Scalable URL Matching with Small Memory Footprint

Anat Bremler-Barr[†], David Hay[*], Daniel Krauthgamer[†], and Shimrit Tzur-David[‡]
[†]Dept. of Computer Science, the Interdisciplinary Center Herzliya, Israel. {bremler,krauthgamer.daniel}@idc.ac.il
[*]School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel. dhay@cs.huji.ac.il
[‡]Dept. of Computer Science, Ben-Gurion University of the Negev, Israel. tzurdavi@cs.bgu.ac.il

***Abstract*— URL matching lies at the core of many networking applications and Information Centric Networking architectures. For example, URL matching is extensively used by Layer 7 switches, ICN/NDN routers, load balancers, and security devices. Modern URL matching is done by maintaining a rich database that consists of tens of millions of URL which are classified to dozens of categories (or egress ports). In real-time, any input URL has to be searched in this database to find the corresponding category.**

In this paper, we introduce a generic framework for accurate URL matching (namely, no false positives or miscategorization) that aims to reduce the overall *memory footprint*, while still having low matching latency. We introduce a dictionary-based compression method that compresses the database by 60%, while having only a slight overhead in time. Our framework is very flexible and it allows hot-updates, cloud-based deployments, and can deal with strings that are not URLs.

I. Introduction

As networks become more application- and service-oriented, *URL matching* is becoming an important component in many network devices and middleboxes.

In particular, URL matching is the basic building block of *layer 7 switches, routers, and load balancers* [18], [20], [25], [36], where routing decisions (e.g., which egress to forward a packet) are often determined by a URL (or a name) within a header of some application-layer protocol. Thus, URL (or, alternatively, service name or any other hierarchical human-readable names) matching is extensively used under Content-Centric Networking (CCN) approaches, such as Service-Centric Naming (e.g., Serval [20]) and Information Centric Networking [13] architectures like *Named Data Networking* (NDN) [2], [14], [38]. In particular, forwarding tables in high-speed NDN routers are expected to hold between $1-10$ millions URLs.

URL matching is also important in traditional networking, where it is primarily used to *enforce usage or security policy*. Modern security devices, especially in enterprises and workplace networks, are now filtering web content according to URLs (see Checkpoint [16], Palo Alto Networks [21], WebSense [33], Sourcefire [26], and others [28], [29]). In the past, such URL filtering consisted only on two categories: a blacklist of URLs that cannot be accessed and a white-list of legitimate URLs. However, contemporary URL filtering tools support tens of categories, allowing fine-grained policies which can be easily customized. Today, URL filtering tools have $1-100$ million URLs in these lists. URL matching is also used for other applications such as URL shortening services [27], [37], and search engines [15].

ISBN 978-3-901882-83-8

As the average length of a URL is 22.6 bytes, the memory footprint of the URL database often becomes humongous. On the other hand, URL matching is often performed as a *bump on the wire*, implying the URL database must be stored in an expensive fast memory to support line-rate queries. Thus, compressing the database while obtaining fast queries is essential, either to make URL-matching–based solutions feasible, or to significantly reduce their costs.

This paper tackles exactly this problem and presents a generic framework to efficiently store URLs in a *database*, along with their *categories* (in the context of Layer 7 routing or NDN/ICN, the database corresponds to a forwarding table/FIB, and a category corresponds to an egress port or ports). When the database is queried, it either returns the category attached to the queried URL or $\perp$ in case the URL is not in the database. Importantly, *we do not allow inaccurate results*—the query must always return the correct answer.

Our approach is generic in the sense that it does not have any assumption on the data structure used to perform the URL matching itself (This data structure is referred to as *database* in Fig. 1). Our framework compresses only the *information* stored in that data structure, and therefore, can leverage from any fast URL matching technique (called *database query* in Fig. 1). Specifically, our compression can be performed on the entire URL at once (applicable mainly for hash-based solution) or in a component-by-component manner (applicable, for example, to Trie-based solutions).

This paper focuses on *reducing the memory footprint of the database, while still enabling fast queries.* Since URLs share many common substrings, a naïve approach would

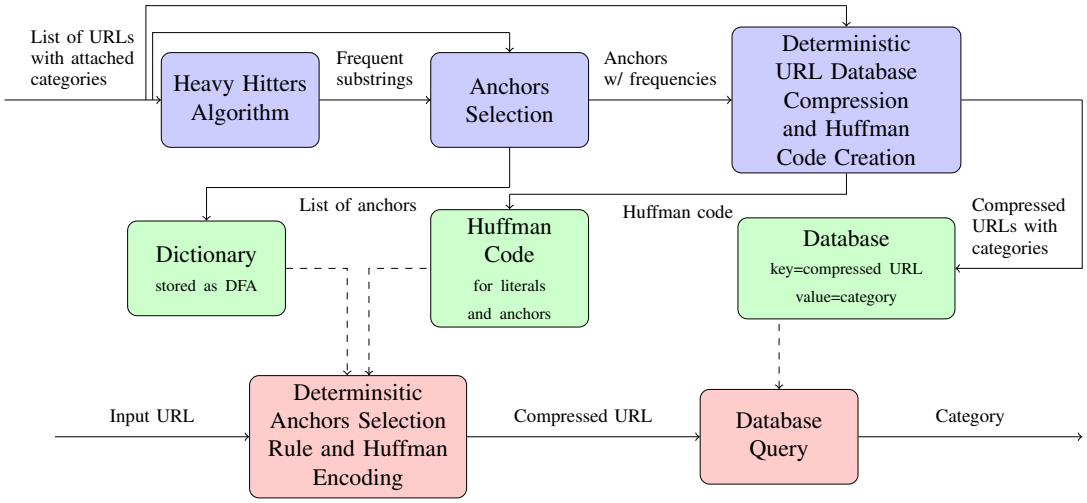

Fig. 1. A block diagram of our framework. The blue blocks describe the offline phase, where the database and auxilary data structures are built. The red blocks are the datapath of our framework. Green blocks represent the data structures, which constitute the memory footprint we try to minimize.

