Networks of Hosts Implementing DNS Are NP-Complete

by

Mak Kolybabi

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2019

Thesis advisor Author

Michael Domaratzki **Mak Kolybabi**

Networks of Hosts Implementing DNS Are NP-Complete

Abstract

The Internet core protocols are the networking protocols that the Internet is built on, a subset of which are implemented on every device connected to the Internet. The specifications of these protocols can span dozens of documents, making accurately understanding and implementing them difficult. The Domain Name System is one of the Internet's core protocols, providing a distributed, hierarchical database primarily intended to map between the names and addresses of hosts. Modern DNS contains a plethora of complex and interacting features, defined across a number of documents, evolving over the course of over thirty years. I show that a network of specially-configured hosts implementing a subset of DNS operate as logic gates. I then describe how such gates can communicate to form Boolean circuits. Finally, I discuss CSAT and NP-completeness.

Acknowledgments

I'd like to start by thanking my thesis advisor, Dr. Michael Domaratzki, for accepting me as a graduate student despite me having already dropped out of Master's program once. Without his patience, assistance, and understanding, I would have dropped out a second time.

I'd also like to thank my friend David Curry for his many years of collaborating with me setting up networks and servers in ridiculous and unwise configurations. Those activities gave me experience with the network protocols and daemons that I needed to understand to be able to dream up this thesis, and the confidence to push them to behave in ways their authors did not intend.

Finally, I am deeply indebted to the LangSec community and all the presentations and papers they have produced. The weird machines they have constructed are an inspiration.

My time since beginning my second attempt at the Master's program has spanned the births of three children, three residences, and five jobs. So this is dedicated to my wife, Katie, whose support makes all things in my life possible.

Contents

List of Figures

Chapter 1

Overview

The Internet consists of billions of devices communicating with each other. These devices all communicate using a handful of well-defined protocols colloquially known as the Internet core protocols [Baker, 2009]. The details of those protocols are described in documents called Requests For Comments (RFCs) [Bradner, 1996]. One of the Internet's core protocols, whose definition spans dozens of RFCs across more than thirty years, is the Domain Name System (DNS) [RFC, 1987b].

DNS provides a queryable, distributed, hierarchical database that was designed to allow devices on the Internet to access information about other devices, destinations, and organizations. DNS is a critical service for the Internet, holding the information necessary for most machines to find one another. DNS supports many different types of records, two of which are of particular interest to us: `A` and `CNAME` [RFC, 1987b, Section 3.2.2]. `A` records define a mapping between a human-readable name (i.e., a lookup key) in the database and an IP address [RFC, 1981] that can be used to communicate with a device. `CNAME` records define an association between an alias name and a canonical name, allowing multiple names to be associated with a single value.

Hosts that implement DNS, acting as part of the distributed, hierarchical database, frequently need to contact other hosts to access portions of the database that the original host does not store. There are three ways of performing lookups: iteration, recursion, and forwarding. Iterative (zero-delegation) lookups are handled by the host that initiated them, with the host performing additional requests as needed when partial answers or references to other records are received. Recursive (single-delegation) lookups allow the host that initiated the lookup to delegate any additional lookups to a single host. Recursive queries received by a host willing to service them are then treated locally as iterative lookups, with their final answers returned to the host that initiated the lookup. Forwarding (multiple-delegation) differs from recursion by sending additional queries as recursive instead of iterative.

In recent years there has been an effort to analyze common file formats, hardware devices, and network protocols from the perspective of building so-called "weird machines": automata built in unexpected media [Oakley and Bratus, 2011; Shapiro et al., 2013; Bangert et al., 2013; Chiesa et al., 2013; Veldhuizen, 2003]. By adopting that perspective in this thesis, we will use a network of hosts implementing DNS, configured in a specific manner employing `CNAME` records and forwarding, to construct logic gates. These logic gates can be joined together to create Boolean circuits.

We claim that instances of the Circuit Satisifiability problem (CSAT) can be reduced to these networks. We will describe the construction and operation of these networks, and define a decision problem named DNS-SIM that applies to them. With that in place, we will show a reduction from CSAT to DNS-SIM, and conclude that DNS-SIM is NP-complete.

Chapter 2

Theory

This chapter will briefly introduce the theoretical computer science concepts that are the background of this thesis. For more background on computational complexity, see Garey and Johnson [1990] or Papadimitriou [1994].

2.1 Logic Gates

Logic gates are implementations of Boolean functions. Logic gates have been constructed in many media, and we will construct three types of gates using networks of hosts implementing the DNS protocol: AND, OR, and NOT.

A mapping of all possible input combinations to their resulting outputs is called a truth table.

2.2 Functional Completeness

A set of Boolean operators is functionally complete if it can be used to construct any possible truth table. Both the sets $\{\text{AND}, \text{NOT}\}$ and $\{\text{OR}, \text{NOT}\}$ are functionally complete, as are $\{\text{NAND}\}$ and $\{\text{NOR}\}$. We will describe how to construct each of these five gates in Chapter 5.

2.3 Boolean Circuits

Boolean circuits are implementations of Boolean functions constructed from logic gates. With a functionally complete set of logic gates, any Boolean function of the form $f : \mathbb{B}^n \rightarrow \mathbb{B}$ of n input variables to one output can be constructed.

2.4 Turing Machines

Turing machines are abstract models of computation. These machines consist of a current state, a segmented tape of infinite length, a read/write head placed over a segment of tape, and a table of state transitions. Using both the current state and the symbol under the read/write head as indexes into the state transition table, the machine performs the described state transition. State transitions consist of a new internal state, a symbol to be written on the tape under the read/write head, and a left-or-right movement of the read/write head. If no transition is defined for the machine's internal state combined with the symbol under the read/write head, the machine halts. A subset of states are designated as accepting states. The word initially written on the tape is said to be accepted if the machine halts in an accepting state.

2.5 P-Completeness

The complexity class P contains all problems that can be solved by a deterministic Turing machine in polynomial time. Any problem in P can be reduced to a P-complete problem using a log-space reduction (i.e., the mapping from problem to problem can be done with a log-space Turing machine). P-complete problems are interesting not only for their usefulness as targets of reductions, but also as inherently sequential problems which resist parallelization. P-complete problems are the hardest problems in P: if a P-complete problem can be solved, e.g., in log space, then all problems in P can be solved in log space.

2.6 NP-Completeness

The complexity class NP contains all problems with answers that can be verified by a deterministic Turing machine in polynomial time, though those answers may be generated by a non-deterministic Turing machine in polynomial time. All problems within NP, including those in P, can be reduced to any NP-complete problem by a deterministic machine in polynomial time. NP-complete problems are interesting because methods of solving them are applicable to all other problems in NP via reduction.

2.7 The Circuit Satisfiability Problem

The Circuit Satisfiability Problem (CSAT) considers a Boolean circuit consisting of a number of inputs and a single output, and asks whether there is any assignment of inputs that results in an output of true. CSAT is NP-complete [Garey and Johnson, 1990].

Chapter 3

Protocols

This chapter will briefly introduce the networking concepts that are the background of this thesis.

3.1 The Internet Protocol

The Internet Protocol (IP) [RFC, 1981] is a stateless protocol for communication between hosts. IP is the de facto protocol that hosts on the Internet use to communicate with one another. There are two versions of IP in common use, IPv4 and IPv6 [Deering and Hinden, 2017]. IP introduces the concept of an 'address' to refer to each of the logical network interfaces attached to hosts on each side of a communication. IPv4 provides an address space of 2^{32} while IPv6 provides an address space of 2^{128}.

3.2 User Datagram Protocol

The User Datagram Protocol (UDP) [RFC, 1980] is a stateless protocol for communication between applications. On the Internet, UDP packets are encapsulated within IP packets for transmission between hosts. UDP introduces the concept of a 'port' to designate the application on each side of a communication. UDP is distinguished by its minimalism, omitting all other features, except for a checksum.

3.3 The Domain Name System

The Domain Name System (DNS) [RFC, 1987b] is a global, distributed, hierarchical database that contains information about hosts on the Internet. The simplest use of DNS is to look up IP addresses by hostname. For instance, when a Web browser navigates to `https://cs.umanitoba.ca/`, it queries DNS for the list of IP addresses associated with the domain name `cs.umanitoba.ca`. Upon receiving a DNS response containing one or more IP addresses, the Web browser will connect to each of those addresses in sequence and request the associated Web page, stopping when a request is successful. DNS runs over UDP, encouraging the former to be stateless in its main use case.

3.4 Hosts

A host is an entity on the network that is addressable via at least one IP address, can communicate over UDP/IP with other hosts, and implements the DNS protocol in some capacity. Hosts may send both DNS requests and responses. Hosts contain caches of previously-seen responses that can be used to respond to requests. Hosts also contain zone files that store the resource records that collectively comprise the 'database' of DNS.

3.5 Interfaces

Interfaces are points on a host where it connects to a network. Hosts may have any number of interfaces, and each may attach to either the same or different networks or types of networks as the host's other interfaces. Each interface will usually have its own address.

In this thesis, all hosts in my constructions 5 are directly connected to other hosts, without the use of a packet-switched network. This means that each host needs a dedicated interface for each host with which it communicates.

3.6 Request Types

DNS is a request-response protocol, meaning a client sends a request to a server, the server responds to the request, and the interaction is then considered complete. Requests may be marked as either iterative or recursive. An iterative request results in an immediate response from the receiving host, which provides the best information it has locally available. If the information in the response is insufficient, it is the responsibility of the requesting host to perform further queries in search of more information.

If a request is marked as recursive, and the receiving host supports recursion, the host may send its own requests into the network to find the information necessary to respond. These requests are necessary if the receiving host does not have the requested information locally, or a response from one of its requests indicates that the requested information is stored elsewhere in the network.

3.7 Forwarding

Recursive requests by themselves are limited by the restriction that the first host to receive a request that is marked as recursive will only issue iterative requests to other hosts. However, forwarding passes a request from one host to another unless the first host can answer the question from its zone files or cache. Forwarded requests are always marked as recursive. Forwarding is commonly used in home routers to pass queries to ISPs instead of resolving them locally.

Recursion alone will be unsuitable for the purpose of this thesis, which will require requests that travel through multiple hosts like electrical impulses travel through circuits. In Section 5.2 we will require that a host receiving a recursive query be capable of generating further recursive queries, for which we need to employ forwarding.

