Leveraging Traffic Repetitions for High-Speed Deep Packet Inspection

Anat Bremler-Barr *, Shimrit Tzur David *, Yotam Harchol †, David Hay †
*School of Computer Science, Interdisciplinary Center, Herzliya, Israel
bremler@idc.ac.il, shimritd@cs.huji.ac.il
†School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel
{yotamhc,dhay}@cs.huji.ac.il

***Abstract*—Deep Packet Inspection (DPI) plays a major role in contemporary networks. Specifically, in datacenters of content providers, the scanned data may be highly repetitive. Most DPI engines are based on identifying signatures in the packet payload. This pattern matching process is expensive both in memory and CPU resources, and thus, often becomes the bottleneck of the entire application.**

In this paper we show how DPI can be accelerated by leveraging repetitions in the inspected traffic. Our new mechanism makes use of these repetitions to allow the repeated data to be skipped rather than scanned again. The mechanism consists of a slow path, in which frequently repeated strings are identified and stored in a dictionary, along with some succinct information for accelerating the DPI process, and a data path, where the traffic is scanned byte by byte but strings from the dictionary, if encountered, are skipped. Upon skipping, the data path recovers to the state it would have been in had the scanning continued byte by byte.

Our solution achieves a significant performance boost, especially when data is from the same content source (e.g., the same website). Our experiments show that for such cases, our solution achieves a throughput gain of $1.25-2.5$ times the original throughput, when implemented in software.

I. Introduction

Content providers, such as Internet Service Providers (ISPs), Google, and Netflix maintain datacenters to host their content, or their customers' content. Usually, such providers also maintain monitoring appliances such as network intrusion detection systems (NIDS), content filtering (such as parental control services), spam filtering, network analytics, and more. All these appliances scan the payload of packets in a process known as Deep Packet Inspection (DPI). In addition, providers sometimes use *Layer 7 routing* and *Layer 7 load-balancers*, which rely as well on scanning the application layer header and work using similar techniques.

Perhaps the most important technique used in today's DPI engines is *signature matching*, in which the payload of the packet is compared against a predetermined set of *patterns* (with exact strings or regular expressions). Signature matching has been a well-established research topic since the '70s, and usually involves a *memoryless* scanning of the packets. For example, the widely-used Aho-Corasick algorithm builds a *Deterministic Finite Automaton* (DFA) to represent the set of patterns; each byte of the packet causes a transition in that DFA, and a pattern is found if the DFA transitions to an accepting state in the automaton. When scanning a byte using the Aho-Corasick algorithm, only the current state of the automaton is used. Informally speaking, this implies that no information from other packets, or different fragments of the same packet, is used to enhance the scanning process. Even if *the same packet arrives at the DPI engine many times*, the engine will *always scan it from scratch*.

A closer look at Internet traffic, and specifically HTTP traffic, clearly indicates many repetitions. Such repetitions can be classified either as *full repetitions*, in which the entire object (e.g., image, stylesheet, javascript) appears several times, or *partial repetitions*, in which only shorter fragments (e.g., shared HTML code) appear in many packets or sessions.

In content providers' networks, most of the data is highly similar and often it is simply the same files, or files with minimal modifications, that are being sent over the network. Moreover, recent trends in content providers' networks include *Software Defined Networking* (SDN), where routing is based on multiple, arbitrary header fields. Several suggestions to make SDNs aware of application layer information have been made [1], and thus we envision that DPI will be the focus of greater attention as a new bottleneck for such networks. Another interesting direction of content providers' networks is *Network Function Virtualization* (NFV), where network functions such as monitoring appliances are virtualized for higher flexibility and scalability. In some cases, these virtual appliances scan traffic from a closed set of servers or even a single server that serves several virtual machines. Thus, the similarity between pieces of data to be scanned is expected to increase. Moreover, using SDN, traffic can be made to flow so that similar traffic (from similar sources) flows to the same monitoring appliances.