have been to compress the database using an off-the-shelf compression algorithm (e.g., DEFLATE used by gzip [9]). Indeed, such compression reduces the memory footprint by about 70%, however it does not allow fast queries of the compressed database. This essentially stems from the fact that the compressed form of each individual URL depends on previous URLs in the database.

Thus, we take a different approach and use a *dictionary-based compression*, which is illustrated in Figure 1. Our framework is divided to two phases: an offline phase, in which the database is built, and a datapath, in which input URLs are queried against the database.

In the offline phase, we start by extracting the frequent substrings[1] that appears in the URLs; (for example, the three most such substrings in our data were "`.com`", "`s.com`", and "`e.com`"); this is done by off-the-shelf heavy hitters algorithms, such as [1], [10]. Notice that frequent substrings intersect each others, and therefore, sometimes a substring may not be useful for compression, even though it appears many times. Thus, for each frequent substring, we first estimate its real frequency in the compressed database, and then try to determine whether it is indeed beneficial to use it (taking into account, for example, also its length and the overhead in storing it in the dictionary). Using these estimates, we select a subset of the frequent substrings as *anchors*, which will be stored in the dictionary. We also use the estimated frequency of the anchors to create a Huffman code for them (which will further reduce the memory footprint). We note that given the Huffman code and the dictionary, we can compress each URL separately, by replacing each anchor (and each *literal*) by the corresponding Huffman code. All compressed URLs are stored along with their categories in an off-the-shelf database (e.g., based on a hash-table [35], [37] or a Trie). We note that sometimes more than one anchor replacement option is available (e.g., suppose our dictionary is §1="`goo`", §2="`.com`", and §3="`ogle`"; the URL `google.com` can be compressed to either §1`gle`§2 or `go`§3§2). Hence, our compression algorithm uses a *deterministic rule* to choose the proper replacement, implying that the compressed form of each URL depends only on the dictionary used.

The datapath works in a similar way: We first identify which anchors are in the input URLs, then we use the same deterministic rule to choose which anchors to use, and finally we use the same Huffman code to encode the selected anchors and literals. Thus, when querying the database, it is sufficient to use the compressed-form of the queried URL as a key to the database.

Overall our framework yields a reduction of up to 60% in memory. Naturally, using compression trades memory space with the overhead of processing (compressing) the data. However, our experiments shows that the processing overhead (namely, the URL compression) works at rate of more than 200 Mb/s on one core, which is acceptable in these settings, as these operations can be highly paralleled (either by compressing many URLs on simultaneously on multiple cores, or by having the URL compression and database query as successive stages in a pipeline). In addition, we note that our framework allows *hot-updates support*, where URLs can be easily inserted, deleted, or modified in the database. In addition, we are able to deal with both *exact matching* and *longest prefix matching (LPM)* of URLs, where the only difference is that under LPM, we must first break the URL to its components (separated by dots or slashes) and then compress each

[1]We will use the terms *string* and *substring* interchangeably when it is clear from the context.

such component separately.

We provide a full implementation of our framework in [6].

The rest of this paper is organized as follows: Section II discusses previous works on URL matching. In Section III, we describe the datapath of our framework, in which we resolve online the category of input URLs. Section IV describes the offline phase of our framework, in which the database and auxiliary data structures are constructed. In Section V, we present experimental results on real-life URL data sets. Finally, we conclude in Section VI.

II. Related Work

Performing fast URL matching has recently received extensive attention [2], [8], [30]–[32], mostly focusing on the URL matching *time* This paper focuses on reducing the memory footprint of the URL database by designing a tailored-made compression method. Once the database is compressed, it can be used in conjunction with any of the previously suggested URL matching techniques.

As opposed to our framework, which employs *lossless compression*, most of the previous works on URL compression have used lossy compression, such as a hash-chain compression [18], CRC compression [39], or hierarchical Bloom filters [11]. Clearly, lossy compression techniques have two clear drawbacks. First and foremost, they might lead to a false categorization of URLs, as several URLs (each with different category) might be compressed to the same form. In addition, these compressions are not reversible, implying updates might become more difficult.

The works in [11], [35], [37], [39] dealt only with two categories of URLs (blacklist and white-list), and cannot be easily extended to multiple categories (e.g., by assuming that each URL that is not in one category, is implicitly in the other). Hence, these works are not applicable to modern devices that have many categories and have to store significantly more URLs.

URL lossless compression using AVL tree was considered in [15]. This compression achieves 50% compression ratio, albeit with high query and insertion time (namely, tens of microseconds). We note that this solution aims to compress crawling information of web spiders, which is less time-sensitive. However, in our case, query time is important, as most URL matching lie in the critical path of the traffic. In other words, the required solution has to represent the data using as little space as possible, yet efficiently answering queries on the represented data.

Moreover, nowadays there are many URL shortening services such as [4], [27]. These services store a database that translates the shortened version of the URL back to its original form, and then redirects the user requests to the original address. Thus, this shortening is basically a *renaming* of the URLs rather than compression; the database holds the URLs in an uncompressed form and its total space is not reduced, and therefore, it is not applicable in our setting.