3.8 Request Timeouts

When a host does not receive a response to a request that it issued within a period of time, the host will forward the request to the host listed in the next `FWD` directive in its configuration file. This timeout period can be configured in each host separately. If all `FWD` directives in the zone file have been exhausted, the host will send a failure response back to the host from which it received the request it is currently trying to resolve.

3.9 Labels

The DNS hierarchy is a tree of nodes, with each node having a label and some data (i.e., a zone file) associated with it. Labels are alphanumeric strings that may contain hyphens. A label may be up to 63 characters in length. The root of the tree is represented by a zero-length string [RFC, 1987a, Section 3.1].

3.10 Names

Each node in the DNS hierarchy is named, with the names specifying the path to the node from the tree's root. These names are composed of labels separated by periods. For example, the name `cs.umanitoba.ca.` is a sequence of four labels: `cs`, `umanitoba`, `ca`, and the root. The combined length of a name's labels must be no longer than 255 characters [RFC, 1987b, Section 2.3.4]. Names are frequently called host names or domain names, depending on their current usage.

Just below the root node in the DNS hierarchy are the top-level domains (TLDs), many of which are commonly known (e.g., `com.`, `ca.`, `wtf.`). At the time of writing in August 2019, there are 1,528 TLDs in existence, with hundreds of millions of domains registered in `com.` alone. While DNS is case insensitive [RFC, 1987b, Section 2.3.3], we will represent all domain names using lowercase for consistency.

3.11 Zone Files

Each node in the DNS hierarchy has an associated zone file, sometimes called a master file [RFC, 1987b, Section 5], containing resource records for nodes in the DNS hierarchy. Each zone file contains resource records, described below, for its node's label and possibly the labels of its children. For example, the zone file for `umanitoba.ca.` would contain a resource record for `cs.umanitoba.ca.`. Hosts may contain any number of zone files for unrelated names. Hosts with information for the same node in the DNS hierarchy may also contain different records, resulting in requests for the same information between one requestor-responder pair giving a different result than a different request-responder pair. Differences in responses have two common causes: stale caches and DNS views. When a host changes a record in its zone file for a name, it will henceforth respond with that new record, but any hosts with the old record cached will continue to respond with the old record until its TTL expires. DNS views are network configurations designed to give different DNS responses to different network segments. Hosts inside an organization's private network may receive internal IP addresses, and have more records available, versus hosts on the Internet.

3.12 Resource Records

Resource records are indexed by label, class, and type at a point within the DNS hierarchy. Under those indices, a time-to-live and a value can be stored. For example, at the `kolybabi.com.` node in the DNS hierarchy, the label `mail` has a record with the class IN and the type A that contains the IPv4 address `45.55.52.41` and a TTL of 1800 seconds. We now break down what each of those attributes mean.

3.12.1 Record Classes

Although each record has a class, the only class that remains valid at the time of writing is IN, meaning Internet [RFC, 1987b, Section 3.2.4].

3.12.2 Time To Live

Records must contain a Time To Live (TTL) field instructing hosts on the maximum length of time the received record should stay in their cache. A host's cache is an ephemeral set of records used for quickly answering previously-seen requests with previously-received responses. Records are removed from the cache when they have been stored longer than their TTL specifies.

TTLs store the number of seconds represented by unsigned 31-bit integers [Elz and Bush, 1997, Section 8], containing values from zero, indicating that the record is only allowed to be used for the current transaction, up to the field's maximum value of more than 68 years. I will omit mentioning a record's TTL and a host's cache unless it affects a specific example.

3.12.3 Record Types

A DNS record's type determines the format of the value it stores. Dozens of records have been defined across many DNS-related RFCs, but only four record types are relevant to this thesis.

Address Records

An `A` record [RFC, 1987b, Section 3.4.1] contains an IPv4 address, indicating that the domain name represented by the label can be accessed at the address in the data field. Domain names may have multiple `A` records, each with a different address, associated with them. Defining multiple addresses for a domain name is a common practice for load balancing and services requiring high availability.

An `AAAA` record [Ksinant et al., 2003, Section 2.2] contains an IPv6 address, and has the same properties and uses as an `A` record.

Nameserver Records

An `NS` record [RFC, 1987b, Section 3.3.11] contains a hostname, and is used by hosts to determine where to direct their requests for information stored in the DNS hierarchy. For example, a host can contact one of the Internet's root DNS servers to request the `NS` records for `com.`, which may provide a name like `ns1.com.`. The root servers can then be queried for the `A` records associated with `ns1.com.`. Finally, `ns1.com.` can be sent requests directly to inquire about labels beneath `com.` in the DNS hierarchy.

CNAME Records

`CNAME` records [RFC, 1987b, Section 3.3.1] provide a method of associating an alias with its canonical name. Several domain name aliases can point to a single canonical domain name. An alias may also point to another alias. For example, `www.cs.umanitoba.ca.` (alias) does not have an `A` record associated with it, but instead has a `CNAME` record that references `copper.cs.umanitoba.ca.` (canonical).

Chapter 4

Related Work

4.1 Computational Complexity

4.1.1 Turing Machines

In 1936, Turing published his model of computation, which we now call the Turing machine [Turing, 1936]. In the same paper, he proved that Turing machines had undecidable properties, meaning that there are certain questions we could ask of Turing machines that are fundamentally unanswerable. This notion of undecidability has far-reaching implications for the definition of file formats, network protocols, and any other input that is passed to a software or hardware interface. The reason is that every input can be considered a program targeting its recognizer [Bratus et al., 2014], imparting the computational power of the recognizer to the source of the input [Sassaman et al., 2011]. If a Turing machine is necessary to recognize or validate instances of an input format, then that format has properties that are undecidable [Sassaman et al., 2011]. Specifically, difficulties arise in determining whether a recognizer halts on a given input and whether two different implementations are equivalent. Both of these properties are of critical importance when handling untrusted data.

4.1.2 Chomsky Hierarchy

Twenty years after Turing, Chomsky introduced a hierarchy to categorize machines in decreasing order of power starting from Turing machines [Chomsky, 1956]. Each level of the hierarchy is capable of emulating machines at all lower levels. The machines comprising the hierarchy have differing properties, including fewer undecidable properties than Turing machines. Referring to a machine (or language) in terms of its place on the Chomsky hierarchy—the categories being regular, context-free, context-sensitive, or recursively-enumerable—is used to associate machines with the well-known properties exhibited by all members of that category. Subtle variations and sub-categories have been introduced to the hierarchy in the last fifty years, but for my purposes I consider the bottom half of the Chomsky hierarchy (regular and context-free languages) to be uninteresting, while the top half (context-sensitive and recursively enumerable languages) is where interesting things become possible.

Recent efforts have been made to analyze the complexity inherent in file formats, hardware interfaces, language features, and network protocols. Many of these analyses use a composition approach [Bratus et al., 2012], which involves framing features or operations of a subject as primitives upon which to build Turing machines. These machines built within file formats, network protocols, and hardware interfaces are known as weird machines, and have been defined as “machines that recognize/generate languages on a surprisingly and/or terrifyingly high position on the Chomsky hierarchy” [Palmer, 2016].

4.1.3 Network Protocols

The Border Gateway Protocol (BGP) is used by Internet Service Providers (ISPs) to exchange routing information between networks. It also has the distinction of being one of the Internet core protocols [**?**] as designated by the Internet Engineering Task Force (IETF), the organization responsible for publishing the Internet’s most important standards. BGP is known to be Turing-complete [Chiesa et al., 2013], as it can be used to effectively construct the circuits necessary to perform arbitrary computation. This makes BGP the only Internet core protocol with a published computational complexity; the other protocols have no such published analyses.

Other analyses of the computational complexity inherent in network protocols have focused on sub-components of those protocols. For example transport layer security (TLS), which provides encryption to other network protocols, has no published complexity analysis. However, a sub-component of TLS, ASN.1, is known to be context-sensitive [Kaminsky et al., 2010]. Due to the lack of analyses of the complexity of network protocols, the remainder of this chapter will focus on software and hardware platforms that have been evaluated in terms of the computational complexity they offer when building weird machines within them.

4.2 Weird Machines

4.2.1 CPU Features

The memory translation and protection machinery and the associated interrupt handling, collectively referred to as the memory management unit (MMU), provide the basis for non-cooperative multitasking in x86 processors. A privileged process, usually the operating system, populates several structures (page tables) and registers to define mappings (page table entries) between physical memory and its virtual representation in unprivileged processes. Unprivileged processes access memory through the MMU, such that the virtual address referenced are translated to physical addresses as specified by entries in the page table. If a page table entry is invalid, the MMU records the specifics of the memory access and passes control to an interrupt handler. It is possible to construct a situation in which the attempt to pass control to the interrupt handler causes another invalid page table entry to be referenced, resulting in a chain of references and indirections. The indirections can be thought of as state changes, while the MMU recording the specifics of the failed memory access can be considered as a tape. Bangert et al. [2013] demonstrated that arbitrary computation can be performed during this process, and created a compiler backend that targets the MMU. This allows the creation of Turing complete weird machines that run within the MMU.

Recent years have seen the introduction of many memory-protection mechanisms in operating systems and microprocessors. One such mechanism is the ability to set certain pages of memory as non-executable via the MMU. To circumvent this new security measure, return-oriented programming (ROP) was invented [Shacham, 2007]. After a vulnerability such as a stack-based buffer overflow has been exploited, and an attacker-controlled payload has been written to the stack, that payload would traditionally change the function's return address to that of the payload, passing it control of the CPU. Modern operating systems commonly instruct the MMU to make the pages comprising the stack non-executable. Modern payloads employ ROP to structure themselves as a series of stack frames with return addresses that target the program's own instructions at known locations within executable pages. These targeted instructions, normally near the end of functions, are known as ROP gadgets.

In the same spirit as ROP, but targeting microcontrollers as opposed to microprocessors, is interrupt-oriented programming (IOP) [Tan et al., 2014]. IOP creates gadgets out of spans of instructions starting from the beginning of interrupt handlers, and ending when the attacker triggers another interrupt, or at the end of a handler. Since both ROP and IOP have access to many instructions from the program or firmware, respectively, they can clearly be used to create a Turing machine, with the specific gadgets varying by target.

4.2.2 File Formats

The Executable Linking Format (ELF), in which most compiled applications on Unix platforms are stored, contains a facility for passing control to an exception-handling mechanism known as DWARF. DWARF provides the flexibility and extensibility to support the stack and register manipulation logic necessary to implement both current and future programming languages. It does this by defining a portable set of operations to be performed between an exception being thrown and it being caught by a language's exception handler. Exception handling, when offered by a programming language standard, is realized by the compiler which stores the language-specific exception handling operations in the program container. While the creation of DWARF sections in an ELF container is commonly performed by the compiler, it is possible to craft custom DWARF sections that perform computation beyond that necessary to implement language exception-handling semantics. Oakley and Bratus [2011] demonstrate that Turing-complete computation is possible using DWARF by treating its stack and register operations as a powerful bytecode that can perform branching and arbitrary arithmetic.

