Learning Regexes to Extract Router Names from Hostnames

Matthew Luckie
University of Waikato
mjl@wand.net.nz

Bradley Huffaker
CAIDA, UC San Diego
bradley@caida.org

k claffy
CAIDA, UC San Diego
kc@caida.org

ABSTRACT

We present the design, implementation, evaluation, and validation of a system that automatically learns to extract router names (router identifiers) from hostnames stored by network operators in different DNS zones, which we represent by regular expressions (regexes). Our supervised-learning approach evaluates automatically generated candidate regexes against sets of hostnames for IP addresses that other alias resolution techniques previously inferred to identify interfaces on the same router. Conceptually, if three conditions hold: (1) a regex extracts the same value from a set of hostnames associated with IP addresses on the same router; (2) the value is unique to that router; and (3) the regex extracts names for multiple routers in the suffix, then we conclude the regex accurately represents the naming convention for the suffix.

We train our system using router aliases inferred from active probing to learn regexes for 2550 different suffixes. We then demonstrate the utility of this system by using the regexes to find 105% additional aliases for these suffixes. Regexes inferred in IPv4 perfectly predict aliases for ≈85% of suffixes with IPv6 aliases, i.e., IPv4 and IPv6 addresses representing the same underlying router, and find 9.0 times more routers in IPv6 than found by prior techniques.

CCS CONCEPTS

• **Information systems → Clustering and classification**; • **Networks → Naming and addressing**.

KEYWORDS

Regular expression learning, Internet topology, alias resolution

ACM Reference Format:
Matthew Luckie, Bradley Huffaker, and k claffy. 2019. Learning Regexes to Extract Router Names from Hostnames. In *IMC '19: ACM Internet Measurement Conference, October 21–23, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3355369.3355589

1 INTRODUCTION

Internet (IP) address alias resolution is a critical step in transforming an interface-level graph captured by traceroutes into a router-level graph that reflects the underlying topology. Alias resolution techniques that use packet-probing rely on artifacts of router implementations to infer if a set of interface IP addresses belong to the same router. For example, Mercator [8] infers two addresses belong to the same router if the source address of ICMP port unreachable responses is the same, and Ally [27], RadarGun [4], MIDAR [12], and Speedtrap [16] all infer two addresses belong to the same router if the IP-ID values in response packets appear to be derived from a central counter. However, because packet-probing techniques depend on specific router behaviors, and on operators not configuring their network to block or ignore the packets, the set of aliases that a single technique can infer is limited even inside a single network.

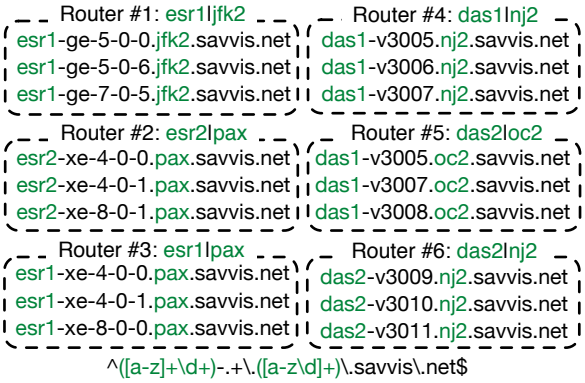

Figure 1: A regex that extracts unique router names for Savvis routers. Each router name is used consistently among interfaces on the same router, and is not found in hostnames on other Savvis routers. §2.1 summarizes regex syntax.

Researchers and network operators have used information encoded in the Domain Name System (DNS) to understand properties of the network for at least 20 years. To aid network management, operators often use DNS hostname strings to encode information about the name of the router, location, role, or interconnection properties of router interfaces – the hardware components that connect to other routers. However, operators have never developed consensus on a universal naming convention – each networked organization independently selects such conventions for their own suffix (e.g., savvis.net). The resulting diversity in conventions prevents researchers and network operators from systematically using information encoded in these hostnames.

Researchers have traditionally manually derived regular expressions (regexes) from apparent router naming conventions to extract network topology information, such as the geographic placement and roles of routers, link speeds, and router names (e.g., [6, 7, 27]). In this paper, we consider the challenge of automatically learning if an operator uses a convention within a suffix that includes a *router name* – i.e., a unique router identifier – by evaluating automatically generated regexes. Our system supervises the learning process using training data comprising a set of router interface IP

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IMC '19, October 21–23, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6948-0/19/10…$15.00
https://doi.org/10.1145/3355369.3355589

addresses inferred to be aliases by another alias resolution technique. Conceptually, if a regex extracts the same name from a set of hostnames associated with the same router in our training data, and a unique name for all routers in the training data within the same suffix, then we conclude the regex captures the convention for storing router names in that suffix. Figure 1 provides examples of hostnames containing router names assigned by operators for Savvis, and a regex that captures those router names.

There are three key challenges. First, we do not know if hostnames within a given suffix have any convention for embedding router names to begin with; if there is a convention, we do not know the specific regular expression components required to extract it, leading to a search space that is infeasible to learn through brute force. Second, network operators may not keep hostnames in their suffix current or free of errors [29]; in these cases, even if an operator has a convention, stale hostnames could lead to an incorrect regex, or the regex could lead to incorrect inferences. Third, we rely on imperfect router training data, as alias resolution techniques are heuristic-based and only feasible for a subset of the Internet topology, and some aliases are incorrect (false positives), or absent (false negatives). In light of these challenges, this paper makes the following four contributions.

(1) We introduce a scalable method for accurately inferring regexes that extract router names from hostnames. Because it is not feasible to learn conventions with brute force, we built a method that, over the course of eight stages, finds general patterns in hostnames, learns any necessary literals and character classes to embed in the regexes, assembles conventions from regexes, and learns regexes that filter out hostnames with no router name component. The method is implemented in C, builds and evaluates regexes using parallel threads of execution, and uses compilation extensions in regex libraries that reduce runtime.

(2) We validate our algorithm using ground truth from 10 network operators. We built a public website containing the regexes that form our inferred conventions, as well as a per-suffix demonstration showing the outcome of applying those regexes to router interface hostnames in our training data. We sent a link to the website to the North American Network Operators' Group (NANOG) mailing list in April 2019. We received validation data covering 11 networks of different classes and scale from 10 operators, from a Tier-1 network and a large U.S. content provider, to smaller access networks. The responses show that our inferred conventions capture the operators' naming intent, though in two cases the conventions could have been improved with better training data. All 10 operators manually maintained their hostnames.

(3) We demonstrate the utility of our algorithm by applying it to 16 sets of training data across 9 years. We used the 16 Internet Topology Data Kit (ITDK [5]) snapshots built by CAIDA between July 2010 and April 2019, which include routers inferred using the MIDAR [12] and Mercator techniques [8], and associated hostname strings, to automatically derive naming conventions for 2550 suffixes. The conventions inferred additional aliases for 19,136 routers in 619 suffixes for the 201904 ITDK, a 105% gain. Conventions we inferred for the IPv4 topology perfectly predicted IPv6 clustering for ≈85% of overlapping suffixes, implying our conventions infer IPv4 and IPv6 router aliases, a step towards analyzing router-level congruity of IPv4 and IPv6 paths.

Term	Definition
hostname	A string stored in a DNS pointer (PTR) record for an IP address.
suffix	A label sequence at the end of a hostname identifying an administrative domain.
extractor regex	A regex that extracts a possible router name from a hostname.
filter regex	A regex that matches but does not extract a router name from a hostname.
training router	A router where prior alias resolution techniques found aliases.
training set	A set of training routers belonging to a suffix we use to infer a naming convention.
application set	A set of router interfaces where prior alias resolution techniques did not find aliases.
router name	A string common to interfaces of a router, different from other routers in the suffix.
candidate name	The longest common substring across hostnames for a training router.
extracted name	The string extracted from a hostname by a candidate regex.
naming convention	A set of filter and extractor regexes that capture the way operators embed router names in hostnames for a suffix.

Table 1: Definitions that we use in this work.

(4) We publicly release the source code implementation and a website containing the inferred naming conventions. We name our tool Hoiho, for Holistic Orthography of Internet Hostname Observations, after a flightless native New Zealand bird [22]. To promote further validation and use of Hoiho, we publicly release our source code implementation as part of scamper [15]. The website we built for validation containing the regexes and their application [17] allows researchers to obtain the regexes, understand how they work, and potential limitations given incongruities between the training data and our conventions.

We provide background in §2, discuss challenges in §3, and identify principles that address tensions in the algorithm in §4. §5 describes our algorithm, while §6 presents limitations of the approach. Finally, §7 shows potential applications of our algorithm, and §8 outlines future work. Table 1 summarizes the definitions we use in this work.

2 BACKGROUND AND RELATED WORK

2.1 Regular Expressions: Crash Course

A regex defines a pattern that can be applied to a string to check if the string conforms to the structure expressed in the pattern. The regex ^[a-z]+\.foo\.com$ applied to bar.foo.com would match, because bar consists solely of letters between a and z, and the remainder of the string is .foo.com. This work uses the regex syntax capabilities provided by the Perl Compatible Regular Expressions (PCRE) library [9]. This section covers the small portion of PCRE syntax that we use.

Patterns within a regex may be expressed as literals (e.g., foo) or as character classes. .+ matches any sequence of characters. \d* matches zero or more digits, \d+ matches at least one digit, \d matches one digit, and \d{4} matches exactly four digits. [a-z]+ matches at least one alphabetic character, [a-z\d]+ matches alphanumeric characters, and [a-z]+\d+ matches a sequence of alphabetic characters followed by a sequence of digits. Patterns may specify what they cannot contain. [^-]+ matches a sequence of characters that does not contain a hyphen; foo matches but foo-bar does not.

