langle \mathrm{p},\ \ell_2\ell_{11}\rangle \arrow[r] & \langle \mathrm{p},\ \ell_8\ell_3\ell_{11}\rangle \arrow[dll] \\
\langle \mathrm{p},\ \ell_2\ell_{11}\ell_3\ell_{11}\rangle \arrow[d] & \langle \mathrm{p}, \textcircled{ℓ_1}\ell_{11}\ell_3\ell_{11}\rangle \arrow[l] & \langle \mathrm{p},\ \ell_9\ell_3\ell_{11}\rangle \arrow[l] & & \\
\langle \mathrm{p},\ \ell_8\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[r] & \langle \mathrm{p},\ \ell_{10}\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[r] & \langle \mathrm{p}, \textcircled{ℓ_5}\ell_{11}\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[d] & & \\
\langle \mathrm{p},\ \ell_{11}\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[d] & \langle \mathrm{p},\ \ell_7\ell_{11}\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[l] & \langle \mathrm{p}, \boxed{\ell_6}\ell_{11}\ell_3\ell_{11}\ell_3\ell_{11}\rangle \arrow[l] & & \\
\langle \mathrm{p}, \boxed{\ell_3}\ell_{11}\ell_3\ell_{11}\rangle \arrow[r] & \langle \mathrm{p},\ \ell_4\ell_{11}\ell_3\ell_{11}\rangle \arrow[r] & \langle \mathrm{p},\ \ell_{11}\ell_3\ell_{11}\rangle \arrow[dr] & & \\
\langle \mathrm{p},\ \epsilon\rangle & \langle \mathrm{p},\ \ell_{11}\rangle \arrow[l] & \langle \mathrm{p},\ \ell_4\ell_{11}\rangle \arrow[l] & \langle \mathrm{p}, \boxed{\ell_3}\ell_{11}\rangle \arrow[l] &
\end{tikzcd}$$

Figure 2: The full trace of PDS $\mathcal{P}_L$ that generates string a^3b^3. The output code lines that are on top of the stack are shown with circles for a and squares for b.

5 DISCUSSION

Our paper represents an initial investigation into the complexity of DNS, and there are several directions for future research.

Impact on DNS Verification. The most efficient algorithms known for PDS reachability — determining whether a given configuration can be reached in a given PDS — have near-cubic time complexity in the number of rules [4–6, 12, 14]. Hence DNS zone-file verification [17], which requires reasoning about all possible query lookups, also has at least this complexity today. This has not only theoretical implications but is also a problem for real zone-file verifiers like GRoot [17]. Even a simple four-record zone file with interacting `DNAME` loops can create close to a million query equivalence classes in GRoot, quickly blowing up its verification time. Interestingly, however, verification in GRoot is linear time in the absence of `DNAME` records. Can we design new verification algorithms that scale well with the number of `DNAME` records for real-world configurations?

Tighter Bounds. Can we show that the DNS is even more expressive than a PDS? Alternatively can we reduce the DNS to a PDS and hence show that they are equivalent?

Applications. The applications that we have presented are somewhat contrived. Can we build a real application that takes advantage of the complexity of DNS? We can take inspiration from existing applications that use the DNS, ranging from service discovery [7] to load balancing [2, 16] to spam filtering [10, 11, 18, 19].

New Record Types. Contributors frequently add new drafts and RFCs to the DNS specification, with new record types intended to enable new use cases. For example, the recent NAPTR record type [21] supports prioritized regular expressions that provide lookup for dynamic resources. How do these newly proposed types affect DNS complexity?

Security Implications. Does the complexity of DNS have security implications? This is a natural direction to explore. However, we note that, unlike for conventional DNS attacks, the attacker must control the target's zone files in order to leverage the complexity of DNS.

6 CONCLUSION

In this paper, we have investigated the computational complexity of DNS and shown its ability to simulate both a finite-state machine and a pushdown system. While this work is in the spirit of earlier investigations into the complexity of protocols like BGP, we note that, unlike BGP, DNS features are available to applications. Thus in addition to the implications for verification complexity, the computational power of DNS is potentially an enabler of new applications.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1704336 and CNS-1901510.

REFERENCES

[1] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory*, Antoni Mazurkiewicz and Józef Winkowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150.

[2] Thomas P. Brisco. 1995. DNS Support for Load Balancing. RFC 1794. (1 April 1995). https://doi.org/10.17487/RFC1794

[3] Olaf Burkart and Bernhard Steffen. 1995. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing* 2, 2 (1995), 89–125.

[4] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[5] Krishnendu Chatterjee and Georg Osang. 2017. Pushdown reachability with constant treewidth. *Inform. Process. Lett.* 122 (2017), 25–29.

[6] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1328438.1328460

[7] Stuart Cheshire and Marc Krochmal. 2013. DNS-Based Service Discovery. RFC 6763. (Feb. 2013). https://doi.org/10.17487/RFC6763

[8] Marco Chiesa, Luca Cittadini, Giuseppe Di Battista, Laurent Vanbever, and Stefano Vissicchio. 2013. Using routers to build logic circuits: How powerful is BGP?. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[9] Internet Systems Consortium. 2021. BIND 9. (2021). https://www.isc.org/bind/

[10] DNS Response Policy Zones 2019. (2019). Retrieved June 2020 from https://dnsrpz.info/

[11] DNSBL information - spam database and blacklist check. 2020. (2020). https://www.dnsbl.info/

[12] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient algorithms for model checking pushdown systems. In *International Conference on Computer Aided Verification*. Springer, 232–247.

[13] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking* 10, 2 (2002), 232–243.

[14] Nevin Heintze and David McAllester. 1997. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 342–351.

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[16] Internet Initiative Japan Inc. 2019. *IP Location Load Balancing Resource Record*. Internet-Draft draft-sonoda-dnsop-lb-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-sonoda-dnsop-lb-01 Work in Progress.

[17] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. https://doi.org/10.1145/3387514.3405871

[18] Scott Kitterman. 2014. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208. (April 2014). https://doi.org/10.17487/RFC7208

[19] Murray Kucherawy, Dave Crocker, and Tony Hansen. 2011. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376. (Sept. 2011). https://doi.org/10.17487/RFC6376

[20] NLnet Labs. 2021. NSD. (2021). https://nlnetlabs.nl/projects/nsd/about/

[21] Michael H. Mealling. 2002. Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database. RFC 3403. (Oct. 2002). https://doi.org/10.17487/RFC3403

[22] P. Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. (Nov. 1987). https://doi.org/10.17487/RFC1034

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

[24] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. (Jan. 2006). https://doi.org/10.17487/RFC4271

[25] Thomas Reps, Akash Lal, and Nick Kidd. 2007. Program Analysis Using Weighted Pushdown Systems. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, V. Arvind and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–51.

[26] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263.

[27] Scott Rose and Wouter Wijngaards. 2012. DNAME Redirection in the DNS. RFC 6672. (June 2012). https://doi.org/10.17487/RFC6672

[28] Stefan Schwoon. 2002. *Model-checking pushdown systems*. Ph.D. Dissertation. Technische Universität München.

[29] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

[30] Paul A. Vixie and Vernon Schryver. 2018. *DNS Response Policy Zones (RPZ)*. Internet-Draft draft-vixie-dnsop-dns-rpz-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-vixie-dnsop-dns-rpz-00 Work in Progress.