Another common strategy to deal with the string retrieval problem is using tries. However, tries-based solutions do not supply efficient solutions to a non-prefix query, unless a very large amount of memory is used. Note that providing a solution to the string retrieval problem usually requires to create an index to the data and the footprint that is required to represent this index is sometimes larger than the footprint of the original data. Compressing trie-based data structure for URL matching was considered in [8], [30]. In these works, the edges of the trie correspond to the components of the URLs and the basic idea is to provide each such component with a unique code; moreover, the same code can be used for different parts of the trie when its decoding is not ambiguous. It is important to notice that the technique compresses the trie-based data-structure, but its memory footprint is larger than the size of a bare list of all URLs, as almost the entire URL information is stored explicitly in the lookup table for finding the code of each component. Our framework, on the other hand, reduces the memory footprint below the size of that list, and can be used in conjunction with the techniques of [8], [30] to further reduce the memory footprint of the trie. Moreover, while the compression techniques of [8], [30] is specific to their trie-based data structure, our framework is generic and can be applied with any matching data-structure.

Usually, solutions that create an index of the data are heavy in space (e.g. [17]). Dealing with this inefficient footprint was a subject recent research (e.g., [12]). Our solution obtains better results, in compression ratio as well as lookup performance. Actually, the lookup performance of this work is two orders of magnitude slower than our solution, as presented at Section V. We also note that compressing string dictionaries was a subject for extensive research (c.f. [7], [19] and references therein); however, these solutions were not geared specifically to URL databases.

Retravi et al. have studied how to compress IP forwarding table [23]. However their solution is not applicable to URL matching, since it assumes either a limited number of categories (corresponding to the possible next hops) or very rare updates.

Finally, femtoZip [10] is a compression library that aims in compressing short documents. As our framework, it constructs a shared dictionary which are used by all documents; unlike our scheme, this shared dictionary is a *concatenation* of all anchors, and therefore, references to this dictionary is done by referring to the offset of the anchor from the beginning of the dictionary and the length of the anchor. Concatenating anchors yields a more compact dictionary (e.g., one can refer to all substrings of

each anchor, by the appropriate offset and length), yet references the dictionary from the compressed document require three bytes for 64 KB dictionary with anchors of up to 256 bytes. femtoZip is not applicable to our setting as, on one hand, it does not provide fast compression after the dictionary is built (needed for our datapath), and, even more importantly, holding the dictionary as a concatenation of anchors rules out efficient encoding of the anchors. As we will show later, Huffman encoding of the anchors achieves significant improvement in the compression ratio, as, on average, each anchor requires less than three bytes (this is due to the fact that there are many frequent anchors whose short encoding reduce the overall size significantly). Nevertheless, we use components of femtoZip as one of the alternatives for a heavy hitters algorithm (to extract frequent substrings), as described in Section IV-A.

III. The Datapath and Data Structures

In this section, we explain the different steps taken to resolve the category of an input URL. Essentially, it include a *compression stage*, and then a *database query* with the URL in a compressed-form as a key. Notice that this process is done online, and it assumes that the database and auxiliary data structures are given (see Figure 1). We will explain how to obtain the database and construct the data structures in Section IV. Moreover, in case that either a trie-based database is used or a longest-prefix matching is required, we first break the URL into components (by "`.`" and "`/`" delimiters) and then compress each component separately. First, given the input URL, we extract the anchors which are contained in that URL. This is done by applying a *pattern matching algorithm* on the URL, where the set of patterns is the set of anchors. In this work, we chose to use the Aho-Corasick algorithm [3], which is based on traversing a Deterministic Finite Automaton (DFA) representing the set of anchors. Thus, the dictionary is stored as a DFA, whose accepting states represent anchors. We note that it is useful to store additional information about each anchor in the DFA, namely the length of the corresponding Huffman code and a pointer to the Huffman code itself. Compressing the size of the DFA for pattern matching algorithm was the subject of an extensive research recently; in this work, we use the most compressed form as presented in [5].

We notice that at each byte traversal of the DFA, the *DFA state* represents the set of anchors which are suffixes of the URL up until the scanned byte. We will need to decide *deterministically* which of the anchors in this set should be used indeed for compression (recall the example in Section I where the URL `google.com` can be compressed into two different forms by different choice of anchors). As we aim to minimize the total length of the compressed URL, we are using the following greedy approach, which traverses the DFA one byte at a time, and (conditionally) pick the anchor that minimize the length of the scanned prefix.

Specifically, for each anchor or literal a, let $\ell(a)$ be its length in bytes and $p(a)$ be its Huffman code length. Let u_i be the i-th byte of the URL, and let S_i be the set of anchors which are suffixes of the prefix of the URL up until u_i (as represented by the DFA state after scanning u_i).

The *deterministic selection rule* works iteratively by maintaining two vectors P and V, such that $P[i]$ is the minimum length for encoding the first i bytes of the URL, and $V[i]$ is the last anchor or literal that achieves encoding length $P[i]$. Hence, for each byte i, we first calculate

$$V[i] = \underset{a \in S_i \cup \{u_i\}}{\arg\min} \left(P[i - \ell(a)] + p(a) \right),$$

and $P[i] = P[i - \ell(a_i)] + p(V[i])$ ($P[0]$ is always 0).