A regex may be anchored so that the pattern expressed must begin at the start of a string with ^ and/or end at the last character of a string with $. All of our regexes use anchors at the end of the string, as the suffix to which they apply is at the end of the string. A regex may extract portions of a string by including the portion of interest in parentheses. The regex ^([a-z]+)\.foo\.com$ extracts bar from bar.foo.com. Our regexes use parentheses to extract portions of the hostname that could contain a router name. Some characters in a regex must be escaped with a backslash (\) to match the character, rather than be interpreted as a control sequence. Ordinarily, a dot (.) matches any character; \. matches a dot. Finally, a regex may contain a logical-or statement that matches one of a series of possible patterns. The pattern (?:foo\d+|bar\d*|baz) will match either (1) foo followed by at least one digit, (2) bar followed by digits, if any are present, or (3) baz.

2.2 Grammar Induction

Learning structure from example text is known as *grammar induction* in machine learning. Methods in the literature range in complexity from trial and error approaches like the one we describe in §5, to genetic algorithms to address more complex examples. We chose a heuristic-guided trial and error approach, as the implementation of the algorithm is simple to explain and understand, we can make use of domain knowledge to constrain the set of candidates, and the execution time of the algorithm is reasonable because the set of regexes we evaluate for each suffix is relatively small. Grammar induction methods usually produce a parse tree to represent valid grammatical constructs, but our method produces practical regexes that researchers can use to analyze Internet topology.

In 2008, Li *et al.* built ReLIE to reduce the manual effort in building a regex [14]. The approach relied on a human providing a starting regex and input data, which their method would then improve. In 2010, Babbar *et al.* [2] introduced a technique that could learn regexes even when a human with lower domain expertise than assumed in [14] provided the starting regex. In 2012, Murthy *et al.* [21] presented a technique to improve recall of regexes that involved human feedback. All techniques were able to improve input regexes for identifying patterns such as software names, phone numbers, and university course numbers. In 2016, Bartoli *et al.* built RegexGenerator, which instead relied on a human to provide examples of valid extractions from a set of input data, for which their method would then build a regex [3]. In our work, we do not have a set of starting regexes or valid extractions for each suffix to learn from, so we assess the extractions we make through trial-and-error for correctness against an input set of router aliases.

2.3 Extracting Information from DNS

Researchers have used information encoded in DNS to understand router-level properties of the Internet for at least 20 years. To identify the hostnames corresponding to routers, researchers query the DNS for pointer (PTR) records for router interface IP addresses observed by traceroute in a path toward a destination.

In 2013, Ferguson *et al.* studied the interconnection, capacity, geography, and growth of Cogent's network. They continuously resolved the hostnames of address space used by Cogent to number their routers, and then applied a regex that they manually constructed to Cogent's hostnames to extract interface speeds, locations, and names of Cogent's routers [7]. They found that Cogent's network grew by 11 routers per week between 2012 and 2013.

Rocketfuel's undns tool [27] released in 2002 contained a list of manually assembled regexes that extracted geographic locations from hostnames to reason about POP-level ISP topology. In 2014, Huffaker *et al.* developed the DNS-based Router Positioning (DRoP) tool [10], which learned geographic components of router hostnames by identifying the position of a geographic label in a hostname relative to punctuation from the end of the hostname. They assembled a dictionary of known airport, CLLI, UN, and city names, which they used to identify candidate locations within hostnames. Their method learned a geolocation convention if the majority of inferred router locations for a suffix did not violate delay-based constraints given the position of known landmarks, and automatically built regexes to extract geolocation information from hostnames.

In 2013, Chabarek *et al.* developed a parser to extract interface types, speeds, and manufacturer information using information encoded in hostnames and a manually-assembled dictionary [6]. They also conducted a NANOG survey, which received 22 responses; 5 of the operators had automatic name generation, and 2 used a script to build their zones. Their dictionary contained 5 known IPv4 address format strings, 26 common interface type strings, and 19 common router role strings covering core, peering, and access roles. They used clustering to group hostnames with similar structures, and inspected the clusters to extract information, congruent with their dictionary. Our method does not use a manually-assembled dictionary to guide regex building, because we cannot assume operators use the common interface types in their hostnames, and a dictionary will become out of date over time. We instead rely on the ability of our method to learn the substrings used by operators and embed them into a regex using available training data.

2.4 Building Router Graphs

Researchers have put considerable effort into alias resolution techniques that can infer if two IP addresses are assigned to the same router, a critical part of building a router-level graph [28], because traceroute returns a sequence of interface IP addresses, rather than a unique identifier for each router. There are two common approaches to alias resolution: probe-based active methods that reveal signatures that imply two IP addresses are aliases, and passive approaches that infer router aliases using graph analysis techniques.

In 2000, Govindan *et al.* developed Mercator [8], which sends active probes to an unused port to solicit port unreachable responses for each candidate alias, and infers two probed addresses are aliases when the same source address is in the responses. In 2002, Spring *et*

al. developed Ally as part of the Rocketfuel ISP mapping system [27]. Ally infers two addresses are aliases if IP-ID values in responses to interleaved probes it sent to each candidate appear to be assigned from a single counter. Because probes are sent to pairs of interfaces, resolving a graph of N interfaces requires $O(N^2)$ probes.

In 2004, Spring *et al.* discussed additional heuristics for resolving aliases [26], including the first passive approaches based on graph analysis: interface addresses immediately preceding a common successor are likely aliases when routers interconnect with point-to-point links, and interface addresses observed in a single traceroute cannot be aliases if there are no forwarding loops. They also extended Rocketfuel's undns tool [27], which previously focused on extracting geolocation information from hostnames (§2.3), to also extract fragments of hostnames that uniquely identify a router using regexes. They built 16 conventions by hand through observing patterns in interface hostnames they clustered using aliases inferred with active probing from Mercator [8] and Ally [27]. In this work, we build an algorithm to automatically derive conventions for 2550 suffixes in 16 sets of training data across 9 years.

In 2008, Sherwood *et al.* developed Discarte [25], which used the IP Record Route option in traceroute probes, as well as graph analysis, to resolve IP aliases and identify hops where routers do not respond to traceroute probes. Both Sherry *et al.* (2010) and Marchetta *et al.* (2013) developed techniques that use the IP prespecified timestamp option to infer aliases using timestamp patterns in packets. Because only 40 bytes of IP options can be contained in a single packet, these techniques can only test pairs or small (up to 4) sets of IP addresses at a time [19, 24].

Recent work has focused on improving the scaling of alias resolution, in order to build more accurate and complete router-level maps. In 2008, Bender *et al.* showed it was possible to solicit IP-ID values from multiple candidate aliases in parallel, and evaluate candidate alias pairs offline, using the RadarGun [4] tool. They demonstrated RadarGun on 9,056 candidate aliases. In 2013, Keys *et al.* [12] and Luckie *et al.* [16] built on the RadarGun approach to build techniques capable of scalably probing millions of candidate addresses for aliases in parallel – MIDAR for resolving IPv4 aliases [12], and Speedtrap for resolving IPv6 aliases [16].

Both Keys *et al.* [12] and Luckie *et al.* [16] used regexes that they manually constructed to extract router names, which they confirmed with network operators, and then used those regexes to validate their alias resolution techniques. While both papers noted some apparent errors in the ISP's hostnames, the operator-validated regexes validated the aliases they inferred. In this work, we learn the router name component of hostnames by evaluating candidate regexes against previously inferred alias sets.

2.5 CAIDA's Internet Topology Data Kit

In this work, we use CAIDA's Internet Topology Data Kit (ITDK [5]) as training data to learn router naming conventions. We use the 16 ITDKs CAIDA built between July 2010 and April 2019, all of which collected IP paths using scamper's implementation [15] of Paris traceroute [1], and performed alias resolution using Mercator [8] and MIDAR [12]. Each ITDK contains an inferred router level graph constructed using traceroutes collected towards every routed IPv4/24 prefix from a globally distributed team of 45 – 153 vantage points (VPs) over the course of two weeks. Most ITDKs also contain a file recording the hostnames associated with each interface IP address; we obtained the hostnames for the 5 ITDKs that did not include this file from archives of CAIDA's ongoing DNS lookups that correspond to when CAIDA constructed the graph.

The number of router interfaces varied with the number of VPs that CAIDA used to collect traceroute paths, from 1.52M interfaces in 2010 to 2.75M in 2019. A consistent fraction (55.9% – 60.4%) of these interfaces had a hostname recorded. Due to visibility limitations in traceroute [13] and coverage limitations in alias resolution techniques [11], only 4.9% – 10.2% of routers in the ITDKs have more than one interface recorded. Our technique relies on routers with more than one recorded interface to evaluate the consistency and uniqueness of names inferred within a given suffix.

3 INTUITION AND CHALLENGES

Our algorithm learns if a network uses a naming convention that includes a router name by evaluating automatically generated candidate regexes using a set of routers that other alias resolution techniques previously inferred. Conceptually, we infer the regex is extracting a router name if three conditions hold: (1) if the regex extracts the same value from a set of hostnames associated with each IP address on the router, (2) the value is unique to that router, and (3) the regex behaves this way for all of the ISP's routers. This inference algorithm is challenging for three key reasons.