Elsewhere in the ELF standard, the run-time loader (RTLD) populates a process's virtual memory environment prior to passing control to the executable. The RTLD is driven by metadata in the ELF file containing the executable. That metadata can direct the RTLD to perform operations that can be framed as addition, assignment, and branching primitives. Shapiro et al. [2013] built a compiler that translates a Turing-complete subset of the Brainfuck programming language [Müller, 1993] into the primitives provided by the RTLD.

4.2.3 Programming Languages

While Brainfuck, C++, and other programming languages are Turing complete, Veldhuizen [2003] analyzed a particular compile-time feature of C++ to prove it was Turing complete. C++ templates are a compile-time replacement system for classes, to effectively allow the creation of generic classes via duplication and substitution of class definitions. Veldhuizen [2003] provides a method for encoding a Turing machine into a set of C++ templates, demonstrating that they can emulate any Turing machine and are therefore Turing complete.

4.3 Methodology

The analyses described in this chapter all used one of two methods to prove that their subjects are Turing complete. The first method was to provide a method for implementing Turing machines in the subject medium. The second method was to indirectly implement a Turing machine by implementing another set of primitives that are known to be Turing complete. Chiesa et al. [2013] used the second method when building logic circuits using BGP, and I will follow their example when making logic circuits using networks of hosts implementing DNS in Chapter 5.

Chapter 5

Construction

In this chapter, we show how to construct a logic circuit from a network of DNS hosts. We will show how each gate can be constructed using DNS hosts and zone files.

We begin with a discussion of how the DNS protocol will be used in this construction.

5.1 Protocol Assumptions

In Chapter 3, I described the aspects of the DNS protocol relevant to this thesis. In this section, I describe the simplifications that I have made to the protocol for the sake of brevity and clarity. These simplifications do not affect the applicability of the results to the unsimplified form of DNS.

5.1.1 Addresses

While either version of IP work with the constructions described in this chapter, I will use IPv4 for the sake of brevity. IPv6 addresses may be used instead without any other changes.

Wherever possible, to avoid confusion, each host's IPv4 addresses are of the form `10.0.<host>.<interface>`. These addresses are used because they exist in a range designated for private use [Moskowitz et al., 1996, Section 3]. The address used by the interface(s) representing a gate's input terminal(s) is `10.1.0.1`, and the address used by the interface representing the gate's output terminal is `10.2.0.1`.

5.1.2 Synthetic Directives

In reality, hosts implementing DNS store their settings in implementation-specific formats that may or may not be in the same files as their zone files. For the sake of terseness, we will include a directive in our zone files that specifies the addresses to which requests will be forwarded when they cannot be answered locally. These directives will be of the form `FWD ip-address`, and will always appear at the top of the zone file. These directives, unlike resource records, are ordered into a list when parsed by the host's DNS implementation.

5.1.3 Synchronization

In my construction, there are situations where we require some hosts to be delayed in their processing to allow synchronization between different hosts in a simulation of a logic gate. To accomplish this, we can add `FWD` directives that point to non-existent hosts to a zone file. Each such directive added to the start of a zone file will delay the host's response to the query it received by its configured connection timeout. This allows us to selectively increase the processing time required at any host in the network. By configuring the hosts in a gate using this method, we can uniformly increase the processing time required for a gate's output to be produced. In this manner, we may negate the effects of network delays by making them insignificant in comparison to the processing times.

We consider the depth of a gate within a circuit to be the number of gates between it and the circuit's input. We can set the timeouts for the hosts in a gate to be proportional to the depth of the gate within its circuit.

Combining these two controls, we can ignore the asynchronous nature of distributed hosts when designing and simulating our logic gates.

5.1.4 Limited Hierarchy

While it is possible to construct my gates in the full DNS hierarchy that exists on the Internet, it is simpler and clearer to use a reduced hierarcy. My reduced hierarchy uses names that have exactly three labels (including the zero-length root label). These names are of the form `<boolean value>.<tld>.`, with the boolean values being represented by the labels `f` (false) and `t` (true). The method of scaling up (i.e., more possibilities per labels) and scaling out (i.e., more labels per name) is a trivial expansion of the construction of the gates.

Several TLDs are necessary to control the routing of requests between hosts, and to allow requests to pass between gates. A `CNAME` resource record cannot have the same name as both an alias and a canonical name, requiring the use of TLDs that are internal to the different types of gates. For example, a zone file that tries to alias `umanitoba.ca.` to `umanitoba.ca.` creates a self-referencing loop. The `external.` TLD is used for requests travelling between gates. Because the output host of each gate emits, and the input hosts of each gate accept, requests using the `external.` TLD, connecting gates to form circuits is possible. The `internal.` TLD is used for requests within a gate exclusively to avoid situations that might otherwise require self-referential loops. Finally, the `cache.` TLD is used to detect when a request received by a host implementing the deduction template (see 5.3.6) was the result of both inputs to a gate having the same boolean value.

5.2 Processing Logic

When a DNS host receives a request, it attempts to respond with the information that request is seeking. A request asks for the resource record associated with a name (e.g., `umanitoba.ca.`) and a type (e.g., `A`). The host will first look locally in its zone files and cache for the requested information. If the host fails to find the information locally, and has been configured with forwarding directives, it will forward the request as-is to the first host in its list of forwarding directives. If the host does not receive a response from the host it forwarded the request to within a limited period of time, it will forward the request to the next host in its list of `FWD` directives and disregard future responses from the host associated with the previous `FWD` directive. If the host receives a response from the host to which it forwarded the request, it will accept the information. Using the new information, the host will determine whether it can now answer the initial request, and if so it will respond. If the new information indicates that additional information is required, it will continue searching for the information locally, and then via forwarding to find the answer. If the host is unable to acquire the information necessary to answer the initial request, it will respond indicating its failure.

The fallback logic that allows hosts to forward requests to multiple hosts is vital to the construction of the AND and OR gates I have designed. I use a cache miss, which appears to the requesting host as a timeout, to partly overcome the DNS protocol's limitation regarding evaluating previous or parallel inputs. Leveraging this fallback logic allows my gate to act differently depending on whether two inputs are the same (cache hit) or different (cache miss).

5.3 Zone File Templates

The zone files used by the hosts within our gates are based on templates, each of which provides a behaviour that can be understood in the abstract. Instead of using seconds as the unit of measurement for the TTL field of resource records, I put in a value T that may depend on the depth of the gate, as discussed in Section 5.1.3.

Comments, written as lines beginning with `#`, are included in some zone files for clarity.

5.3.1 Passthrough

Passthrough zone files exist in hosts whose primary responsibility is sending a similar request to the next host. Delay nodes are the most common users of this template. The Passthrough template is of the form:

```
FWD ip-address
```

5.3.2 Mirror

Mirror zone files respond to each requested name with the same Boolean value that was received, but a different TLD. The Mirror template is of the form:

```
f.domain-one. T CNAME f.domain-two.
t.domain-one. T CNAME t.domain-two.
```

Note that these zone files contain resource records that indicate they should be cached for T units of time, differentiating it from the Passthrough template (see 5.3.1).

5.3.3 Negative

Negative zone files associate each requested name with the negative of the Boolean value that was received, also with a different TLD. The Negative template is of the form:

```
f.domain-one. 0 CNAME t.domain-two. # NOT f = t
t.domain-one. 0 CNAME f.domain-two. # NOT t = f
```

5.3.4 Switching

Switching zone files ensure that the first request is always sent to one host, but a failure will result in a query to the second host. This behaviour supports the Deduction templates (see 5.3.6).

```
FWD ip-address-1
FWD ip-address-2
```

5.3.5 Translation

Translation zone files translate the `external.` TLD to the `internal.` TLD. This behaviour supports the Deduction template by ensuring that it never receives a request for a name under `external.`, since it must translate all names to that TLD and you may not have a `CNAME` with both the alias and canonical names having the same value.

```
f.external. 0 CNAME f.internal.
t.external. 0 CNAME t.internal.
```

5.3.6 Deduction

Deduction zone files represent the result of the binary operation performed on the two inputs accepted by the gate. These templates must handle each of the four possible combinations of inputs, representing each as a `CNAME` resource record. All `CNAME` resource records must use the `external.` TLD since they will be used in requests output from the gate.

Within these templates, the placeholder TLDs of `domain-one` and `domain-two` are used to encode information about the gate's inputs that was discovered previously in the network, specifically whether the two inputs were the same or different. Knowing whether the inputs were the same, represented by the TLD, and knowing the value of one of the two inputs, represented by the leftmost label, provides all the information necessary to generate the output corresponding to a Boolean operator.

AND Deduction

Like the truth table of an AND gate, the AND Deduction template will only cause its host to output a request for `t.external.` with one combination of inputs. See Chapter 6 for a full explanation of the behaviour of the AND gate.

```
f.domain-one. 0 CNAME f.external. # t AND f = f (inputs differ)
t.domain-one. 0 CNAME f.external. # f AND t = f (inputs differ)
f.domain-two. 0 CNAME f.external. # f AND f = f (inputs match)
t.domain-two. 0 CNAME t.external. # t AND t = t (inputs match)
```

By changing this template slightly, we could instead create a NAND gate. The NAND Deduction template would be:

```
f.domain-one. 0 CNAME t.external. # t NAND f = t (inputs differ)
t.domain-one. 0 CNAME t.external. # f NAND t = t (inputs differ)
f.domain-two. 0 CNAME t.external. # f NAND f = t (inputs match)
t.domain-two. 0 CNAME f.external. # t NAND t = f (inputs match)
```

OR Deduction

Like the truth table of an OR gate, the OR Deduction template will only cause its host to output a request for `f.external.` with one combination of inputs. See Chapter 6 for a full explanation of the behaviour of the OR gate.

```
f.domain-one. 0 CNAME t.external. # t OR f = t (inputs differ)
t.domain-one. 0 CNAME t.external. # f OR t = t (inputs differ)
f.domain-two. 0 CNAME f.external. # f OR f = f (inputs match)
t.domain-two. 0 CNAME t.external. # t OR t = t (inputs match)
```

By changing this template slightly, as we did to make the NAND gate in 5.3.6, we could instead create a NOR gate. The NOR Deduction template would be:

```
f.domain-one. 0 CNAME f.external. # t NOR f = f (inputs differ)
t.domain-one. 0 CNAME f.external. # f NOR t = f (inputs differ)
f.domain-two. 0 CNAME t.external. # f NOR f = t (inputs match)
t.domain-two. 0 CNAME f.external. # t NOR t = f (inputs match)
```

5.4 DNS Logic Gates

Logic gates are constructions that perform Boolean operations, transforming inputs to outputs in a consistent and stateless manner. We construct analogs of logic gates using networks of DNS hosts. All our DNS gates are designed so that the longest request path between any input and output host in the gate is less than or equal to seven, regardless of their inputs. All gates, regardless of type (i.e., NOT, AND, OR), will produce no more than seven requests between an input host and the output host. These constant maximum request path lengths allow us to disregard the internal behaviours of the gates, once we understand their operation, so that we may instead focus on their inputs and outputs.