When completing the traversal of the entire URL, we go backwards on the elements of $V[i]$ and concatenate them, skipping non-useful elements (namely, after adding $V[i]$, we add element $V[i - \ell(V[i])]$, skipping all elements in between). It is easy to verify by induction that this selection results in an optimal-length encoding (given the set of anchors and Huffman code), thus achieving the best compression ratio.

Fig. 2 depicts a step-by-step example of compressing the URL `comrgnetwork.com` in a component-by-component manner. Note that, for example, in the 5th step, the DFA finds a matching with anchor `mrg`; however the value of $P[5 - \ell(\texttt{mrg})] + p(\texttt{mrg}) = P[2] + 3 = 14$ is larger than choosing the last literal `g`. When scanning the arrays backwards, the anchors and the literals `network`, `g`, `r`, and `com` are selected. The second component `com` is compressed by running the first three steps of the above-mentioned execution.

The final step is to use the compressed-form to query the database. Since we require accurate results (namely, no false positive and miss categorizations), the database maintains also the compressed-form of the URL and not only its category. Most current implementation uses a hash-table to maintain the database [37]. Thus, by comparing the lookup key with the stored key, one avoids miss-categorization due to hash collisions. Trie-based solutions and longest-prefix matching usually require a component-by-component compression and lookup. Clearly, our framework readily supports such data-structures and matching, albeit with smaller compression ratio as some compression opportunities (e.g., anchors that span more than one component) may be missed.

IV. The Offline phase: Building the Database

As illustrated in Figure 1, the offline phase of our framework, consists of three steps:

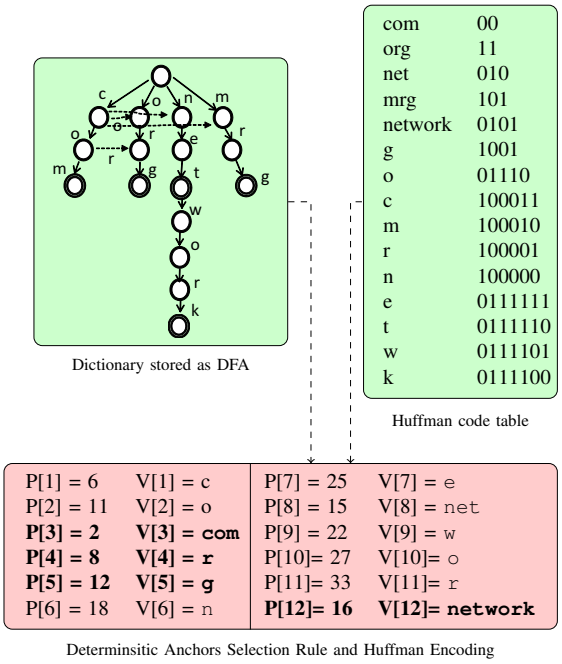

Fig. 2. Compressing the URL `comrgnetwork.com` in a component-by-component manner. As shown in Fig. 1, the compression uses a dictionary stored as a DFA and a corresponding Huffman code table. The resulting compressed URL comprises of the anchors `com,r,g,network,com` and is of length 18 bits (16 bits for the first component whose code is 0010000110010101 and 2 bits for the second component whose code is 00).

Step 1: **Heavy hitters algorithm**, in which we find a set of k frequent substrings in the set of URLs database.

Step 2: **Anchors selection**, in which we pick, from the frequent substrings, a final set of anchors. For each anchor and literal, we also calculate the estimated number of occurrences in the compressed URL database

Step 3: **Deterministic URL database compression and Huffman code creation**, in which we use the selected anchors to replace substrings in each URL separately. We also create an Huffman code using the given frequencies of literals and anchors, which we then use to encode each URL.

Next we elaborate on each step.

A. Heavy Hitters Algorithm

We compare between two off-the-shelf alternatives to extract the most frequent substrings in the list of URLs.

The algorithm described in [1] is geared to find frequent substrings of variable length. Specifically, it returns all the substrings which have unique frequency larger than n/k, where n is the number of URLs and k is a parameter that aims to calibrate the number of frequent substrings we need to find. The algorithm is approximated and the frequency of each substring is only estimated (with an error bound of $3n/k$). Note that, in any case, this frequency is not used later by our algorithm, as subsequent steps will estimate the actual number of times each substring is used for compression.

This algorithm works in time complexity of $O(n \cdot L)$, where L is the average URL length, in $O(k \cdot \ell)$ space complexity. The algorithm requires only one pass on the URL database and its space requirement is proportional to the number of heavy hitters. Yet, the results are only approximated with small error in the estimated frequency of the substrings and thus the algorithm might not find the real k frequent substrings.

Moreover, we note that this algorithm avoids space pollution and if a substring s is selected as a heavy hitter then the algorithm would not count appearances of a substring s', such that $s' \subseteq s$, when s' is within s. Nevertheless, appearances of s' not within s are counted, and if s' appears frequently alone, it might be selected as a heavy hitter together with s. See, for example, Fig. 3 that presents component-by-component compression of URLs. The heavy-hitters algorithm with $k = 5$ picks the substrings "`network`" and "`net`" as anchors, but not the substring "`netwo`" that never appears by itself. Naturally, when processing the entire URLs at once (see Fig. 4) the heavy-hitters algorithm finds longer anchors such as "`network.com`".

The second alternative is to use the heavy-hitters algorithm of femtoZip library. Unlike [1], this heavy-hitters algorithm is accurate, and it works by essentially enumerating all the possible substrings. Thus, it is significantly more time and space intensive, with time complexity of $O(n \cdot L \cdot \log(n \cdot L))$ and space complexity of $O(n \cdot L)$.