1. Heterogeneous Naming Conventions. We do not know, a priori, if a given suffix uses a convention that embeds router names in hostnames. Neither do we know what sequence of regex components is required to capture the naming convention, leading to a search space that is infeasible to learn through brute force. Instead, we must use heuristics to narrow the search space. When a single network uses a convention, it may use multiple different formats depending on their internal needs and the roles of their routers; a single suffix may require multiple regexes to capture the diversity of formats within the suffix and minimize false inferences.

2. Imperfect Naming Training Data. Network operators have complete control over the information they store in their zones. Some network operators maintain their zones automatically, using information stored in well-maintained centralized databases [6]. Other operators maintain their zones manually, or the centralized database might not be kept up to date. These artifacts hamper our ability to learn naming conventions, as the interfaces may appear as if they do not belong to a particular router (false negatives) or belong to a different router (false positives).

3. Imperfect Router Training Data. Alias resolution techniques (§2.4) are only feasible for a subset of the Internet topology. The most feasible IPv4 technique, MIDAR, was applicable on up to ≈80% of ≈2.3M interfaces probed in 2013 work [12], and the most feasible IPv6 technique, Speedtrap, was applicable on up to ≈30% of ≈53K interfaces probed in 2013 work [16]. Because these techniques actively probe routers, through probe scheduling, router rate-limiting, router implementations, and packet loss, it is possible for these techniques to miss aliases (false negatives). Further, these techniques may associate interfaces that are not aliases through coincidence of returned values (false positives).

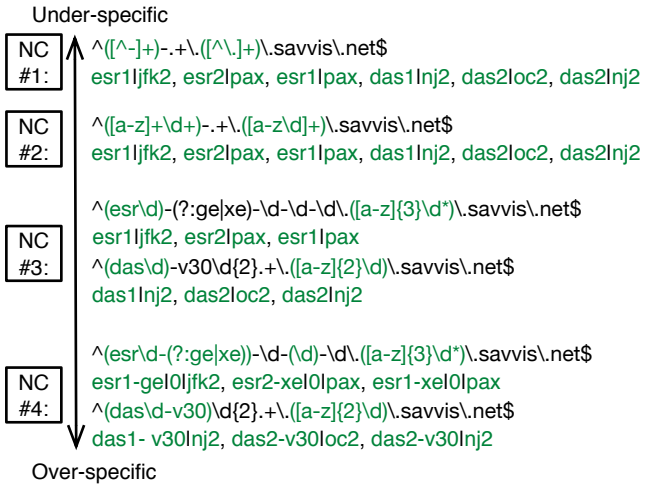

Figure 2: Specificity of naming conventions for the routers in figure 1 on a continuum. To avoid over-fitting to training data, we choose the most specific convention with the fewest regexes when choosing between conventions with similar clustering (NC #2).

Regex component	Example	Sum
Anything (score: 0 per component)	.+	0
Exclude specified punctuation (score: 1 per component)	[^-]+	1
	[^\.]+	1
	[^\.]+\.[^\.]+	2
	[^-]+-[^\.]+\.[^-]+	3
Specified classes (score: 2 per [a-z\d]+, 3 per [a-z]+ or \d+)	[a-z\d]+	2
	[a-z]+	3
	[a-z]+\d+	6
IPv4 address (score: 3 per \d+)	\d+\.\d+	6
	\d+-\d+-\d+-\d+	12
IPv6 address (score: 3 per [a-f\d]+)	[a-f\d]+	3
	[a-f\d]+-[a-f\d]+	6
Literal (score: 4 per character)	foo	12
	infra\.cdn	36

Table 2: Scores of individual regex components sum to give a specificity score. The more specific a component is, the larger the contribution to the specificity score. If two regexes evaluate the same, we break ties using the specificity score.

66.161.134.161	66-161-134-161.meyertool.com
154.126.82.122	tgn.126.82.122.tgn.mg
94.199.152.9	152-9-f7m000p01cern.core.as8723.net
92.60.81.5	5.81.unused-addr.ncport.ru
2804:321c::1	2804-321c-0-0-0-0-0-1.nslink.net.br
2a00:aa40:0:235::96	gum-core-rou-235-096.oberberg.ne
2001:4060:1:3001::2	prt-cbl-sw1-vlan-3001.gw.imp.ch

Figure 3: Examples of IP addresses embedded in hostnames. Operators do not always embed all of the IP address in the corresponding hostname.

4 PRINCIPLES

Before discussing our method, we first outline tensions and principles we arrived at for addressing them. The key issue facing our work is that it is impossible to know the intent an operator had when assigning a hostname to a router interface, or whether or not the training data for a suffix reflects their intent. That is, it is not possible for anyone other than an operator with ground truth to distinguish between an operator using multiple conventions for different routers in their suffix, and errors in the training data that follow a pattern. This section describes our approach to establishing a sound basis for naming convention (NC) inference.

4.1 Specificity

Because we use a machine learning approach to infer a naming convention, it is possible that we could derive a convention that overfits to the training data so that there is perfect alignment between the clustering in the training data and the clustering of interfaces by the naming convention. We know, however, that the training data is not perfect (§3).

Naming conventions should be as specific as possible so that they capture patterns in the training set, but no more specific than necessary. Figure 2 shows a specificity continuum for candidate naming conventions for savvis.net routers in figure 1. If a naming convention with fewer regexes achieves similar clustering against training data compared to a convention with more regexes, then we prefer the convention with fewer regexes, i.e., we prefer NCs #1 and #2 over #3 and #4 in figure 2. We do this to avoid overfitting to the training data, as our method will otherwise infer naming conventions with many regexes, each of which apply to a small fraction of hostnames, including those with errors following a pattern in them, and not representing the operator's intent.

When we build a regex, we assign each regex component a score according to how specific the component is, which we sum to obtain a *specificity score*. Table 2 lists the specificity scores per component, where more specific components have higher component scores. We chose the component scores so that we would choose the regex with the highest (most specific) score when breaking ties between regexes that perform the same clustering, i.e., we would prefer NC #2 over #1 in figure 2. Table 2 shows that we can include a variable number of components to cover IP addresses embedded in hostnames, because operators do not always embed all of the IP address in a corresponding hostname, as illustrated in figure 3.

We also prefer regexes that contain fewer extraction elements. The first regex in NC #4 in figure 2 contains three extraction elements, selecting the middle digit (i.e., 0) for savvis.net routers 1-3 in figure 1. Extracting this digit is a symptom of over-fitting, as the middle digit refers to the interface card and is not part of the router name. Further, extracting this digit provides no additional clustering benefit over the first regex in NC #2 in figure 2, which contains two extraction elements.

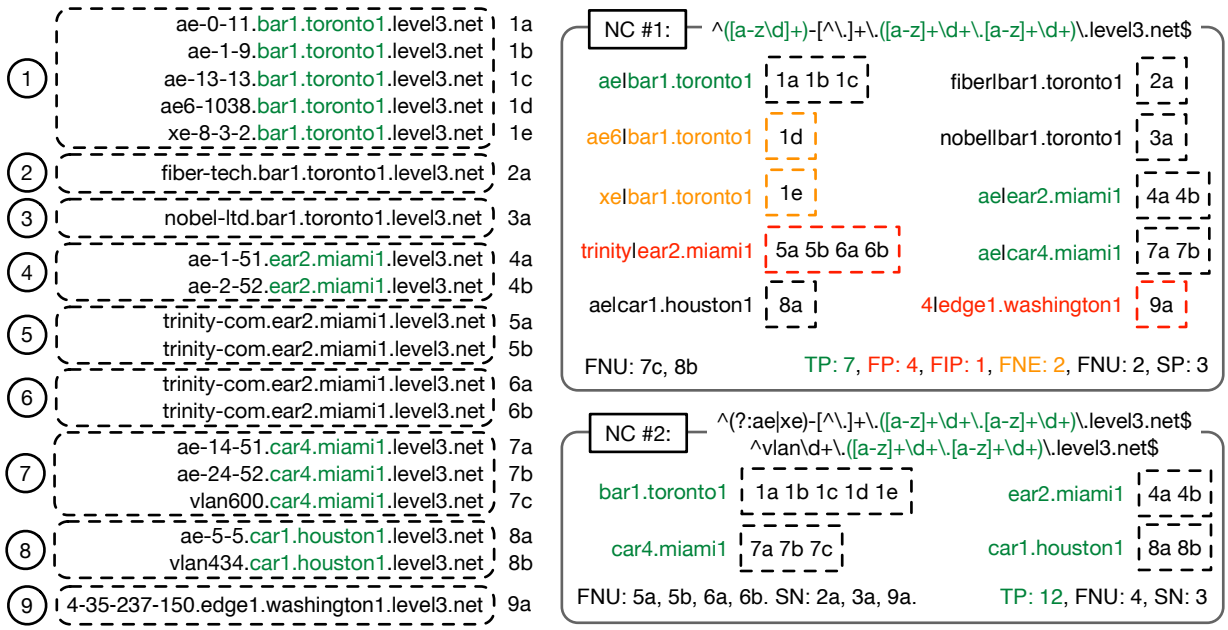

Figure 4: Evaluating two naming conventions (NCs) on Level3 training data. Table 3 defines the per-interface classifications we assign. NC #1 splits 1d + 1e from router 1 (FNE), clusters 5a + 5b with 6a + 6b (FP), and includes an IPv4 literal in the name for router 9 (FIP). NC #2 correctly clusters all hostnames where clustering is possible.