Hosts named with the pattern D_i produce delays by implementing the Passthrough template. Hosts named with the pattern H_i implement other templates, producing more complex behaviour.

Gates have one or two inputs, named I for unary gates, and I_A and I_B for binary gates, respectively. The output of a gate is always named O. All of these inputs and outputs are hosts that are not part of the logic gate themselves, but are instead part of other gates. For example, the host in a NOT gate that receives input from I is named D_0 and the host that emits the result to O is named H_1, so when two NOT gates are connected the first gate's H_1 is connected to the second gate's D_0, substituting for I and O, respectively.

5.4.1 NOT Gate

A NOT gate takes a single input and outputs its inverse. Because the NOT gate evaluates a single input, its construction is much simpler than the other gates. Five delay hosts are used in this gate's construction, with each delay host introducing an additional request into the gate's longest request path. These extra requests in the NOT gate result in the length of the longest request path equalling the length of the longest request path in both the AND and the OR gates.

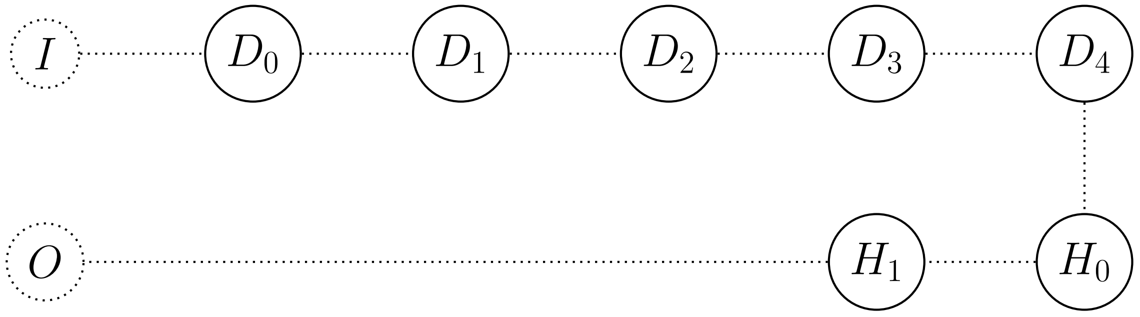

Figure 5.1: NOT Gate Schematic

Configuration

The majority of the hosts in this gate implement the Passthrough template and exist to ensure the gate has a request path distance equal to the binary gates, but for completeness we will include all of their configurations here.

Host D_0 D_0 uses the following addresses for its interfaces:

`10.1.0.1` connected to I

`10.0.1.2` connected to D_1

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.2.1 # to D1
```

Host D_1 D_1 uses the following addresses for its interfaces:

`10.0.2.1` connected to D_0

`10.0.2.2` connected to D_2

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.3.1 # to D2
```

Host D_2 D_2 uses the following addresses for its interfaces:

`10.0.3.1` connected to D_1

`10.0.3.2` connected to D_3

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.4.1 # to D3
```

Host D_3 D_3 uses the following addresses for its interfaces:

`10.0.4.1` connected to D_2

`10.0.4.2` connected to D_4

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.5.1 # to D4
```

Host D_4 D_4 uses the following addresses for its interfaces:

`10.0.5.1` connected to D_3

`10.0.5.2` connected to H_0

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.6.1 # to H0
```

Host H_0 H_0 uses the following addresses for its interfaces:

`10.0.6.1` connected to D_4

`10.0.6.2` connected to H_1

This host's zone file implements the Passthrough and Translation templates as follows:

```
FWD 10.0.6.1 # to H1

f.external. 0 CNAME f.internal.
t.external. 0 CNAME t.internal.
```

Host H_1 H_1 uses the following addresses for its interfaces:

`10.0.6.1` connected to H_0

`10.2.0.1` connected to O

This host's zone file implements the Negative template as follows:

```
FWD 10.1.0.1 # to O


f.internal. 0 CNAME t.external. # NOT f = t
t.internal. 0 CNAME f.external. # NOT t = f
```

5.4.2 AND Gate

An AND gate accepts two inputs and, if they are both true, emits a single output that is true. This simple operation is complicated by the stateless nature of DNS, combined with its inability to evaluate more than one value at a time. These problems can be overcome by delaying the evaluation of one of the two inputs, and using DNS's caching behaviour to infer the value of the input that was not delayed while evaluating the input that was.

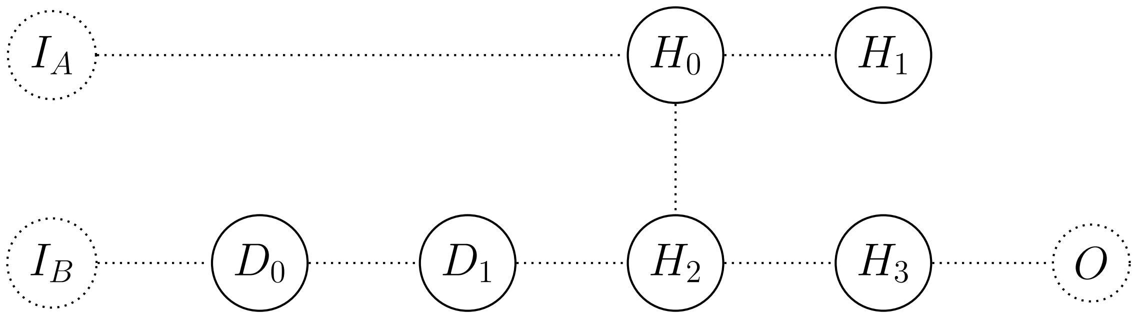

Figure 5.2: AND Gate Schematic

Configuration

Unlike the NOT gate, the AND gate uses every template previously discussed.

Host D_0 D_0 uses the following addresses for its interfaces:

`10.1.0.1` connected to I_B

`10.0.1.2` connected to D_1

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.2.1 # to D1
```

Host D_1 D_1 uses the following addresses for its interfaces:

`10.0.2.1` connected to D_0

`10.0.2.2` connected to H_2

This host's zone file implements the Passthrough template as follows:

```
FWD 10.0.5.1 # to H2
```

Host H_0 H_0 uses the following addresses for its interfaces:

`10.1.0.1` connected to I_A

`10.0.3.2` connected to H_1

`10.0.3.3` connected to H_2

This host's zone file implements the Passthrough and Translation templates as follows:

```
FWD 10.0.4.1 # to H1


f.external. 0 CNAME f.internal.
t.external. 0 CNAME t.internal.
```

Host H_1 H_1 uses the following addresses for its interfaces:

`10.0.4.1` connected to H_0

This host's zone file implements the Mirror template as follows:

```
f.internal. T CNAME f.cache.
t.internal. T CNAME t.cache.
```

The resource records in the above zone file will be cached for T, and indicate that they originated from this host.

Host H_2 H_2 uses the following addresses for its interfaces:

`10.0.5.1` connected to D_1

`10.0.5.2` connected to H_3

`10.0.5.3` connected to H_0

This host's zone file implements the Switching and Translation templates as follows:

```
FWD 10.0.3.3 # to H0
FWD 10.0.6.1 # to H3

f.external. 0 CNAME f.internal.
t.external. 0 CNAME t.internal.
```

Host H_3 H_3 uses the following addresses for its interfaces:

`10.0.6.1` connected to H_2

`10.2.0.1` connected to O

This host's zone file implements the AND Deduction template as follows:

```
FWD 10.1.0.1 # to O

f.internal. 0 CNAME f.external. # t AND f = f (inputs differ)
t.internal. 0 CNAME f.external. # f AND t = f (inputs differ)
f.cache.    0 CNAME f.external. # f AND f = f (inputs match)
t.cache.    0 CNAME t.external. # t AND t = t (inputs match)
```

5.4.3 OR Gate

An OR gate takes two inputs and, if either are true, produces a single output that is true. OR gates have the same schematic, shown again in Figure 5.3 for completeness, as AND gates. The differences between the two gates exist in the zone files, which govern the sequence of requests/responses the hosts perform.

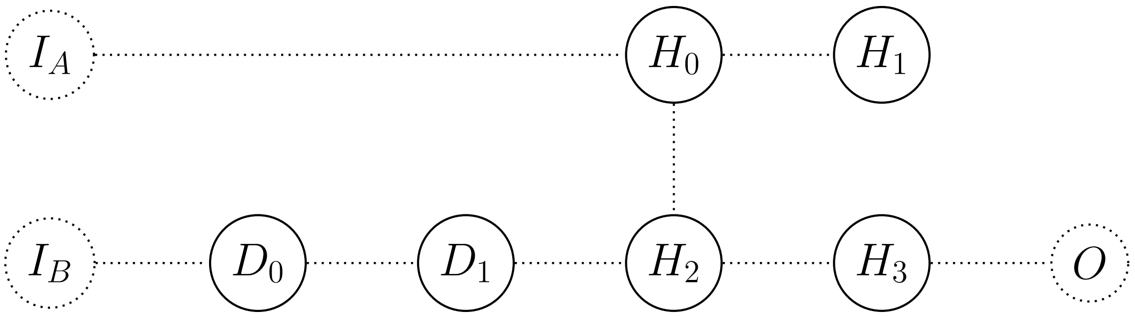

Figure 5.3: OR Gate Schematic

Configuration

All of the addresses and zone files used in the AND gate are the same for the OR gate, with the exception of the zone file for H_3.