While we are less concern with the performance of the components in the offline phase, it is still desirable to reduce them as much as possible. In the experimental section, we show that, in practice, the compression ratio stays almost the same, whether we use the accurate or the approximate algorithm.

B. Anchors Selection

As explained before, the fact that frequent substrings intersect implies that a substring might not be used to compress sufficiently many URLs, even though it is frequent. Yet, these substrings increase the size of the dictionary, and therefore, should be eliminated.

Thus, in this step, we pick *anchors* out of the frequent substrings. Specifically, we first estimate, for each frequent substring, the *database compression frequency*—the number of times it will be used in the database compression (which is smaller than the frequency attached to it by the heavy hitter algorithm). Then, based on this estimated frequency, we will approximate both the gain in selecting the substring and the loss in terms of dictionary size, so that each substring whose gain is larger than its loss will be selected as an anchor. Finally, given the definitive

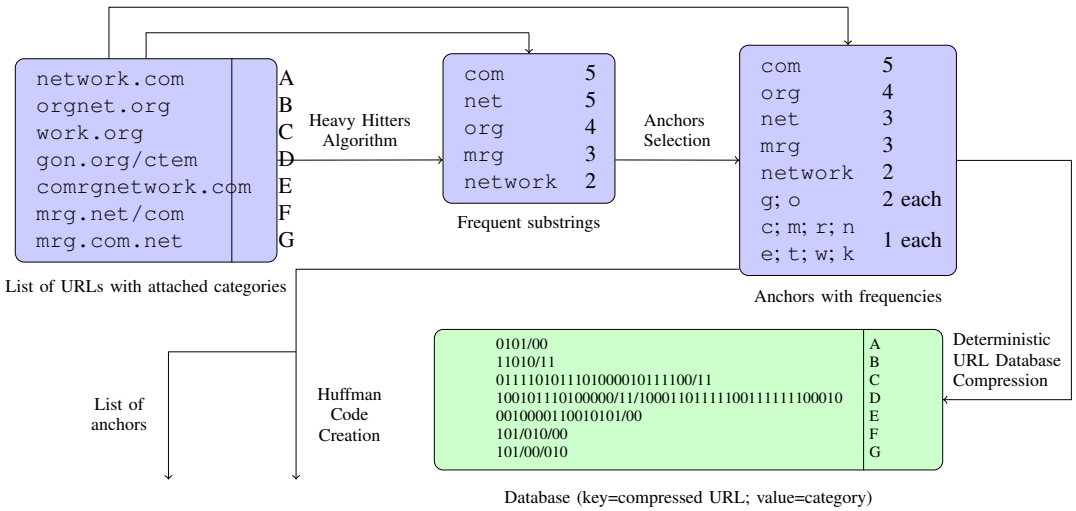

Fig. 3. An example of the offline phase of the algorithm, where compression is done in a component-by-component manner. In the compressed database, we use "/" as a delimiter between components; this delimiter is either used within a trie-based data structure or should be encoded separately. The corresponding auxiliary data structures used by the datapath (namely, the dictionary as a DFA and the Huffman code table) appear in Fig. 2.

selection of anchors, we adjust the frequency of both anchors and literals.

1) Estimating the database compression frequency of a substring: In order to calculate the estimated compression frequency, we try to estimate the compression process, as explained in Section III. Since, at this point, we cannot know what will be the Huffman code of each anchor or literal, we assume in this phase that the length of encoding each literal and each anchor is the same and, without loss of generality, is set to 1. This implies that the length of a compressed URL is estimated by the sum of the number of anchors in the compressed URL and the number of the remaining literals (e.g., in the example given in Section I, the estimated length of §1gle§2 is 5 and the estimated length of go§3§2 is 4).

As explained in I, a single URL compression involves a *deterministic selection rule* of specific anchors out of a larger set of anchors. In this step, we apply the same rule to select anchors out of the set of frequent strings, which implies we build a *temporary* DFA for all frequent substrings, set $p(a) = 1$ for each literal and anchor a, and run the greedy algorithm of Section III, one URL at the time, for all URLs in the list. Each time a frequent substring is selected as an anchor when compressing a single URL, we increase the substring's database compression frequency by 1. In the end of the process, we will have an estimation of the database compression frequencies of each substring. Notice that this is just an estimation, since not all substrings (with a frequency of at least 1) will be selected as anchors, implying the deterministic selection rule in Step 3 might yield different results. In addition, another difference in selection might be as a results of variable length encoding (with Huffman code) in Step 3; see Fig. 3 that also illustrates the new calculated frequency, where the frequency of the substring "net" is reduced as in two of its appearances the substring "network" was selected. A sketch of the DFA representing the DFA appears in Fig. 2, where some of the edges are omitted for clarity; accepting states are marked with double circles.

2) Selecting anchors out of frequent substrings: We note that while replacing a parts of a URL by anchors reduces the URL size, it comes with a price: each anchor increases the size of the dictionary's DFA and, in addition, the anchor's encoding needed to be tracked, implying even further memory footprint. Therefore, we need to avoid picking up substrings that are not used sufficiently many times.