Class	Relationship to training data
TP	True positive: clustered to same training router.
FP	False positive: clustered to different training router.
FIP	False IP: extraction includes portion of IP address embedded in hostname.
FNE	False negative extraction: interfaces of training router clustered to separate routers.
FNU	False negative unmatched: regex does not match.
SP	Single positive: assigned to own cluster, no other hostnames in same suffix on training router.
SN	Single negative: regex does not match, no other hostnames in same suffix on training router.

Table 3: Per-interface classifications of clustering according to training data, guiding refinement of regexes.

4.2 Fidelity to Training Data

We evaluate naming conventions according to their ability to cluster hostnames congruent with corresponding routers in the training data. We chose an evaluation approach that guides refinement of regexes that form a naming convention. We illustrate our evaluation approach using the Level3 routers shown in figure 4; NC #2 is better than NC #1 by this principle. Table 3 summarizes the definitions for per-interface classifications we assign during evaluation.

We assign a true positive (TP) to an interface when a NC clusters at least two interfaces congruently with the clustering on the corresponding training router in ITDK. We assign a false positive (FP) to an interface when a NC clusters the interface incongruently with the clustering on the corresponding training routers.

We distinguish two classes of false negative. A *false negative extraction* (FNE) occurs when a NC separates interfaces of a training router into distinct clusters, for example interfaces 1d and 1e in NC #1 in figure 4. A *false negative unmatched* (FNU) occurs when a NC does not extract a name from a hostname on a training router that has more than one hostname in the same suffix, for example interfaces 7c and 8b in NC #1 in figure 4. We use FNE and FNU classifications to guide refinement. A FNE can indicate that a regex contains an unnecessary extraction; the first extraction element in regex in NC #1 separates interfaces from the same router, but the logical-or statement in the first regex of NC #2 retains the cluster. A FNU can indicate that a naming convention does not cluster interfaces that it should; NC #1 does not cluster 7c and 8b with their training routers, but the second regex in NC #2 does.

We assign a *false IP* (FIP) when the extraction includes a portion of an IP address that an operator embedded in a hostname – for example, for interface 9a in NC #1 in figure 4 – as a router name does not include a portion of an IP address. We detect this class of error by noting the position in the hostname of sequences of at least two IPv4 address byte values or four contiguous IPv6 hexadecimal digits that match the IP address of the interface, and determining if they overlap with the extracted name. This class of hostname often follows a pattern because operators can automatically populate these hostnames using macros provided by DNS server software. We learn filter regexes to ignore these hostnames when necessary.

We also distinguish two classes of inference when the training router has a single interface in a suffix. A *single positive* (SP) occurs when the extracted name does not cluster the interface with any other interface belonging to a training router. A *single negative* (SN) occurs when the regex does not extract a name from the hostname.

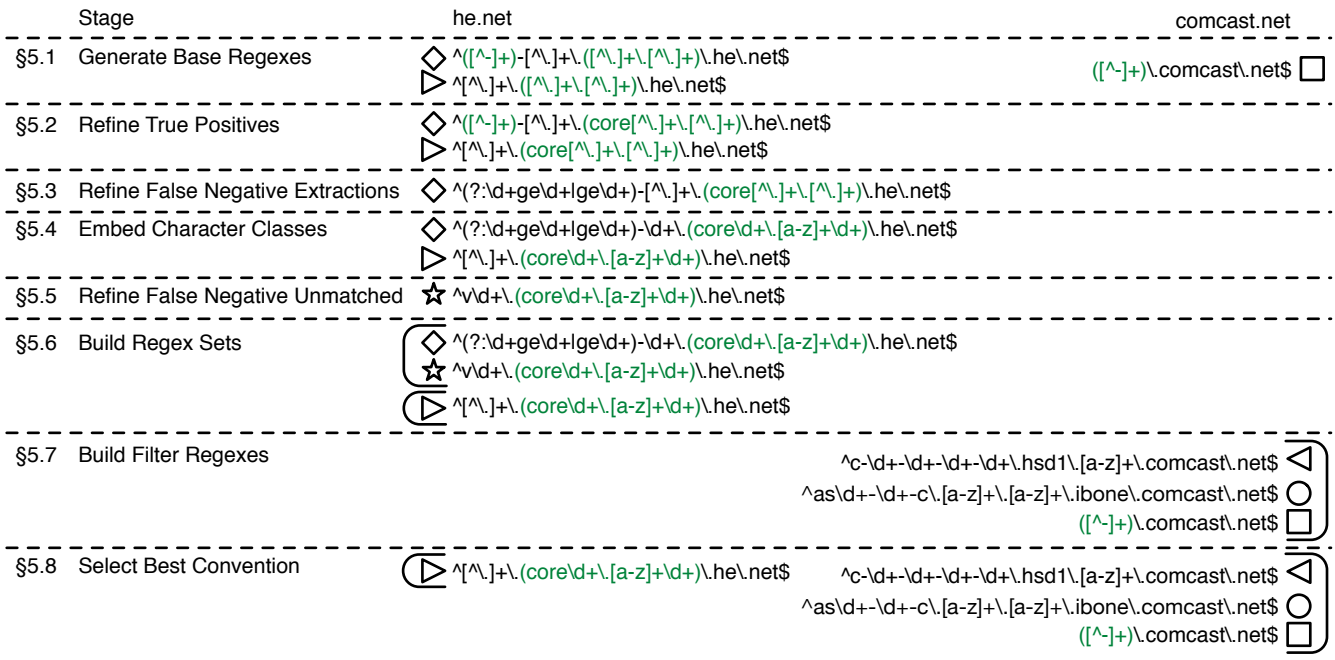

Figure 5: Method for inferring naming conventions across eight phases, and an illustration of progress through these phases for two suffixes. Not all phases may contribute to the final naming convention, but all phases are required to overcome heterogeneity in operator naming conventions. Each symbol identifies an evolving regex as our method refines it.

4.3 Ranking Regexes

Our metric for ranking regexes, which we call Absolute True Positives (ATP), is the number of true positives (TP) minus false positives (FP), false negative extractions (FNE), and false IP extractions (FIP). We do not include false negative unmatched (FNU) in our metric, as some interfaces could never be correctly clustered using their hostname. For example, when an operator names an address assigned to a neighbor for interconnection (e.g., hostnames on routers 5 and 6 in figure 4) they are not naming the neighbor's router. Neither do we include either single negatives (SN) or single positives (SP) in our metric, for the same reason. In addition, extracting router names for routers with a single interface in a suffix at best would not cluster them with any other interface; at worst, they could be incorrectly clustered with interfaces on other routers.

We considered two other approaches to ranking regexes. First, we considered using the Positive Predictive Value (PPV) – TP / (TP + FP), which is the primary statistic reported by prior work evaluating packet-probing alias resolution techniques, including MIDAR [12], Speedtrap [16], and PSTS [24]. However, a PPV ranking would prefer conventions that congruently cluster a small set of interfaces within a suffix over conventions that cluster a larger set of interfaces with a small number of errors. Second, we considered using the Rand Index [23], which is a pairwise measure of clustering accuracy [18] – (TP + TN) / (TP + FP + FN + TN). However, in large training sets this metric is dominated by true negatives – whether or not two hostnames matched by a convention belong to different routers, such that a convention with pervasive false positives can have a high Rand Index.

4.4 Refinement Conditions

Figure 5 provides a roadmap of our method, showing the evolution of regexes and candidate naming conventions for two suffixes, which we discuss in detail in §5. The first five phases build *extractor regexes* that obtain *extracted names* from the hostnames. The first phase (§5.1) builds base regexes which consist solely of components that do not contain the punctuation character specified in the component. The next three phases add specificity. The second and third phases (§5.2, §5.3) embed literal strings in these regexes, and the fourth phase (§5.4) embeds specific character classes. The fifth phase (§5.5) builds regexes that could be paired with existing regexes in the set to increase coverage, and the sixth phase (§5.6) builds sets of regexes that increase coverage using the set of regexes built in the first five phases. The seventh phase (§5.7) builds *filter regexes* to filter out hostnames that extractor regexes should not match because they assign interfaces to wrong routers, or extract a portion of an IP address embedded in the hostname, and add these filter regexes to applicable sets. A *naming convention* is therefore a set of filter and extractor regexes; the eighth phase (§5.8) selects the best naming convention among the conventions for each suffix.

We define two *refinement conditions* that a new regex building on an existing regex must meet, in order for the new regex to be included in the working set. First, the PPV of the new regex must not be more than 0.5% worse than the PPV of the existing regex; a more specific regex matching fewer hostnames with a lower PPV is worse than the existing regex. Second, the regex must infer TPs for at least three training routers for us to have confidence that the regex is capturing a component of the naming scheme.