Host H_3 This host's zone file implements the OR Deduction template as follows:

```
FWD 10.1.0.1 # to O


f.internal. 0 CNAME t.external. # t OR f = t (inputs differ)
t.internal. 0 CNAME t.external. # f OR t = t (inputs differ)
f.cache.    0 CNAME f.external. # f OR f = f (inputs match)
t.cache.    0 CNAME t.external. # t OR t = t (inputs match)
```

Chapter 6

Results

6.1 Overview

Given a network of hosts implementing DNS, and a set of queries sent simultaneously into the inputs of that network, how difficult is it to determine the result? In the previous chapter I described the behaviours, structures, and configurations used by my logic gates. In this chapter, I will walk through their operation, processing various inputs. By the chapter's end, I will have demonstrated that my gates can be used to build circuits so that my claim of a reduction of CSAT to DNS-SIM is understandable.

The figures in this chapter illustrate the operation of my DNS logic gates. Each arrow represents a message between hosts in the network. Single-tipped arrows indicate requests, while double-tipped arrows indicate responses. The labels near the arrows show the question (request) or partial answer (response) contained within the message. The caption below each figure indicates the step (S_n) it describes.

Since the NAND and NOR gates are trivial variations on the AND and OR gates, respectively, their function can be understood by looking at the gates I cover in this chapter, without a detailed walkthrough of their behaviour.

6.2 NOT Gate

6.2.1 Operation

While I will walk through the complete operation of the NOT gate here, as a stepping stone to help understand the AND and OR gates, I will only walk through a single input: `t.external.`. The sequence of requests within the gate does not change when `f.external.` is passed as input, only the values of the final two requests are inverted.

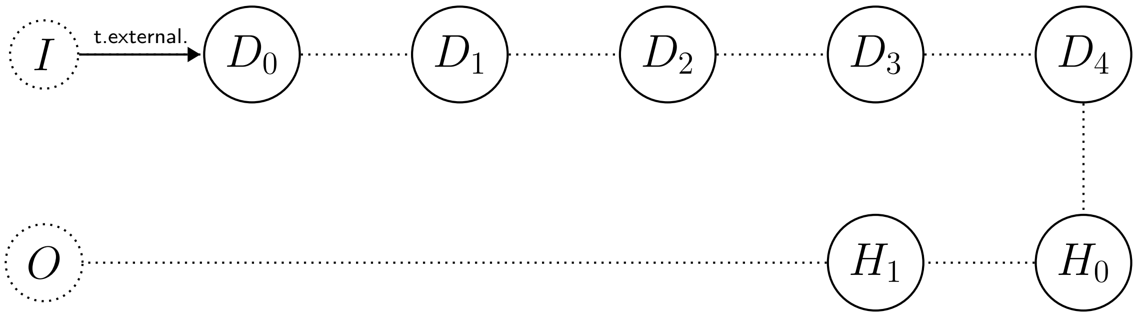

Figure 6.1: NOT t at S_0

At step S_0, the initial input is coming from outside the gate. This request will be forwarded through the remaining delay hosts through step S_5.

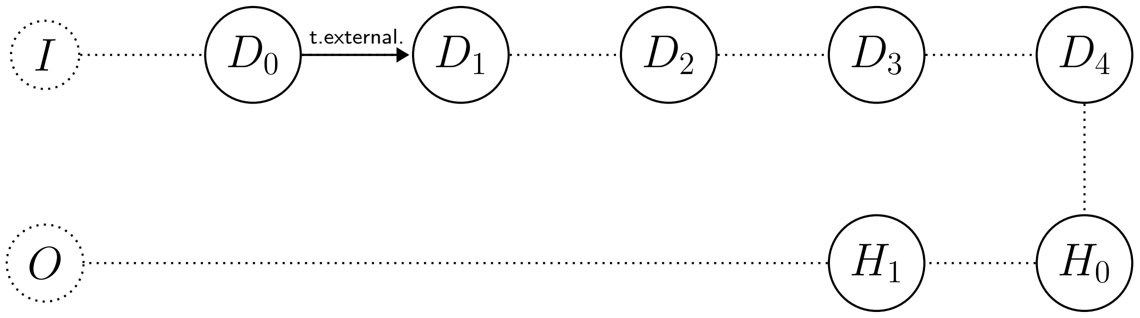

Figure 6.2: NOT t at S_1

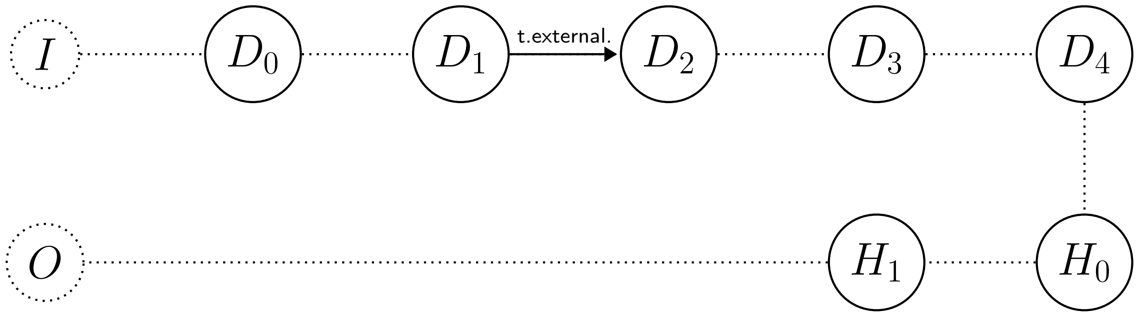

Figure 6.3: NOT t at S_2

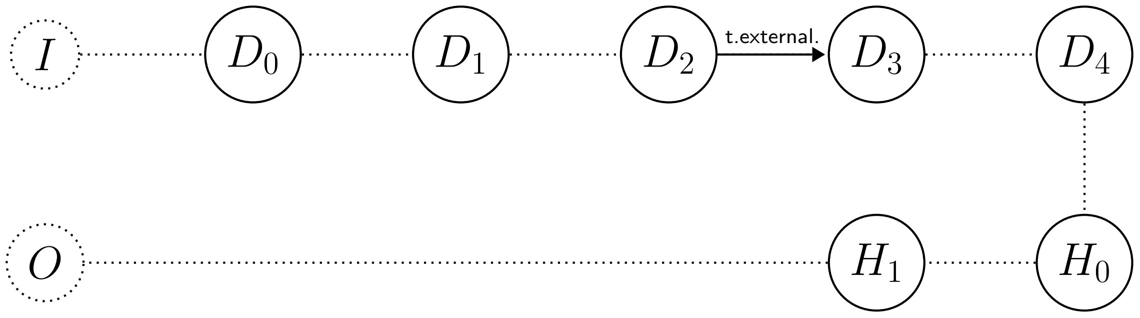

Figure 6.4: NOT t at S_3

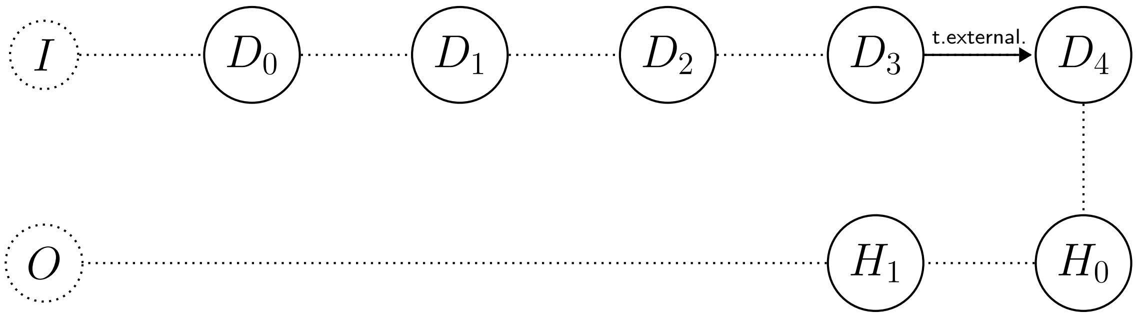

Figure 6.5: NOT t at S_4

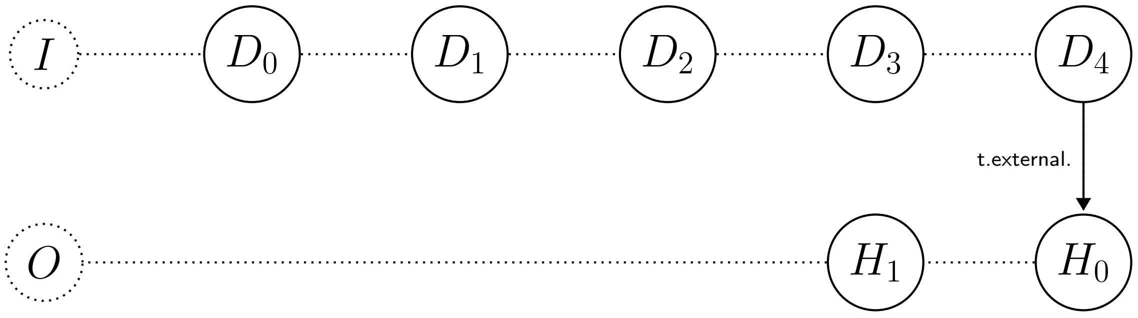

Figure 6.6: NOT t at S_5

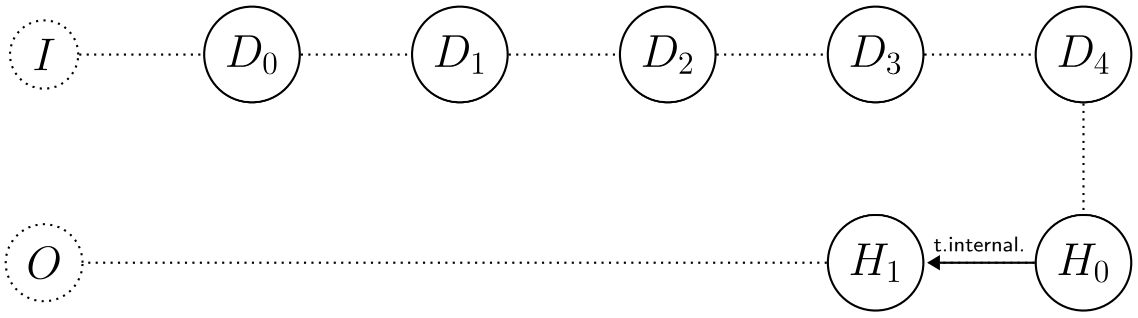

Figure 6.7: NOT t at S_6

At step S_6, the Translation zone file on H_0 has translated the request's TLD from `external.` to `internal.`. This allows H_1 to avoid having a zone file that associates between two Boolean values in the same TLD. DNS does not allow a `CNAME` to have the same value for both the alias and the canonical name, since that would create a self-contained loop of a name referencing itself. In the Mirror template, we use the same boolean value as the alias and canonical name, so we must use a Translation zone at the same time to ensure that the TLDs differ, preventing a loop.