Let A be the set of all frequent substrings, let Σ be the set of all literals, and let $f(a)$ be the number of times substring $a \in A \cup \Sigma$ was used in the previous step. If an anchor $a \in A$ is selected, for each of these $f(a)$ times, we save $\ell(a) - \text{huffman}(a)$ bytes, where $\ell(a)$ is the length of a in bytes and $\text{huffman}(a)$ is the length of the Huffman code of a. Since we cannot calculate the Huffman code of a yet, we estimate it by anchor a's *information content*:

$$h(a) = \frac{1}{8} \log_2 \frac{\sum_{a \in A \cup \Sigma} f(a)}{f(a)}.$$

Note that Huffman code strives to encode each anchor a with $h(a)$ bits. Thus, the total *gain* of selecting a is $f(a)\,(\ell(a) - h(a))$.

On the other hand, inserting a to the data structures, requires adding states to the DFA and storing its Huffman code. As explained before, we estimate the Huffman code cost by $h(a)$. As described in [5], the footprint of the DFA in its compressed form, is approximately $C_{state} = 4$ bytes

per state. Notice that two anchors that share a common prefix, share also common states in the DFA. Hence, we use the following procedure to decide whether a substring is selected as an anchor. We first have an empty DFA, and sort the substrings in descending order of their *gain*. For each frequent substring $a \in A$ in turn, we calculate the number of states $states(a)$ it requires (on top of the existing DFA) and $h(a)$. If $f(a)\,(\ell(a) - h(a)) \geq C_{state} \cdot states(a) + h(a)$, we select a as an anchor, update the DFA, and continue to the next substring. Otherwise, we leave the DFA unchanged and skip substring a. In the end of the process, we will have the set of all anchors and the corresponding DFA, that is used in the datapath.

3) Re-estimating the frequency of anchors and literals: Since only a subset of the frequent strings was selected as anchors, the frequency of anchors and literals can be changed significantly. Thus, we ran the greedy algorithm of Section III, using the DFA that was created in Section IV-B2 and with $p(a) = h(a)$ for each literal and each anchor a. This will result in an updated frequency estimation of each anchor and literal.

C. Deterministic URL database compression and Huffman code creation

Now that we have the anchors and their estimated frequency, as well as the estimated frequency of all literals, we construct the Huffman codes in a standard way, treating all anchors and literals as symbols (and, thus, ignoring their original size). The result is stored in the Huffman code data structure (namely, a table with entry for each literal and anchor, where the entries of anchors are pointed out by the corresponding DFA state). We then run once again the algorithm of Section III with the correct $p(a)$ value for each anchor and literal a. This will results in compressing each URL separately. Each compressed URL will be then inserted into the database along with its category; see Fig. 3.

D. Hot-updates Support

In order to insert a new URL, we first obtain its compressed form by going over all the steps of the datapath. Then, instead of querying the database, we perform an `insert` operation with the compressed-form URL as a key and the category as a value. Similar operation should be done in order to update a category of a URL.

Periodically, the algorithm can rebuild the database from scratch, as the frequency of substrings might change over time, resulting in suboptimal encoding.

V. Experimental Results

We have used an open-source database of URLs available in `URLBlackList.com` [28]. This daily-generated list consists of $2,200,000$ unique domain names and 95 different categories. We note that than $800,000$ URLs have also paths and not just domain names.

Compression Method	*Compression Ratio*
femtoZip	0.57
Huffman encoding only	0.59
Our Framework (accurate heavy hitter)	0.43
Our Framework (approximate heavy hitter)	0.44

TABLE I
COMPARISON BETWEEN THE COMPRESSION RATIOS OF DIFFERENT METHODS FOR MODERATE SIZE DATABASE OF 128,000 DOMAIN NAMES AND 128,000 URL COMPONENTS. THE SIZE OF THE DATABASE IN AN UNCOMPRESSED FORM IS 57.2 MB, WHILE OUR FRAMEWORK COMPRESSES THE DATABASE UP TO 24.3 MB.

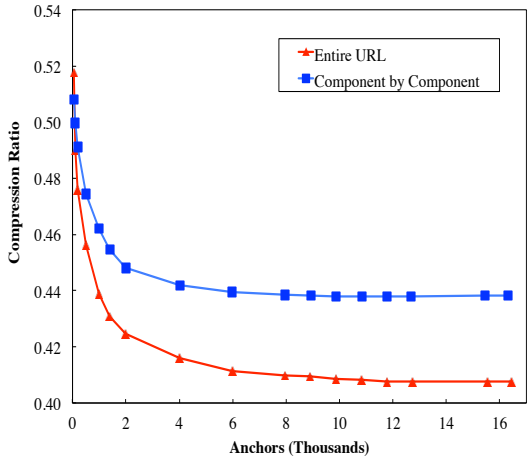

Fig. 5. The effect of the number of anchors used on the compression ratio. 50 anchors achieve 52% compression ratio for entire URLs, and 49% compression ratio where the URL is compressed component-by-component. With 16K anchors, the compression ratio improves to 40.77% and 43.8%, respectively.

The focus of this paper is the memory footprint of our framework (namely, the size of the database in its compressed-form and the size of the auxiliary data structures). This is captured by the *compression ratio*, which is the ratio between its memory footprint and the size of the database with uncompressed URLs. Notice that smaller compression ratio is better. More specifically, we have calculated our memory footprint by summing up the size of the dictionary (represented as DFA), the size of the Huffman code table, and the length of each URL in the database in its compressed form. We compare this footprint with the total length of all URLs in their uncompressed form.

We note that the memory footprint of the auxiliary data structures is only $0.1\% - 3.3\%$ of the overall memory footprint (the exact percentage depends on the number of anchors used). In practice, padding and fragmentation issues may increase this memory footprint significantly. Therefore, we have designed and implemented tailored-made memory allocator that reduces the overhead to 30%, implying that, in practice, at most 4.2% of the memory footprint is used for the auxiliary data structures.