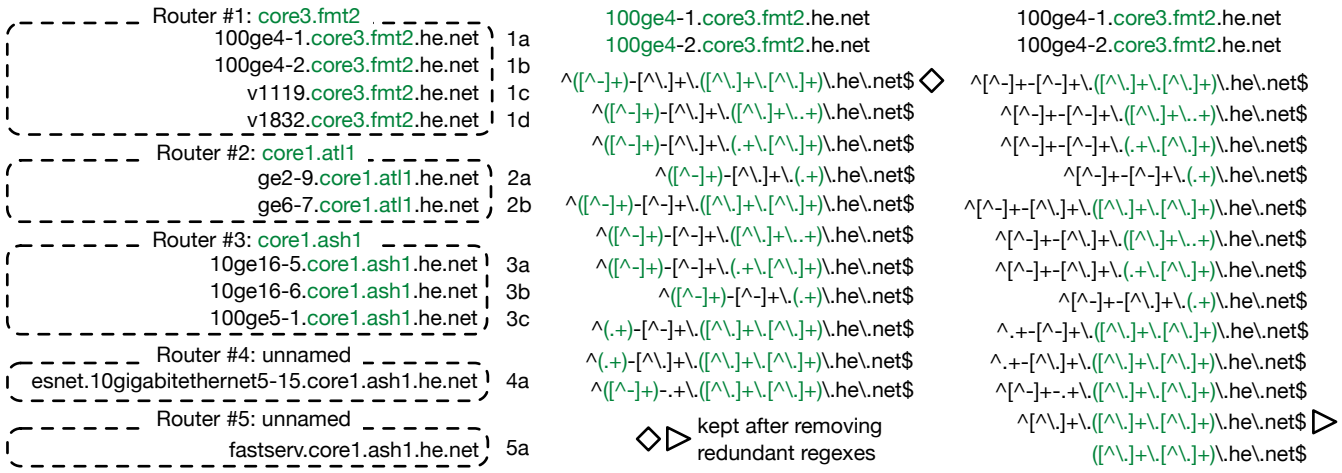

Figure 6: Example he.net routers we use to explain stages §5.1-§5.6, with the base regexes (§5.1) built for interfaces 1a and 1b.

4.5 Removing Redundant Regexes

At the end of each phase 1-5, we remove regexes to reduce unproductive exploration. We first remove regexes that do not correctly cluster hostnames for at least one training router. We then rank regexes in descending ATP (§4.3), and remove regexes whose TPs are contained in a higher-ranked regex with no additional FPs. If two regexes have the same ATP, we choose the regex with fewer extraction components, or higher specificity, and remove the other.

5 METHOD

Our method uses three data sources: router aliases inferred with MIDAR and Mercator, hostnames of those interfaces, and a list of public DNS suffixes. CAIDA's ITDK (§2.5) provides the first two, and Mozilla's public suffix list [20] provides the third. Each ITDK contains all IP addresses that available Ark vantage points observed using traceroute over a ≈2 week period. Because alias resolution techniques are only feasible for a subset of the addresses (§3) our *training set* consists only of ITDK routers with multiple aliases; these routers are *training routers*. The *application set* consists of the remaining ITDK routers with no inferred aliases.

5.1 Build Base Regexes

For each training router, we build regexes that extract candidate names, based on common substrings (CSs) between hostname pairs. We use punctuation (non-alphanumeric) characters to build structure in regexes, in line with how operators use punctuation in practice. For each hostname pair on each training router, we identify CSs in the hostnames using a variation of the dynamic programming solution to the longest common substring (LCS) problem. The conventional LCS solution extracts a single substring, but a router name can be assembled from multiple substrings within a hostname, as is the case for savvis.net in figure 1, so we greedily select non-overlapping substrings to identify CSs. Because not all substrings may be needed to uniquely identify a router (the middle digit in savvis.net routers 1-3 in figure 1 is not part of the name) we build regexes that extract all combinations of substrings.

Figure 6 shows the base regexes our method builds for a single he.net router that extract 100ge4|core3.fmt2 and core3.fmt2 when processing the hostname pair (1a, 1b); our method also builds regexes that extract 100ge4, but we do not show these regexes for brevity. Using the CSs, we divide a hostname into portions that we do and do not extract, and recursively build regexes using all combinations of regex components that match hostname components delimited by punctuation. This phase builds regexes using only regex components that exclude specific punctuation (e.g., [^-]+), or match anything (.+) at most once per regex (table 2). We do not include literals or character classes in this phase, as a full expansion using all combinations of regex components is intractable. Finally, we remove redundant regexes using the method in §4.5.

5.2 Refine True Positives

This phase refines the set of regexes by identifying common literals in correctly clustered hostnames, i.e., those that were true positives, and then refines regexes to embed those literals in the regexes. Because these literals are in common across matched hostnames, they are found in the candidate names. We illustrate this phase using routers 1-3 in figure 6, where ^[^\.]+\.([^\.]+\.[^\.]+)\.he\.net$ extracts core3.fmt2, core1.atl1, and core1.ash1 as candidate names. We recursively extract CSs from pairs of extractions, breaking on changes in character class: alphabet, digits, and punctuation – i.e., core1 from core1.atl1 and core1.ash1, and core from core3.fmt2 and core1.ash1. We then build new extraction components for the regexes, embedding the CSs, and then replace the extraction component in a copy of the base regex. We evaluate the new regexes using the method in §4.2, and add each new regex to our working set provided the two refinement conditions in §4.4 hold. At the end of this phase, we have built ^[^\.]+\.(core[^\.]+\.[^\.]+)\.he\.net$ and ^([^-]+)-[^\.]+\.(core[^\.]+\.[^\.]+)\.he\.net$. Finally, we remove redundant regexes; because the base regex ^[^\.]+\.([^\.]+\.[^\.]+)\.he\.net$ is less specific than ^[^\.]+\.(core[^\.]+\.[^\.]+)\.he\.net$, but performs the same clustering, we remove the base regex from our working set, using the method in §4.5.

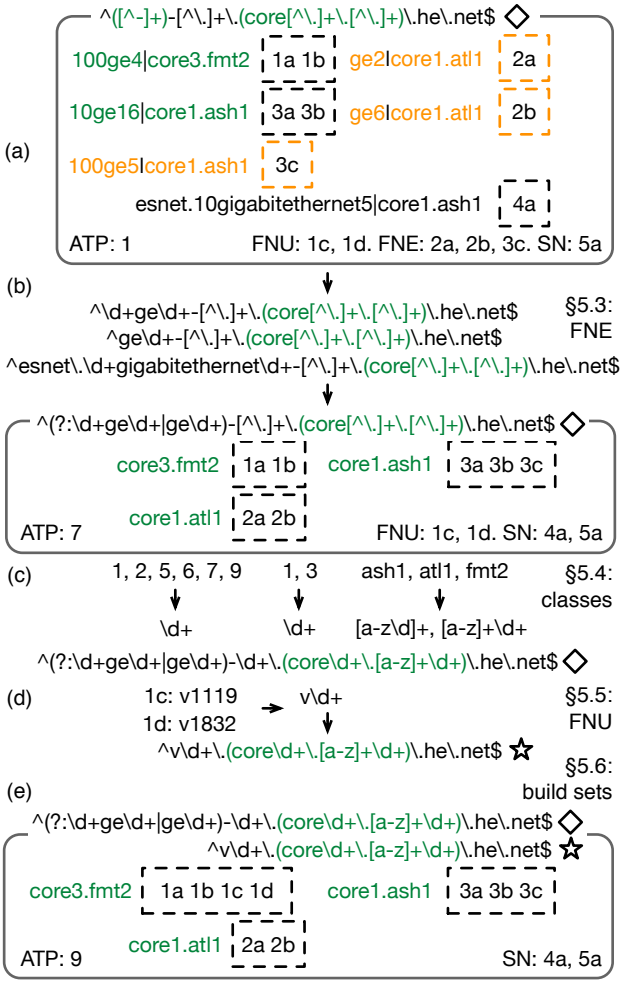

Figure 7: Refinement of he.net regexes: §5.3 – §5.6.

5.3 Refine False Negative Extractions

This phase identifies literals in the hostnames that are in common in pairs of matched hostnames, but do not form part of the router name. Regexes that extract more than the router name can separate interfaces of a training router into different clusters (FNE). For router #2 in figure 6, ^([^-]+)-[^\.]+\.(core[^\.]+\.[^\.]+)\.he\.net$ clusters interfaces 2a and 2b into ge2|core1.atl1 and ge6|core1.atl1, as shown in figure 7a. We therefore find the extraction component that is separating the hostnames, ([^-]+) in this case, and assemble all the literals obtained by that extraction component. We recursively extract CSs from these extractions, breaking on changes in character class: alphabet, digits, and punctuation as before, but also replacing digits in the CSs with regex components that match digits. For the routers in figure 6, we obtain \d+ge\d+, ge\d+, and esnet\.\d+gigabitethernet\d+.

Figure 7b shows how we build an intermediate regex set, where we replace the extraction in the regex with these patterns. We evaluate each regex in the intermediate set using the method in §4.2 and rank the regexes using the ATP method in §4.3. We add the highest ranked regex from the intermediate set to a working set, and then iteratively add other regexes from the intermediate set to the working set. In each iteration, we choose the regex that increases the ATP the most, provided the two refinement conditions in §4.4 hold. Finally, we condense the patterns into a logical-or statement – (?:\d+ge\d+|ge\d+) – and embed the statement in the extraction component that separated the hostnames in the original regex.

5.4 Embed Character Classes

This phase identifies character class sequences in common across correctly clustered hostnames, and replaces less specific regex components with components that specify character classes. Figure 7c shows that of the three [^\.]+ components in ^(?:\d+ge\d+|ge\d+)-[^\.]+\.(core[^\.]+\.[^\.]+)\.he\.net$, the first two obtain digits from the hostname, and the third obtains alphanumeric characters. For the first two components, we substitute \d+. For the third, we build [a-z\d]+ to match alphanumeric characters, and the more specific [a-z]+\d+ to match the sequence of alphabet characters followed by digits as observed in individual hostnames. We add these derived regexes to the working set provided the two refinement conditions in §4.4 hold, and then remove redundant regexes with less specific patterns using the method in §4.5 to arrive at ^(?:\d+ge\d+|ge\d+)-\d+\.(core\d+\.[a-z]+\d+)\.he\.net$.