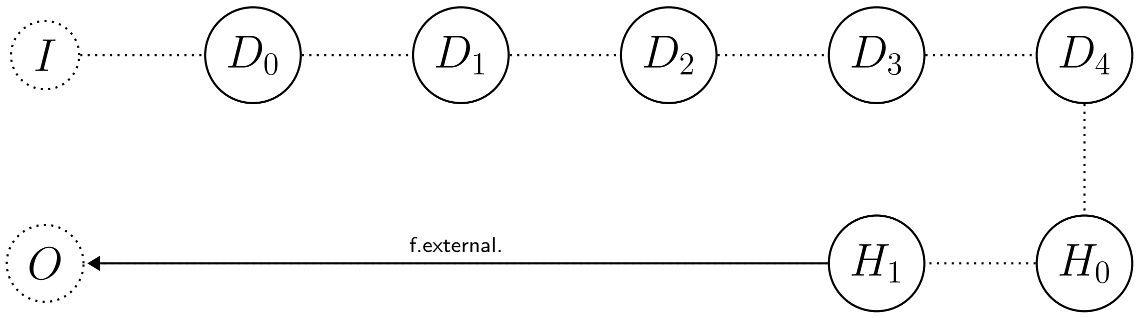

Figure 6.8: NOT t at S_7

At step S_7, the Negative zone file on H_1 inverts the value of the request, and emits a new request using the standard inter-gate addresses and TLD.

6.3 AND Gate

6.3.1 Operation

To understand the operation of my AND gate, it is useful to group the four possible input permutations into two situations: inputs that match (`f` AND `f`, `t` AND `t`) and inputs that differ (`f` AND `t`, `t` AND `f`).

Matching Inputs

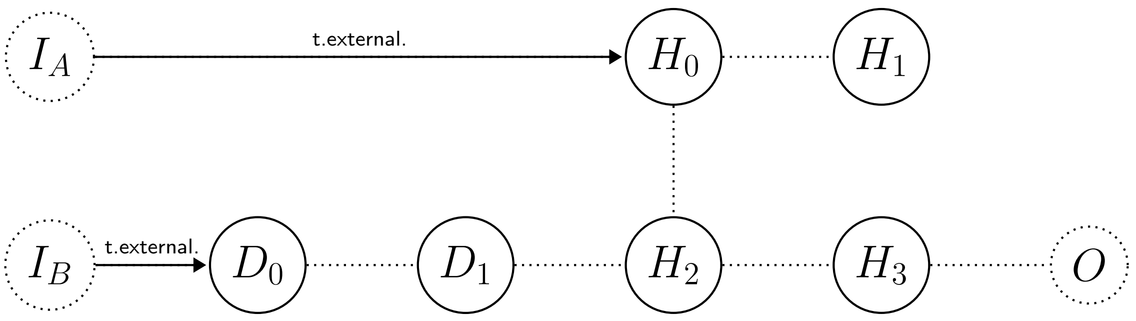

Figure 6.9: t AND t at S_0

At step S_0, both inputs, implemented as DNS queries, enter the gate's network at the same time from external sources. The input from I_B will be sent through hosts that serve only to lengthen the number of steps in its journey through the gate. These delay hosts perform an identity function, not altering the input. These delays are implemented by the hosts using the Passthrough template, which ensures that no request can be answered by the data on the host, since there is none. To answer any request, a delay host must forward the query to the first host configured in the nameserver's list of forwarding hosts for the requested TLD.

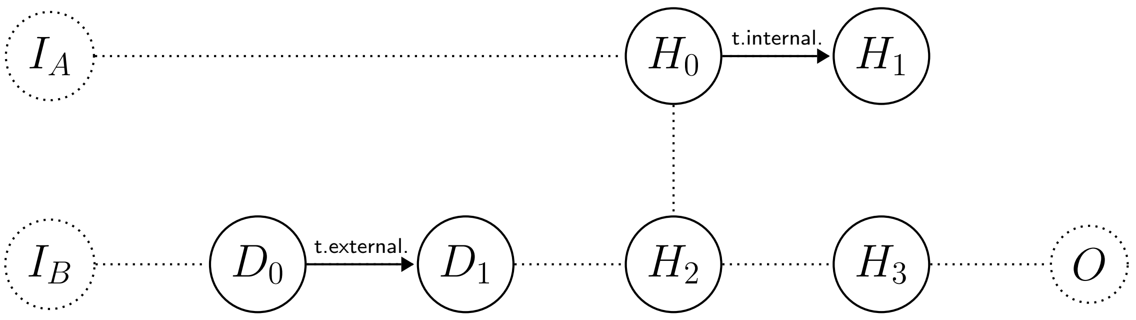

Figure 6.10: t AND t at S_1

At step S_1, H_0, having received a request on the previous step, will now try to answer it. H_0 has an empty cache, and no zone files that contain the answer. Since it can't answer the question itself, lacks any resource records stored locally, and it has not yet found the `A` record it seeks, H_0 will forward the request to H_1 as indicated by the `FWD` directive.

The second input continues to be delayed.

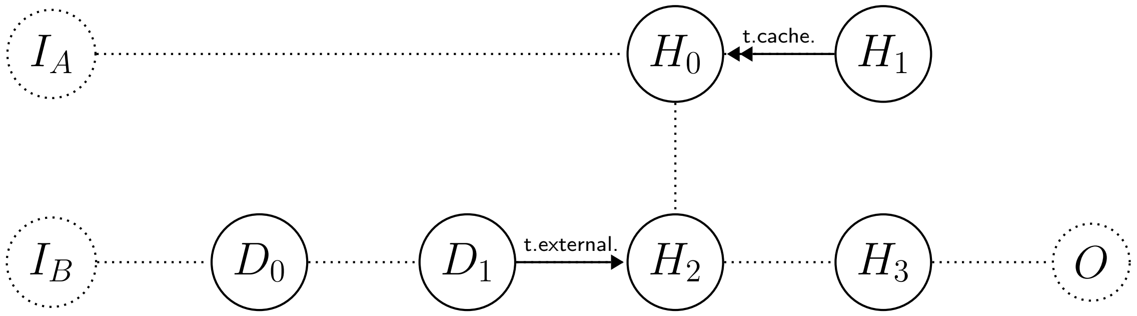

Figure 6.11: t AND t at S_2

At step S_2, H_1 processes the forwarded request for the `A` record of `t.external.`, and finds that it does not have an `A` resource record, but it does have a corresponding `CNAME` resource record. H_1 responds to H_0 with the value contained in the `CNAME` resource record: `t.cache.`. If the query was for `f.external.` instead, the result would have been `f.cache.`, since the zone file implements the Mirror template.

The second input exits the last delay host on this step, headed toward a host that will actually put it to use.

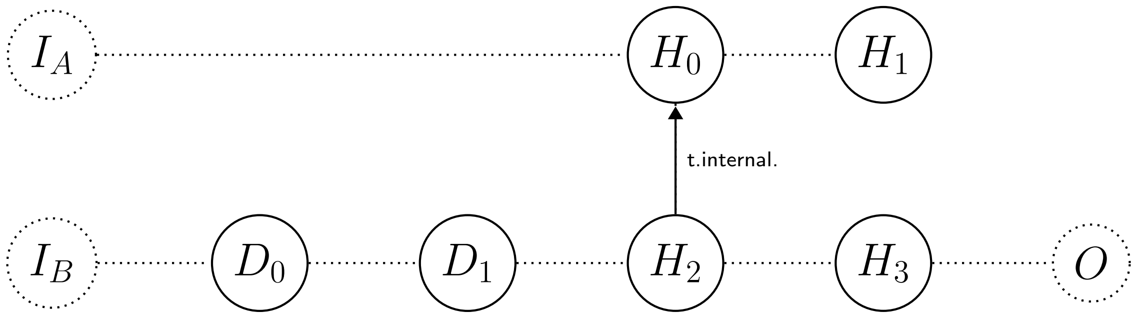

Figure 6.12: t AND t at S_3

At step S_3, H_0 receives the response from H_1. Seeing that the resource record in the response has a TTL greater than zero, H_0 caches the `CNAME` record mapping of `t.external.` to `t.cache.`, which makes it available to requests received during the next step. H_0 will then try to resolve `t.cache.`, but will fail and the remainder of its actions have no effect on the rest of the logic. H_0 will send a second request to H_1 for the address of `t.cache.`, which will generate a response indicating failure. H_0 will then respond to I_A, informing it of the mapping from `t.external.` to `t.cache.`. I_A will ignore the response, because it came after it had stopped waiting for the response. These requests and responses are not pictured because hosts can process several requests in parallel, and including them would complicate the figures. Unused requests and responses will not be mentioned again.

Meanwhile, H_2 receives the request from D_1. H_2 has a Translation zone file, so it translates the request from `t.external.` to `t.internal.`. Having no cache or zone files with which to answer the request, H_2 forwards the request to the first host in its set of `FWD` directives: H_0.

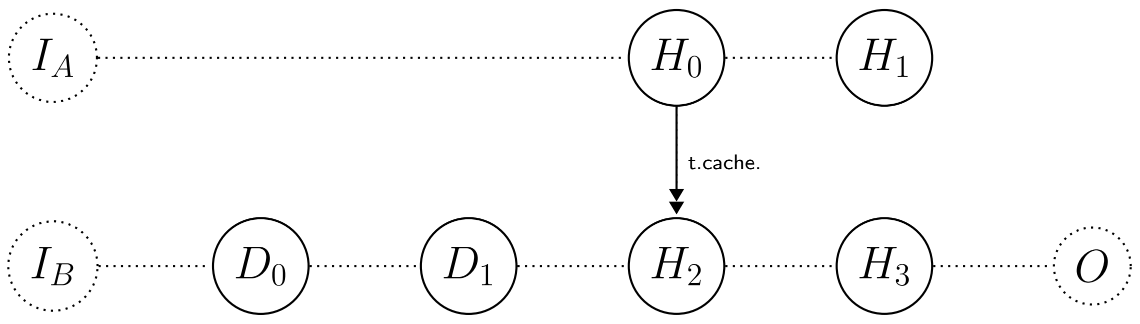

Figure 6.13: t AND t at S_4

At step S_4, H_0 receives the request from H_2 and, unlike when processing the input from I_A, can answer this query from the cache since both the requests asked for the same name. H_0 responds with `t.cache.` and removes the entry from its cache since it expires at the end of this step.