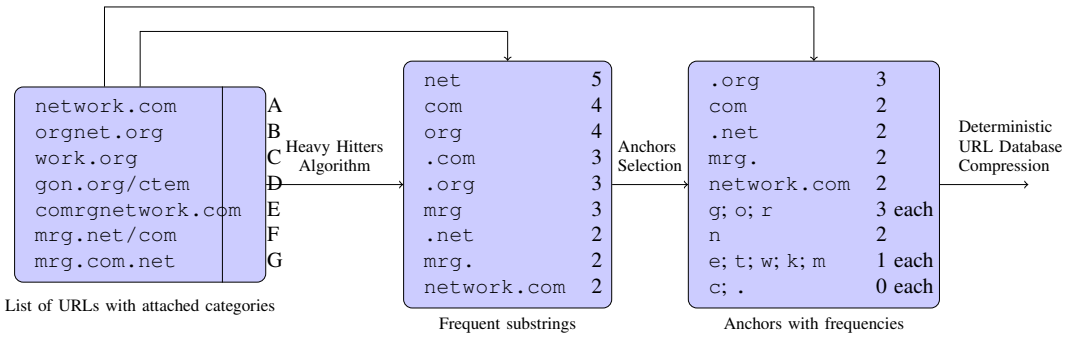

Fig. 4. An example of the offline phase of the algorithm, where compression is done in on the entire URL at once. In this example, the heavy hitter algorithm is configured with $k = 9$ and we consider only the domain part of the URLs. We note that anchors that are left with one appearance are not selected, and some literals always appear as part of one of the anchors.

A. Comparison with other lossless compression methods

Table I shows the compression ratio for several compression methods. We note that as we need to maintain low matching latency, we must compress each URL separately. Thus, methods that use backward-references (namely, applying either zip [22], bzip2 [24], or lzw [34], [40], on each URL separately) are not useful in this case (and in fact, even increase the size of it due to overhead of information they store for compression). As expected, femtoZip [10], which aims in compressing short strings, achieves reasonable compression ratio when the number of URLs is sufficiently large. Yet, it lacks an efficient datapath (namely, after the database compression, only decompression of URL is easy, while compression of new URLs using the same dictionary is difficult). Encoding the literals of the URLs with Huffman codes also reduces the memory footprint by approximately the same factor. Nevertheless, our framework outperforms all other methods. There are negligible difference between the performance of our framework with accurate or approximate heavy hitter algorithm.

B. The throughput of the datapath compression

We have implemented our datapath in C and used Intel Xeon E5-2690V3 CPU, whose processor speed is 2.6 GHz with 16 GB memory (per core), L2 cache of 256 KB (per core), and L3 cache of 30 MB. The system runs Ubuntu 14.04.2 LTS.

Each performance number was measured by applying the datapath compression 20 times on 10 randomly-selected sets of 10,000 URLs (namely, 200 runs per performance number, each representing compression of 10,000 URLs in a batch). All our experiments ran only on a *single* core. We did not measure the data-base lookup as this is orthogonal to our framework and can be implemented as a successive pipeline stage.

Fig. 6 shows the throughput (per core) of the datapath as a function of the number of anchors. The performance

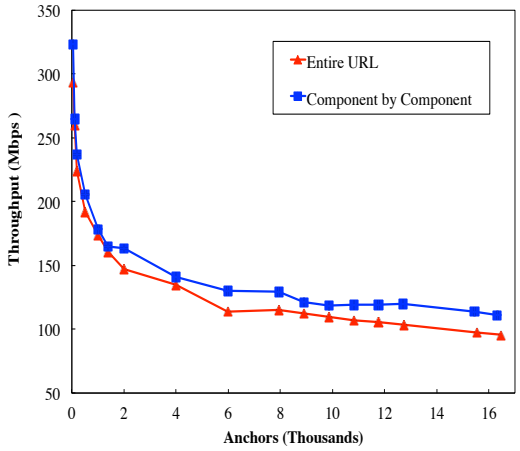

Fig. 6. The throughput of the URL compression stage in the datapath.

of the datapath depends both on the size of the auxiliary data structure and on the number of anchors (per URL) used for compression. As the number of anchors increases, longer anchors are added to the dictionary, implying the average number of anchors per URL decreases. However, larger number of anchors implies larger data structures that might not fit entirely in cache, thus causing performance degradation. In any case, the throughput is between 320 Mb/s to 100 Mb/s.

VI. Conclusions

This paper introduces a framework to significantly reduce the memory footprint of URL-based databases and forwarding tables, while maintaining the accuracy of the lookup processing (namely, no false positives or miscategorizations) and incurring only a small overhead in time. The framework also allows hot updates of the database and a longest prefix matchings.

We note that a common deployment of URL-matching–enabled devices is to store the rich database in a cloud

and store locally (in a cache) recently-matched URLs. Upon a cache-miss, the networking device will query the database in the cloud for the correct category of a URL. Our framework can be deployed in both locations. In this setting, the databases will be stored in a compressed form in the cloud, and upon a cache-miss, the networking device will compress the input URL and use the compressed form to query the database. This implies that all traffic between the networking device, its cache (if applicable), the cloud, and intra-cloud communication is done with the compressed URL, whose size is only approximately 40% of the uncompressed one. Naturally, the latency of the URL matching processes is dominated by the latency between the security tool and the cloud (the processing overhead is negligible). This implies that it may be beneficial to store in the cache also compressed URLs, increasing their number by a factor of 2.5.