5.5 Refine False Negative Unmatched

This phase identifies hostnames that an existing regex did not match, but that contain the same string as the extracted name from the same training router, and then builds additional regexes that match these unmatched hostnames to extract the candidate name. These hostnames were assigned FNU during evaluation (§4.2). Figure 7d shows that the regex has two FNU assignments, for interfaces 1c and 1d in figure 6. We assemble the literals from each hostname that were not part of the regex extraction (v1119 in 1c, and v1832 in 1d) and recursively extract CSs from these literals, breaking on changes in character class: alphabet, digits, punctuation, and replacing digits in the CSs with regex components that match digits – ^v\d+. We build additional regexes, embedding the CSs in the non-extraction portion of the regex – ^v\d+\.(core\d+\.[a-z]+\d+)\.he\.net$. We evaluate these regexes alongside the existing regex, and include these additional regexes in the working set provided the two refinement conditions in §4.4 hold.

5.6 Build Regex Sets

This phase increases coverage of suffixes where the operator has multiple conventions. We rank regexes by ATP (descending), and then evaluate the outcome of pairing a regex with each of the regexes below it in the rank order. We include an expanded regex in our working set provided that the two conditions in §4.4 hold, and that the ATP of the expanded regex is at least 4% more than the regex we began with. This final condition is to avoid building a naming convention that has overfitted to the training data by including many regexes, each of which apply to a small fraction of hostnames, including those with errors following a pattern, as discussed in §4.1. We execute this phase in rounds, considering additional pairings until we find no expansion that is better. Figure 7e shows that we built a set of two regexes that cluster more interfaces than the individual regexes alone.

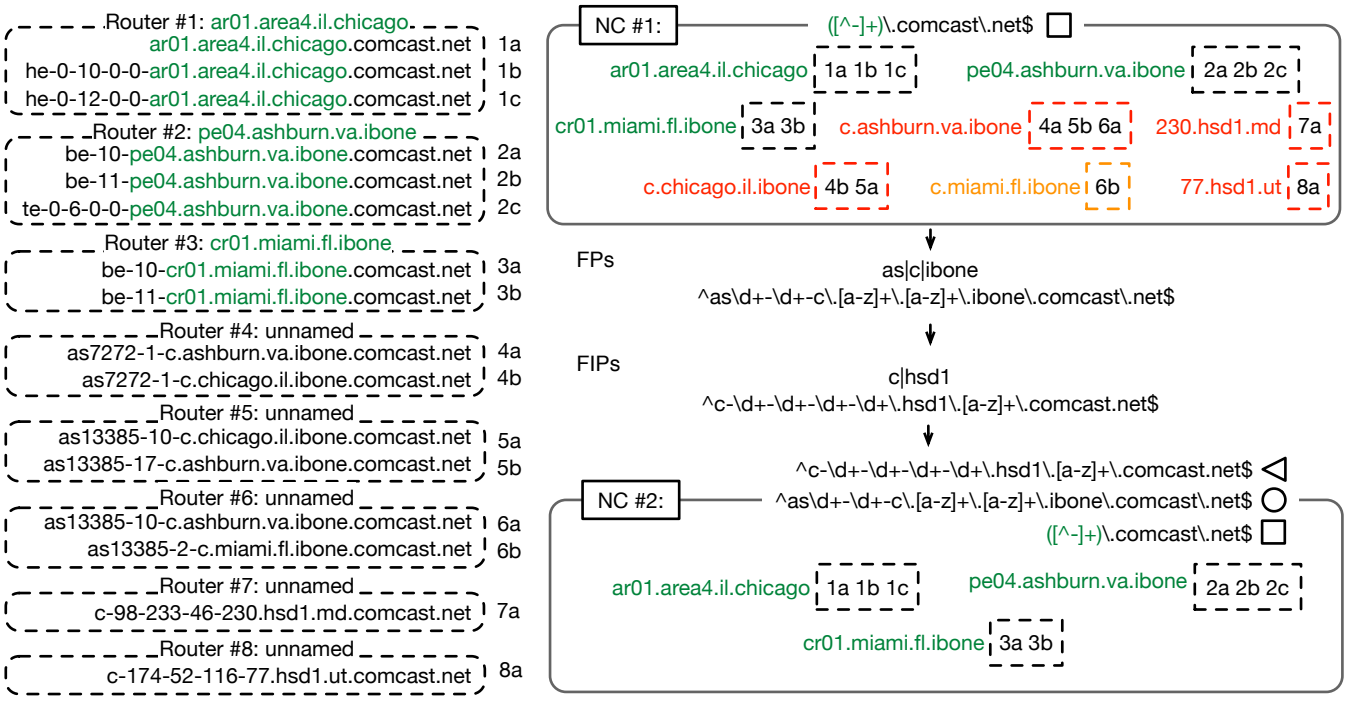

Figure 8: Refinement of comcast.net regexes in phase §5.7. NC #1 incorrectly clusters hostnames assigned to client interfaces together, and extracts portions of IP address literals, so we build filter regexes to exclude these hostnames in NC #2.

5.7 Build Filter Regexes

A regex may cluster hostnames together that are not clustered in training data, and the operator might use a convention that allows them to be distinguished. A regex may also infer candidate names for hostnames that embed a portion of a literal IP address. This phase identifies filter regexes that match incorrectly clustered hostnames (FP or FIP), so we do not use an extractor regex on those hostnames. In figure 8, ([^-]+)\.comcast\.net$ incorrectly clusters hostnames 4a, 5b, and 6a into c.ashburn.va.ibone, and interfaces 4b and 5a into c.chicago.il.ibone. Similarly, this regex incorrectly extracts 230.hsd1.md and 77.hsd1.ut, which contain a component of a literal IP address embedded in the hostname.

We assemble the hostnames with false assignments (i.e., FP and FIP) and recursively extract CSs from these components. For the FPs we extract as13385|c-ashburn.va.ibone, as7272-1-c|ibone, and as|c|ibone, and for the FIPs we extract c|hsd1. We then build filter regexes, embedding CSs in the regexes, and rank the regexes by the number of false assignments filtered (descending), then by the number of true positives filtered (ascending). We expand a candidate regex with the best filter regex, provided the following conditions hold. First, the regex must correctly filter false assignments from at least three routers for us to have confidence that the regex is capturing a component of the naming scheme. Second, the regex must filter more false assignments than true positives, i.e., must improve the PPV of the naming convention. Finally, for the FP case, the regex must reduce the inferred FPs by at least 10%, to avoid overfitting to the training data. We embed additional filter regexes until we find no additional filter that meets these conditions.

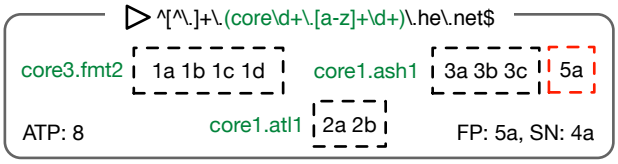

Figure 9: The best naming convention for he.net given the training set in figure 6. This convention is simpler than the one in figure 7 and results in a single FP.

5.8 Select Best Convention

It is possible to assemble a naming convention that contains multiple regexes, each covering a small portion of a suffix's routers. However, complex naming conventions may overfit to the training data, which can contain errors, and miss operator intent. We therefore penalize model complexity when selecting a best convention, with the following approach.

We rank naming conventions by ATP (§4.3) and select the highest ranked convention. Then, we consider conventions with a lower ATP value. If a lower ranked convention has an ATP value within 4% of the higher ranked convention, i.e., the higher ranked convention is not significantly better than the lower ranked convention, then we select the lower ranked convention if either of the following conditions hold. First, if the PPV of inferences unique to the higher ranked convention is at least 10% lower than the PPV of the lower ranked convention – that is, the delta of the higher ranked convention is poor, then we choose the lower ranked convention.

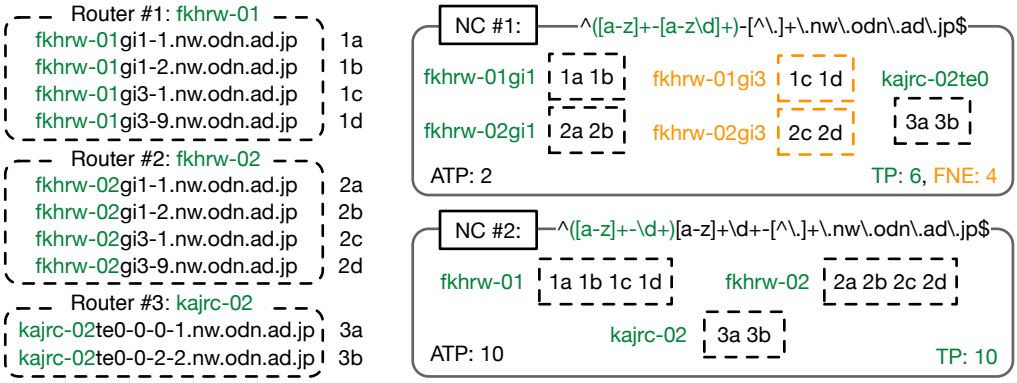

Figure 10: Routers with names not delimited by punctuation. NC #1 separates hostnames belonging to the same training router. NC #2, which we do not currently build, retains the clustering by building a regex that separates on change in character class.

Otherwise, if the lower ranked convention consists of fewer regexes and yields no more than one additional FP, then we choose the lower ranked convention according to the principle outlined in §4.1 to avoid overfitting to the training data.