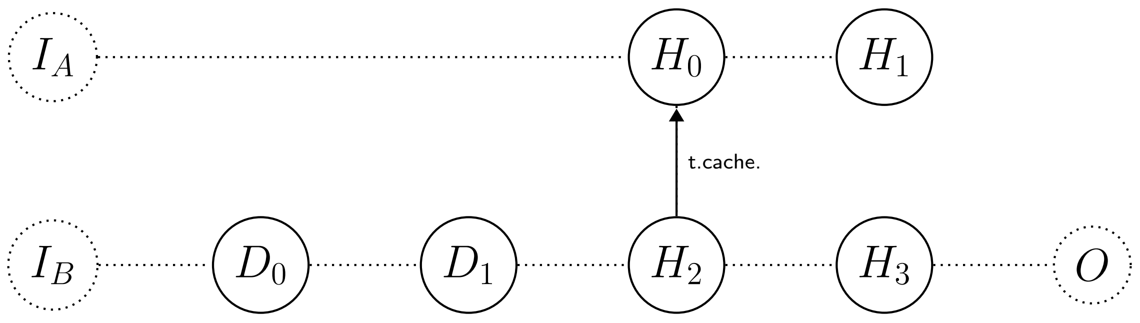

Figure 6.14: t AND t at S_5

At step S_5, H_2 receives the response from H_0, but still does not have the `A` record it's looking for. H_2 sends a request to H_0 for `t.cache.`, again following the first `FWD` directive. [1]

[1] H_2 must always forward requests to H_0 first. I could include an advanced feature called conditional forwarding that introduces directives used by the forwarding algorithm that map TLDs to hosts. Not including conditional forwarding adds one step, requiring the gate to waste this request and fall back to the secondary forwarding host, but keeps my design simpler.

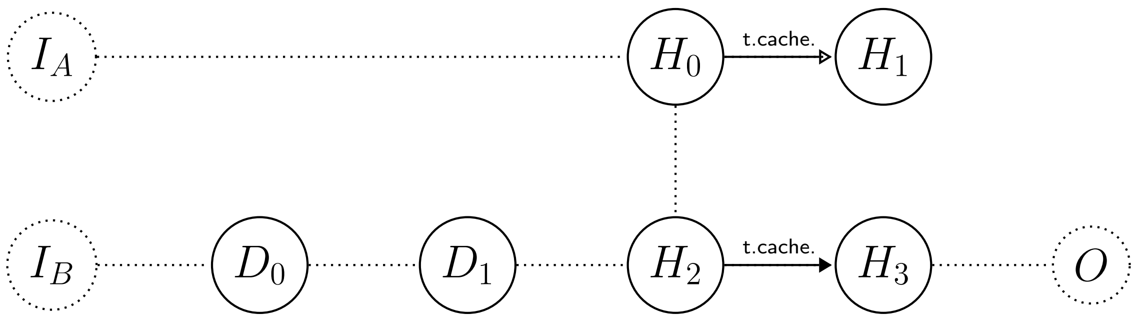

Figure 6.15: t AND t at S_6

At step S_6, H_0 receives the request seeking `t.cache.`, but does not have any information on that name. Not only has the cached information in H_0 expired, but that information only had `t.cache.` as a canonical name (right-hand side of zone file), not as a lookup key (left side of zone file). H_0 must therefore forward a request to H_1 for the information, but this request is not shown since its results will not be used by this gate.

Since H_2 has not received a response from H_0 within its timeout window, it forwards the request to the next host in its set of forwarding directives: H_3.

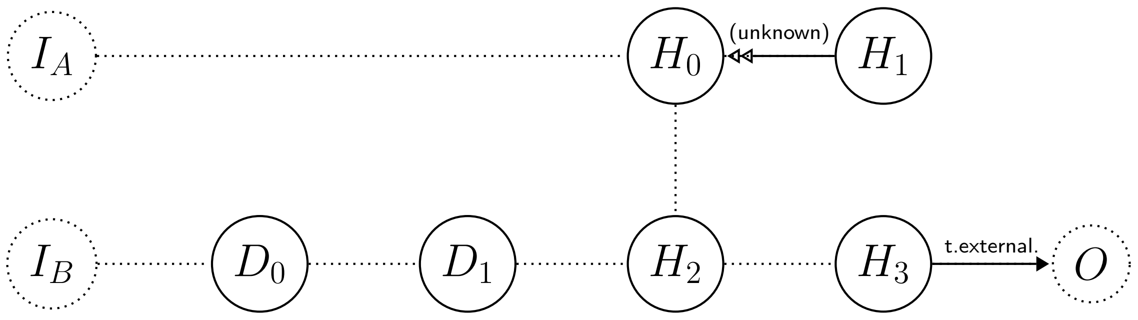

Figure 6.16: t AND t at S_7

At step S_7, H_3 receives the request, and fails to find the requested `A` record, but finds a corresponding `CNAME` record. This `CNAME` record's value, its canonical name, represents the Boolean output of this gate: true (`t.external.`). The zone file on H_3 causes the host to send a request to resolve the `A` record associated with `t.external` to the host connected to the AND gate's output terminal.

Differing Inputs

When processing differing inputs, the pattern of signals (if not the values) are the same from S_0 through S_3, and so will be shown without repeating the previous explanations.

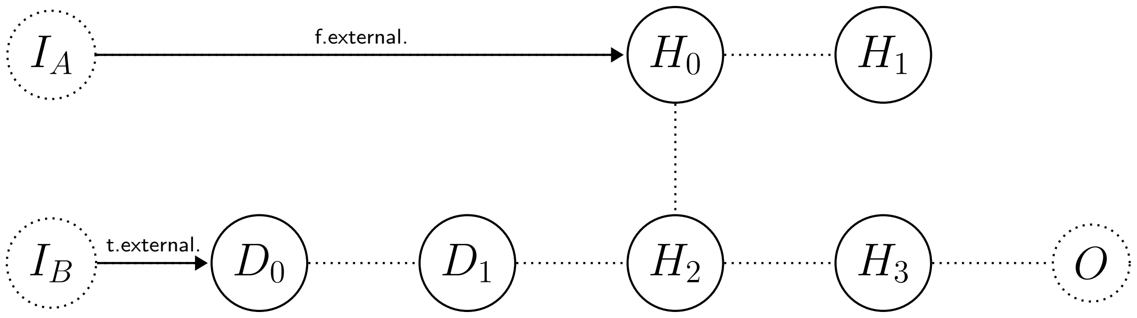

Figure 6.17: f AND t at S_0

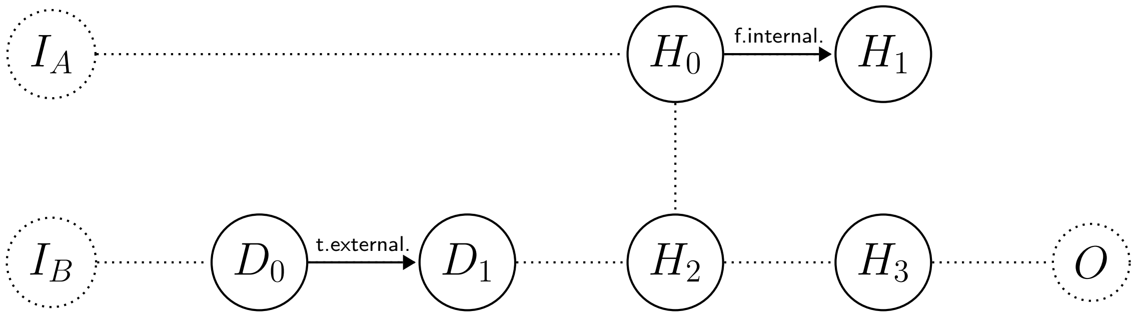

Figure 6.18: f AND t at S_1

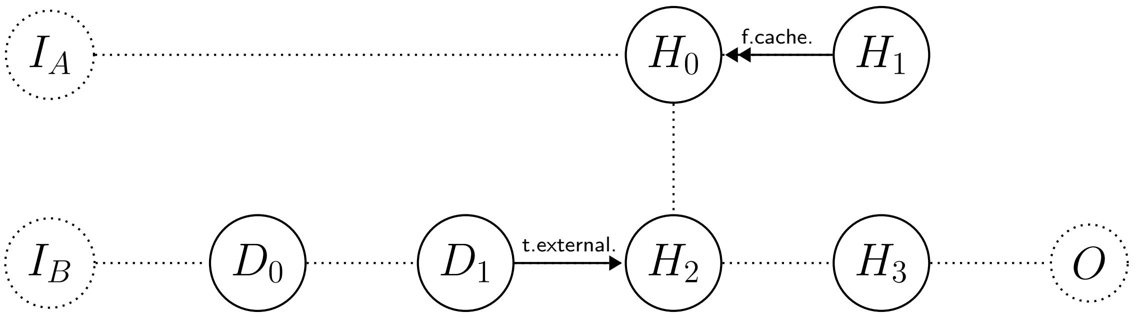

Figure 6.19: f AND t at S_2

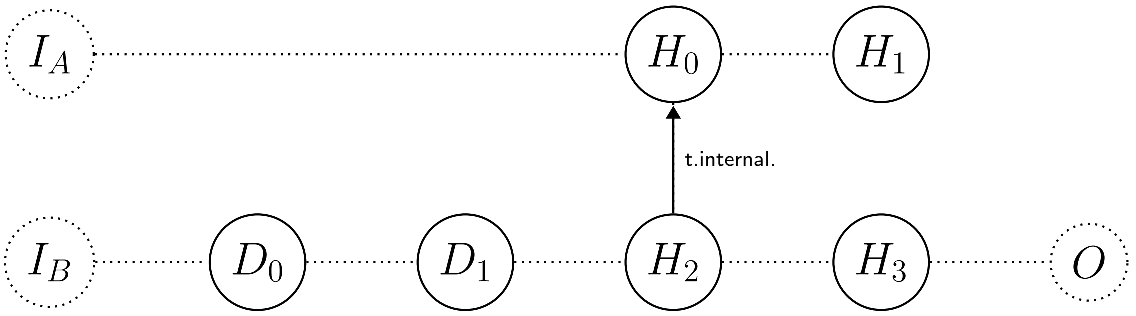

Figure 6.20: f AND t at S_3

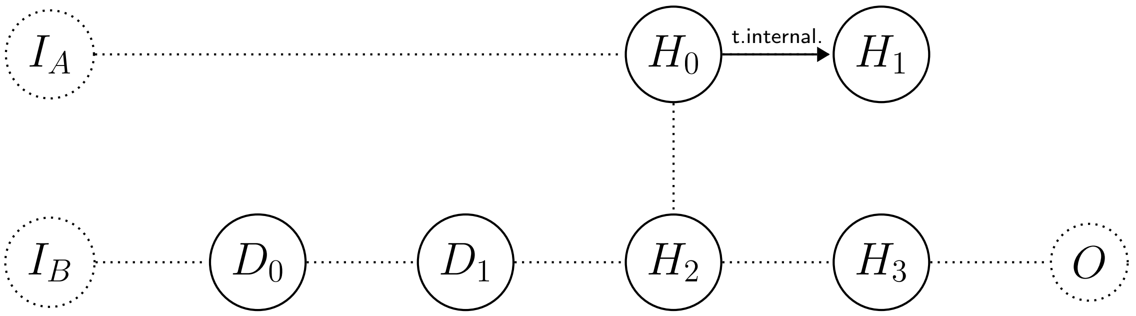

Figure 6.21: f AND t at S_4

At step S_4, the first difference between matching and differing inputs occurs: the request from H_2 to H_0 cannot be answered locally by H_0 in the case of differing inputs. H_0, finding itself unable to answer the question it has received, forwards the question to H_1 similar to its action at S_1.