Acknowledgments: This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n^{o} 259085, and also by the Neptune Consortium, administered by the Office of the Chief Scientist of the Israeli ministry of Industry and Trade and Labor.

REFERENCES

[1] Y. Afek, A. Bremler-Barr, and S. Landau Feibish. Automated signature extraction for high volume attacks. In *ACM/IEEE ANCS*, 2013.

[2] R. Ahmed, F. Bari, S.R. Chowdhury, G. Rabbani, R. Boutaba, and B. Mathieu. αroute: A name based routing scheme for information centric networks. In *IEEE INFOCOM*, pages 90–94, 2013.

[3] AV. Aho and MJ. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. of the ACM*, pages 333–340, 1975.

[4] bitly, Inc., 2013. `https://bitly.com`.

[5] A. Bremler-Barr, Y. Harchol, and D. Hay. Space-time tradeoffs in software-based deep packet inspection. In *IEEE HPSR*, 2011.

[6] Anat Bremler-Barr, David Hay, Daniel Krauthgamer, and Shimrit Tzur-David, 2015. `https://github.com/DeepnessLab/urlmatching`.

[7] N.R. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto A, and G. Navarro. Compressed string dictionaries. In *Experimental Algorithms*, pages 136–147. 2011.

[8] H. Dai, B. Liu, Y. Chen, and Yi Y. Wang. On pending interest table in named data networking. In *ACM/IEEE ANCS*, pages 211–222, 2012.

[9] P. Deutsch. GZIP file format specification version 4.3. IETF RFC 1952, 1996.

[10] femtoZip. Shared dictionary compression library, 2013. `https://github.com/gtoubassi/femtozip/wiki`.

[11] Y.-H. Feng, N.-F. Huang, and C.-H. Chen. An Efficient Caching Mechanism for Network-Based URL Filtering by Multi-Level Counting Bloom Filters. In *IEEE ICC*, 2011.

[12] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Trans. Algorithms*, 7(1):10:1–10:21, December 2010.

[13] IRTF. Information-centric networking research group (icnrg). https://irtf.org/icnrg.

[14] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, and R.L. Braynard. Networking named content. In *ACM CoNEXT*, pages 1–12, 2009.

[15] K. Koht-Arsa and S. Sanguanpong. In-memory URL compression. In *National Computer Science and Engineering Conference*, 2001.

[16] Check Point Software Technologies LTD. Check point URL filtering software blade, 2013. `http://www.checkpoint.com/products/url-filtering--software-blade/`.

[17] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[18] B. Scott Michel, Konstantinos Nikoloudakis, Peter Reiher, and Lixia Zhang. URL Forwarding and Compression in Adaptive Web Caching. In *IEEE INFOCOMM*, pages 670–678, 2000.

[19] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[20] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman. Serval: an end-host stack for service-centric networking. In *USENIX NSDI*, 2012.

[21] Palo Alto Networks. Next-generation firewall features, 2013. `http://www.paloaltonetworks.com/products/features/-url-filtering.html`.

[22] PKWARE, Inc. zip, an archive file format, 1989. `http://www.pkware.com/support/zip-app-note/`.

[23] G. Retvari, J. Tapolcai, A. Korosi, A. Majdan, and Z. Heszberger. Compressing ip forwarding tables: Towards entropy bounds and beyond. *IEEE/ACM Transactions on Networking*, 2014.

[24] J. Seward. bzip2 and libbzip2, a program and library for data compression, 2007. `www.bzip.org`.

[25] W. So, A. Narayanan, D. Oran, and Y. Wang. Toward fast NDN software forwarding lookup engine based on hash tables. In *ACM/IEEE ANCS*, pages 85–86, 2012.

[26] Sourcefire, Inc. Intelligent cybersecurity solutions, 2013. `http://www.sourcefire.com/security-technologies/-network-security/next-generation-firewall`.

[27] TinyURL!^{TM}. Making over a billion long urls usable! serving billions of redirects per month., 2013. `http://tinyurl.com`.

[28] URL.Blacklist.com, 2013. `http://urlblacklist.com/`.

[29] URLfilterDB. URL filter for the Squid web proxy, 2013. `http://www.urlfilterdb.com/`.

[30] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen. Scalable name lookup in ndn using effective name component encoding. In *IEEE ICDCS*, pages 688–697, 2012.

[31] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong. Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In *IEEE INFOCOM*, pages 95–99, 2013.

[32] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, and Z. Xu. Wire speed name lookup: A gpu-based approach. In *USENIX NSDI*, pages 199–212, 2013.

[33] Websense, Inc. The Websense Master Database. `http://www.websense.com/content/urlcategories.aspx`.

[34] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[35] T. Yamauchi, H. Yuan, and P. Crowley. Implementing URL-based forwarding on a network processor-based router platform. In *ACM/IEEE ANCS*, pages 171–172, 2009.

[36] H. Yuan, T. Song, and P. Crowley. Scalable NDN Forwarding: Concepts, Issues and Principles. In *IEEE ICCCN*, 2012.

[37] H. Yuan, B. Wun, and P. Crowley. Software-based implementations of updateable data structures for high-speed URL matching. In *ACM/IEEE ANCS*, page 15, 2010.

[38] Lixia Zhang et al. Named Data Networking Project (NDN), 2010.

[39] Z. Zhou, T. Song, and Y. Jia. A high-performance url lookup engine for url filtering systems. In *IEEE ICC*, pages 1–5, 2010.

[40] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.