We illustrate this by comparing the clustering by a more complex convention with a higher ATP (9) in figure 7 with the clustering by a less complex convention with a lower ATP (8) in figure 9. The less complex convention has only a single additional FP (caused by a stale hostname) but captures the operator intent, so we choose the less complex convention.

6 LIMITATIONS

Zhang *et al.* established in 2006 that because operators do not necessarily maintain hostnames in DNS, Internet topology mapping efforts using hostnames can be distorted [29]. Errors in hostnames can impact the accuracy of alias inferences using our regexes.

Our method currently builds regexes that extract names delimited by punctuation from hostnames, but operators do not always delimit names with punctuation. Figure 10 illustrates the problem, where NC #1 separates interfaces belonging to the same training routers in odn.ad.jp, because it extracts part of the hostname, delimited by punctuation, it should not. This limitation could be fixed by including additional heuristics in our method to build NC #2.

A fundamental limitation is that our technique cannot always cluster hostnames in different suffixes. Figure 11 illustrates the problem, where yahoo.net operators assigned addresses belonging to other networks on two of their routers, in order to connect to those networks. Because the operators of these different networks control the assignment of hostnames to their addresses, and operators can choose their own naming convention, there is no opportunity to cluster these interfaces using hostnames. In the April 2019 ITDK, 18.9% of training routers had hostnames in more than one suffix.

7 RESULTS

We evaluated our algorithm by applying it across 16 ITDKs assembled by CAIDA between July 2010 and April 2019; all ITDKs contain IPv4 topology data, and two ITDKs contain IPv6 topology data. We classify a naming convention as *poor* if it clusters interfaces on fewer than three routers (because we cannot have confidence we have found a convention), or has a PPV on the training data of less than 80% (because it did not perform well). We classify a naming convention as *promising* if it clusters interfaces on at least three but fewer than seven routers with a PPV of at least 80% (because a single FP in a small network has a significant impact on the PPV), or has a PPV of less than 90% (because the convention has predictive power but does not evaluate well). Finally, we classify the remaining naming conventions with a PPV at least 90% on more than three routers as *good*.

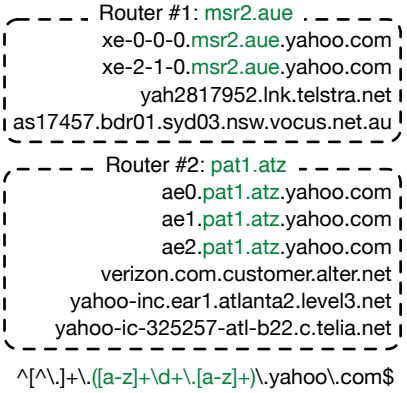

Figure 11: There is usually no way to cluster interfaces for routers with hostnames in more than one suffix because individual networks have their own naming conventions.

Figure 12 shows that we classified ≈33.5% of conventions for each IPv4 ITDK as good, covering ≈1K suffixes in each ITDK; promising conventions covered ≈3.8% of suffixes. Good conventions covered ≈60 suffixes (≈31.6%) for the two IPv6 ITDKs. We inferred at least one good convention for 2550 different suffixes across the 16 IPv4 ITDKs. However, the fraction of suffixes we inferred good conventions for IPv4 ITDKs has reduced over time: for July 2010, 35.7% of suffixes had a good convention; by April 2019, the fraction was 30.6%. This drop may reflect a reduction in coverage of active alias resolution techniques, as some operators configure their routers to ignore probes or do not announce routes for their infrastructure, and some routers do not respond to probes with a signature that is useful to active alias resolution techniques (§2.4).

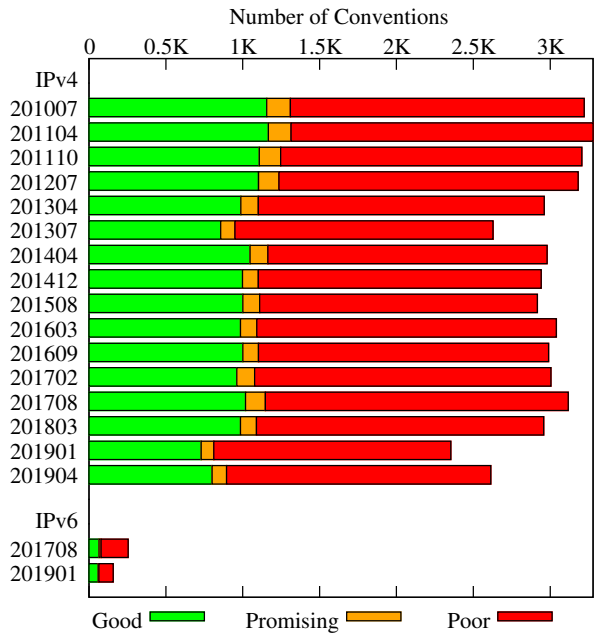

Figure 12: Summary statistics for 16 IPv4 ITDKs across nine years, and two IPv6 ITDKs. We classified ≈33.5% of conventions inferred for each IPv4 ITDK (≈1K) as good, and ≈31.6% conventions inferred for each IPv6 ITDK (≈60) as good.

7.1 Validation

We created webpages showing the naming conventions inferred over time for each suffix across the 16 ITDKs we used, and sent the webpage to NANOG in April 2019. We asked operators for each suffix whether the regexes we learned reflected their intent. We received private feedback for 11 suffixes from 10 operators. We asked each operator about discrepancies between the training data and the inferred naming convention, and about how they maintained their zones. These operators maintained zones either manually, or semi-automatically, with different approaches to automation. We summarize the validation data in table 4, identifying suffixes where the operators consented to their suffix being shared.

Of the 11, all but two naming conventions were reported as correct. Operator B confirmed that most of our inferred names were correct, but that our convention failed to extract a portion of the router name for some of their routers. Some of their routers also had incorrect hostnames; we inferred a second convention that clustered these incorrect hostnames congruently with the training data. Operator F reported that our inferred convention was not precise because it clustered some customer interfaces; the training data had incorrectly clustered them (FPs in the ITDK), and our algorithm had no opportunity to learn the correct convention.

Finally, operator C replied that our convention was correct, but supplied a second regex that filtered hostnames assigned to customer interfaces. Our training data contained 299 customer interface hostnames, 298 of which were correctly not matched by our naming convention. The filter regex would have filtered the single stale hostname that we classified as a FP; however, we require a regex to filter at least three FPs from different routers for a filter regex to be included in a convention.

A	Large North American content provider. 117 TR, 99.7% PPV, 301 TP, 1 FP, 4 FNE. Correct. Semi-automated, PTR record auto-derived from manually entered A record.
B	European Tier-1 transit provider, eurorings.net. 37 TR, 100% PPV, 110 TP, 3 FNE. Semi-correct. Manual. Two regexes, one overfitted.
C	Large European transit provider. 70 TR, 99.4% PPV, 174 TP, 1 FP, 2 FNE. Correct. Semi-automated, once per month.
D	Medium North American access network. 26 TR, 100% PPV, 52 TP. 4 TR, 100% PPV, 8 TP. Correct. Manual.
E	Medium North American access network. 8 TR, 100% PPV, 20 TP. Correct. No DNS maintenance followup.
F	Medium North American access network, ebox.ca. 5 TR, 92.8% PPV, 64 TP, 5 FP, 7 FNU. Semi-correct. Errors in training data.
G	Small North American access network, clearrate.com. 4 TR, 100% PPV, 9 TP, 11 FNU. Correct. Script periodically manually run against RANCID database. FNU are customer interfaces.
H	Small U.K. hosting provider. 3 TR, 100% PPV, 6 TP. Correct. Small, relatively static network. Manual, automation not a priority.
I	North American university. 6 TR, 100% PPV, 14 TP. Correct. No DNS maintenance followup.
J	European university, bme.hu. 2 TR, 100% PPV, 4 TP. Correct. Semi-automated from router configs.

Table 4: Summary of validation data received, with the number of training routers (TR) in each suffix.

7.2 Incongruity with the ITDK

We investigated two classes of incongruity between the April 2019 ITDK and the outcome of applying the IPv4 naming conventions we classed as good or promising. The incongruity could be because the hostnames are stale, or because the ITDK contained false negatives. The first class of incongruity is where the naming convention clustered interfaces from different training routers together. The second class of incongruity is where the naming convention clustered interfaces in the application set with interfaces on training routers; we would expect these interfaces from the application set to be on training routers in the ITDK because the training routers were responsive to alias resolution techniques used in the ITDK.

We conducted additional alias resolution probing in May 2019 to estimate the lower bound of false negatives in the April 2019 ITDK. Because we were investigating if pairs of interfaces were

	FNs in training	TNs in training	Unresp.
Training Set			
Good	98 (27.7%)	256	112 (24.0%)
Promising	28 (17.3%)	134	85 (34.4%)
Application Set			
Good	6281 (75.1%)	2086	6866 (45.1%)
Promising	429 (69.8%)	186	1217 (66.4%)

Table 5: Results of followup probing investigating incongruity with the ITDK. FNs in training data manifest as FPs in evaluation, which are actually TPs.