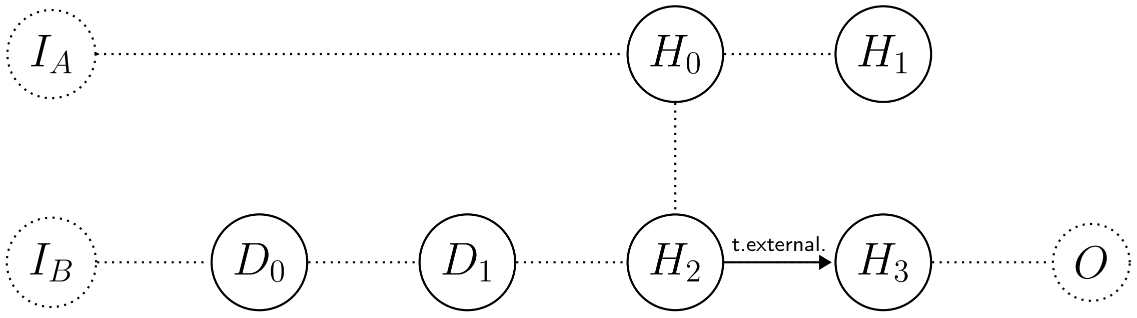

Figure 6.22: f AND t at S_5

At step S_5, after querying H_0, H_2 moves on to query H_3. The query that it will send to H_3 uses the `CNAME` that was returned from the cache of H_0.

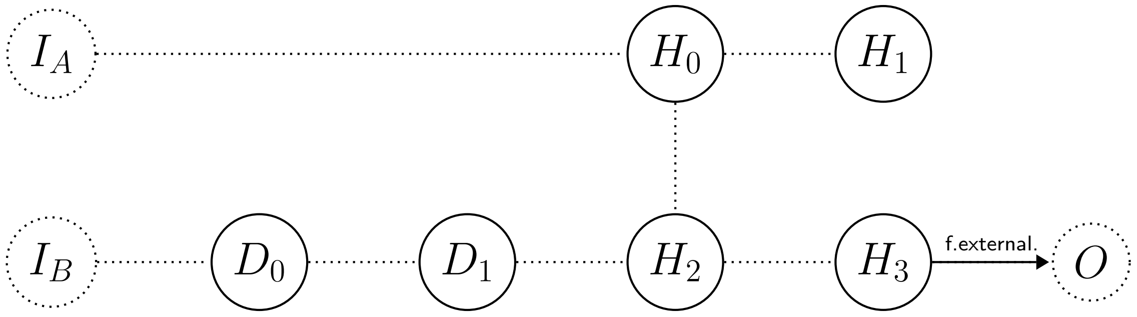

Figure 6.23: f AND t at S_6

At step S_6, H_3 receives the query and notes that the rightmost label is `external.`, from which it can infer that the first input was different from the second input (which is represented by the leftmost label). H_3 then chooses the final Boolean result of the gate, `f`, and attaches the inter-gate label of `external.` to make the gate's output suitable for input to any other gate.

6.4 OR Gate

6.4.1 Operation

The operation of the OR gate matches the AND gate except at S_6, where the zone file of H_3 results in requests being sent to O that correspond to the output of an OR gate rather than an AND gate.

6.5 DNS-SIM

Given a network of hosts implementing DNS, I consider some hosts to be inputs to the network, and one host to be the output of the network. An assignment of inputs to the network is a mapping of simultaneously-issued DNS requests to the network's inputs. I define DNS-SIM as the decision problem of determining whether there is any assignment of inputs that result in a request for `t.external.` being output from the network.

I have reduced CSAT to DNS-SIM. CSAT is known to be NP-complete, and therefore DNS-SIM is NP-complete.

Chapter 7

Conclusions

7.1 Future Work

While we have shown that networks of DNS hosts can implement logic gates, future research could try to produce constructions that do not rely on forwarding. Removing the construction's reliance on forwarding would be the logical next step, since it is the most advanced feature in use by the design. To remove forwarding, the construction would have to be rooted at a single host sending iterative requests to all other hosts, creating a hub-and-spoke organization. With all requests being issued from a single host, it might be possible to use that host's cache as the tape of a Turing machine. With the inclusion of memory, it may be possible to design a construction that directly emulates a Turing machine.

Another interesting construction would be a latch or flip-flop. While such a construction would likely require the use of forwarding, it may be possible to use a node's cache as the current state (i.e., memory) of the construction. With latches or flip-flops we could construct more useful circuits.

There is another promising resource record type, `NAPTR` [Mealling, 2002], which may provide additional computational power. The `NAPTR` record allows incoming queries to be rewritten according to a regular expression. We could attempt to implement Binary Cyclic Tag (BCT) programs using `NAPTR` records. A zero command, indicating the left-most bit should be deleted, could be implemented with the regex `"!^.(.*)$!\1!"`. A one command, operating as a single-bit append operation of a bit I'll call 'x', could be implemented as `"!^(.*)$!\1x!"`. If you relaxed DNS's length restrictions on names, and used `NAPTR` records to encode the BCT program, it might be possible to implement a BCT interpreter into a network of hosts running DNS. Implementing SKI combinator calculus may also be possible using similar methods.

7.2 Fundamental Result

DNS has a number of safety mechanisms within it to limit the amount of work performed in the service of a single request, but when DNS hosts are configured to use forwarding and `CNAME` resource records exist that require many requests to resolve, it is possible to construct Boolean circuits. While my constructions have focused on circuits with one or two inputs, there is nothing preventing constructing families of circuits that perform Boolean operations on any positive number of inputs.

Having defined DNS-SIM above, and proven by reduction that it is NP-complete, I can now say that the task of evaluating the behaviour of a network of hosts implementing DNS is NP-complete.

Bibliography

Baker, F. (2009). Core protocols in the internet protocol suite. draft-baker-ietf-core-04.

Bangert, J., Bratus, S., Shapiro, R., and Smith, S. W. (2013). The page-fault weird machine: Lessons in instruction-less computation. `https://www.usenix.org/conference/woot13/workshop-program/presentation/Bangert`.

Bradner, S. O. (1996). The Internet Standards Process – Revision 3. RFC 2026.

Bratus, S., Bangert, J., Gabrovsky, A., Shubina, A., Bilar, D., and Locasto, M. E. (2012). Composition patterns of hacking. *Cyberpatterns 2012*, pages 80–85.

Bratus, S., Darley, T., Locasto, M., Patterson, M. L., Shapiro, R., and Shubina, A. (2014). Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *Security & Privacy, IEEE*, 12(1):83–87.

Chiesa, M., Cittadini, L., Di Battista, G., Vanbever, L., and Vissicchio, S. (2013). Using routers to build logic circuits: How powerful is BGP? In *21st IEEE International Conference on Network Protocols*, pages 1–10.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Deering, D. S. E. and Hinden, R. M. (2017). Internet Protocol, Version 6 (IPv6) Specification. RFC 8200.

Elz, R. and Bush, R. (1997). Clarifications to the DNS Specification. RFC 2181.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Kaminsky, D., Patterson, M. L., and Sassaman, L. (2010). PKI layer cake: New collision attacks against the global X.509 infrastructure. In *Financial Cryptography and Data Security*, pages 289–303. Springer.

Ksinant, V., Huitema, C., Thomson, D. S., and Souissi, M. (2003). DNS Extensions to Support IP Version 6. RFC 3596.

Mealling, M. H. (2002). Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database. RFC 3403.

Moskowitz, R. G., Karrenberg, D., Rekhter, Y., Lear, E., and de Groot, G. J. (1996). Address Allocation for Private Internets. RFC 1918.

Müller, U. (1993). Brainfuck–an eight-instruction Turing-complete programming language. `http://en.wikipedia.org/wiki/Brainfuck`. Accessed: 2019-05-29.

Oakley, J. and Bratus, S. (2011). Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code. In *5th USENIX Workshop on Offensive Technologies*, pages 91–102, Berkeley, CA. USENIX.

Palmer, C. (2016). Words matter: Language in security engineering. `https://noncombatant.org/security-seminar/`. Accessed: 2016-03-19.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

RFC (1980). User Datagram Protocol. RFC 768.

RFC (1981). Internet Protocol. RFC 791.

RFC (1987a). Domain names - concepts and facilities. RFC 1034.

RFC (1987b). Domain names - implementation and specification. RFC 1035.

Sassaman, L., Patterson, M. L., Bratus, S., and Shubina, A. (2011). The halting problems of network stack insecurity. *;login: The USENIX Magazine*, 36(6).

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA. ACM.

Shapiro, R., Bratus, S., and Smith, S. W. (2013). "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. `https://www.usenix.org/conference/woot13/workshop-program/presentation/shapiro`.

Tan, S. J., Bratus, S., and Goodspeed, T. (2014). Interrupt-oriented bugdoor programming: A minimalist approach to bugdooring embedded systems firmware. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 116–125. ACM.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265.

Veldhuizen, T. L. (2003). C++ templates are Turing complete. `http://citeseer.ist.psu.edu/581150.html`.