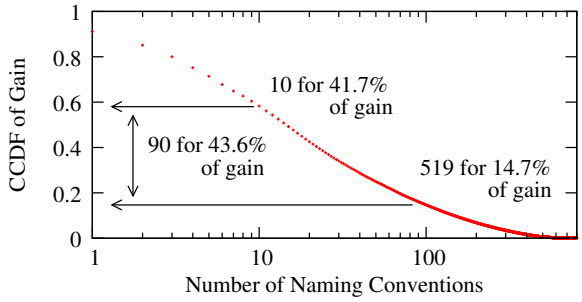

Figure 13: CCDF of the alias resolution gain of the 800 naming conventions we classified as good for suffixes in the April 2019 ITDK. While the 10 best conventions obtained 41.7% of the alias resolution benefit, a further 609 are required for the remaining 58.3%. 181 (22.6%) provided no gain.

aliases, we used Ally [27] with ICMP, UDP, and TCP probes, and Mercator [8]. Table 5 shows the results for both classes of error, for naming conventions we classed as good or promising. For the interfaces in the good class that were responsive to alias resolution at the time of our probing in May 2019, 27.7% of apparent FPs in the training set and 75.1% of interfaces in the application set were actually FNs in the ITDK.

The fraction of FNs we found in the promising class was fewer, and dominated by FNs for ntt.net routers. The FNs for ntt.net in the training data were due to MIDAR using a single probe type for each address, because MIDAR assumed that all probe types observed the same counter when multiple probe types observed a counter for an address [12]. However, TCP and UDP probes were deriving responses from different counters, yielding an inference that two interfaces were not aliases when they were. This limitation likely applies beyond ntt.net. Other FNs may derive from the fact that MIDAR's sliding window can schedule aliases into different windows, so that they have no opportunity to be resolved as aliases. Increasing the number of probe types per IP address to compensate for routers that use different counters for different probe types could result in fewer IP addresses per sliding window. Importantly, our naming conventions can guide followup alias resolution probing, increasing the accuracy and coverage of future ITDKs.

7.3 Alias Resolution Gain

We applied the 800 naming conventions we classified as good to the April 2019 ITDK. There were 18,208 routers with hostnames across these 800 suffixes; when we applied the conventions to other router interfaces in the application set, we inferred another 19,136 routers, a gain of 105%. Figure 13 shows a CCDF of the alias resolution gain per additional naming convention. Of the 800 conventions, 619 (77.4%) inferred additional aliases, and 181 (22.6%) provided no alias resolution gain. While 10 suffixes provided 41.7% of the gain, we required an additional 90 conventions to obtain an additional 43.6% gain. Finally, the remaining 14.7% gain required applying 519 additional conventions. These results show the benefit of our automated approach; building regexes by hand is labor intensive, and provides diminishing returns in the long tail.

7.4 Evaluation of IPv4 regexes against IPv6

Two ITDK datasets (August 2017 and January 2019) contain router-level graphs inferred using Speedtrap [16]. We applied the conventions that we classified as good for the IPv4 graph to the IPv6 graph. For August 2017, there were 107 suffixes in IPv6 with at least one training router; our conventions predicted the clustering of hostnames for 86.3% of these suffixes with no FPs. For January 2019, there were only 60 suffixes in IPv6 with at least one training router; our conventions predicted the clustering of hostnames for 84.5% of these suffixes with no FPs.

Operator B (§7.1) assigned hostnames to IPv6-addressed router interfaces. Our training set for this network in IPv6 consisted of a single router. However, when we applied our IPv4 naming convention to their IPv6 router interface hostnames, we found 147 hostnames on 40 routers. The operator confirmed that they used a consistent naming convention across address types, and that IPv4 and IPv6 hostnames with the same extracted name belonged to the same router.

Taking these findings to their logical conclusion, we applied the IPv4-inferred naming conventions to the IPv6 topology. For the January 2019 ITDK, there were 192 suffixes where our naming conventions applied, 124 had no routers in the training set, and there were only 416 routers in the set that did. After we applied our naming conventions to the router interfaces in the application set, we had inferred 3757 routers, a 9.0 multiplier, and nearly an order of magnitude more routers than we began with.

8 CONCLUSION

We designed, implemented, evaluated, and validated our system that automatically learned to extract router names. Our algorithm scalably builds naming conventions in phases, learning to build specificity into regexes. We publicly release our source code implementation as part of scamper [15] as well as our inferred conventions [17], allowing researchers to investigate IPv4 and IPv6 router-level congruity in the Internet. Using our system, we find 9.0 times more IPv6 routers and 105% more IPv4 routers than we started with, in applicable suffixes. Further, our conventions can guide follow-up probing to improve the accuracy of current Internet-scale alias resolution techniques, and provide a sound basis for new learning systems that use hostnames to infer router ownership, link speeds, and roles of routers in the Internet ecosystem.

ACKNOWLEDGMENTS

We thank Robert Beverly for providing detailed feedback and compute resource during the development of our system, Young Hyun and Ken Keys for assistance with the ITDK, and the anonymous reviewers for their helpful comments. This work was supported by NSF CNS-1513283, and by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via contract number 70RSAT18CB0000013 and cooperative agreement FA8750-18-2-0049, but this paper represents only the position of the authors. For Tony McGregor.

REFERENCES

[1] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. 2006. Avoiding traceroute anomalies with Paris traceroute. In *IMC*. Rio de Janeiro, Brazil, 153–158.

[2] Rohit Babbar and Nidhi Singh. 2010. Clustering Based Approach to Learning Regular Expressions over Large Alphabet for Noisy Unstructured Text. In *AND*. 43–50.

[3] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (May 2016), 1217–1230.

[4] Adam Bender, Rob Sherwood, and Neil Spring. 2008. Fixing Ally's growing pains with velocity modeling. In *IMC*. 337–342.

[5] CAIDA. 2019. Macroscopic Internet Topology Data Kit (ITDK). https://www.caida.org/data/internet-topology-data-kit/.

[6] Joseph Chabarek and Paul Barford. 2013. What's in a Name? Decoding Router Interface Names. In *HotPlanet*. 3–8.

[7] Andrew D. Ferguson, Jordan Place, and Rodrigo Fonseca. 2013. Growth Analysis of a Large ISP. In *IMC*. 347–352.

[8] Ramesh Govindan and Hongsuda Tangmunarunkit. 2000. Heuristics for Internet Map Discovery. In *INFOCOM*. 1371–1380.

[9] Philip Hazel. 2019. Perl-compatible regular expression library. https://www.pcre.org/.

[10] Bradley Huffaker, Marina Fomenkov, and kc claffy. 2014. DRoP:DNS-based Router Positioning. *CCR* 44, 3 (July 2014), 6–13.

[11] Ken Keys. 2010. Internet-Scale IP Alias Resolution Techniques. *CCR* 40, 1 (Jan. 2010), 50–55.

[12] Ken Keys, Young Hyun, Matthew Luckie, and k claffy. 2013. Internet-Scale IPv4 Alias Resolution with MIDAR. *IEEE/ACM Transactions on Networking* 21, 2 (April 2013), 383–399.

[13] Anukool Lakhina, John W. Byers, Mark Crovella, and Peng Xie. 2003. Sampling biases in IP topology measurements. In *INFOCOM*.

[14] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *EMNLP*. 21–30.

[15] Matthew Luckie. 2010. Scamper: a Scalable and Extensible Packet Prober for Active Measurement of the Internet. In *IMC*. 239–245.

[16] Matthew Luckie, Robert Beverly, William Brinkmeyer, and k claffy. 2013. Speedtrap: Internet-scale IPv6 Alias Resolution. In *IMC*. 119–126.

[17] Matthew Luckie, Bradley Huffaker, and k claffy. 2019. Data supplement for "Learning Regexes to Extract Router Names from Hostnames". https://www.caida.org/publications/papers/2019/hoiho/.

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. 2009. *An Introduction to Information Retrieval.* Cambridge University Press. https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf.

[19] Pietro Marchetta, Valerio Persico, and Antonio Pescapé. 2013. Pythia: Yet Another Active Probing Technique for Alias Resolution. In *CoNEXT*. 229–234.

[20] Mozilla Foundation. 2017. Public Suffix List. https://publicsuffix.org/list/.

[21] Karin Murthy, Deepak P., and Prasad M. Deshpande. 2012. Improving Recall of Regular Expressions for Information Extraction. In *WISE*. 455–467.

[22] New Zealand Department of Conservation Te Papa Atawhai. 2019. Yellow-eyed penguin/hoiho. https://www.doc.govt.nz/nature/native-animals/birds/birds-a-z/penguins/yellow-eyed-penguin-hoiho/.

[23] William M. Rand. 1971. Objective Criteria for the Evaluation of Clustering Methods. *J. Amer. Statist. Assoc.* 66, 336 (Dec. 1971), 846–850.

[24] Justine Sherry, Ethan Katz-Bassett, Mary Pimenova, Harsha V. Madhyastha, Thomas Anderson, and Arvind Krisnamurthy. 2010. Resolving IP Aliases with Prespecified Timestamps. In *IMC*. 172–178.

[25] Rob Sherwood, Adam Bender, and Neil Spring. 2008. DisCarte: A Disjunctive Internet Cartographer. In *SIGCOMM*. 303–314.

[26] Neil Spring, Mira Dontcheva, Maya Rodrig, and David Wetherall. 2004. *How to Resolve IP Aliases.* UW-CSE-TR 04-05-04.

[27] Neil Spring, Ratul Mahajan, and David Wetherall. 2002. Measuring ISP topologies with Rocketfuel. In *SIGCOMM*. 133–145.

[28] Walter Willinger, David Alderson, and John C. Doyle. 2009. Mathematics and the Internet: a Source of Enormous Confusion and Great Potential. *Notices of AMS* 56, 5 (May 2009).

[29] Ming Zhang, Yaoping Ruan, Vivek Pai, and Jennifer Rexford. 2006. How DNS Misnaming Distorts Internet Topology Mapping. In *USENIX ATC*. 34–39.