Our paper presents a mechanism that uses such repetitions efficiently in order *to accelerate the signature matching component of the DPI engine*. Our mechanism is based solely on modifications to the signature matching algorithm, and thus does not involve any change to the inspected traffic or require any cooperation from any other component in the network. Conceptually, it is divided to two parts: a *slow path* that samples the traffic and creates a *dictionary* with the fixed-length popular strings (which we call *grams*), and a *data path* that scans the traffic byte by byte and checks the dictionary for matches; if a gram is found in the dictionary, the data path *skips* the gram and adjusts its state according to an information saved along this gram.

Our solution is based on the DFA-based Aho-Corasick

algorithm, but it can also be merged with other existing techniques. In the slow path, we save the state of the automaton after scanning the saved gram from the initial automaton's state. In the data path, we show that after skipping a gram, one should continue scanning from that saved state.[1] To accelerate the data path operations, we use a *Bloom filter* that represents the set of grams in the dictionary. Since Bloom filters are compact data structures, they reside in fast memory, thus reducing the overhead presented by our mechanism when there is no match in the dictionary.

We further note that our mechanism is generic and can be implemented either in software or in hardware. In software, the data path is implemented as a thread, while the slow path is implemented as another thread, possibly with lower priority. In a typical multi-core, multi-threaded environment, our solution uses a single slow-path thread that gets packet samples and calculates dictionaries, and many data-path threads (possibly on many cores), each inspecting different packets (or different connections). Since the slow path runs periodically, the marginal loss of computation power is very low, and is also adjustable. Moreover, if repeated strings are known, one can use them as a dictionary without rebuilding it. In hardware, on the other hand, we can parallel the operation in finer granularity (for example, checking the Bloom filter in parallel with scanning a byte), for a significant performance boost. Section V-A analyzes our software implementation and our proposed hardware implementation. In the software implementation, the experimental results match the predictions of the model. We analyze the real weight of each parameter in the model and apply these weights to the proposed hardware model to evaluate the benefits of a hardware implementation.

One of the greatest challenges in implementing our mechanism is deciding which grams should be saved in the dictionary at a given time. We chose to implement a variation of the algorithm suggested in [2], which is able to efficiently find the most popular strings of variable length. We then chop the strings to fixed-length grams and store those in the dictionary, as fixed-length grams are easier to handle in the data path. Naturally, the performance boost gained by our mechanism depends on the inspected traffic. We provide analysis and experimental results for several DPI use-cases that describe real-life situations, and discuss the potential speedup of our mechanism when scanning such traffic.

Our solution targets the exact string matching problem as it is an essential building block of most contemporary DPI engines. In many monitoring tools (such as Snort [3]), even if most patterns are regular expressions, string matching is performed first and constitutes most of the work performed by the engine. Specifically, Snort extracts the strings that appeared in the regular expressions and performs string matching over these extracted strings; only if all strings extracted from a specific regular expression are matched, Snort invokes regular expression engine (e.g., PCRE [4]) to check that expression. This is a common procedure since regular expression engines work inefficiently on a large number of expressions.

[1]Small modifications, explained in Section III-B, are required to avoid missing patterns in these skips.

II. Related Work

Deep packet inspection (DPI) relies on a string matching algorithm, which is an essential building block for numerous other applications as well. Therefore, it has been extensively studied [5]. Some of the fundamental algorithms are Boyer-Moore [6], Aho-Corasick [7] and Wu-Manber [8]. The seminal algorithm of Aho-Corasick (AC) is the de-facto standard for pattern matching in bump-in-the-wire. The AC algorithm constructs a Deterministic Finite Automaton (DFA) for detecting all occurrences of a given set of patterns by processing the input in a single pass. The input is inspected byte by byte. We describe the algorithm in detail in section III-A. The string matching algorithm is often a bottle-neck of the system.

There is extensive research on accelerating the DPI process, both using hardware and software implementations. The hardware implementations [9]–[15] usually use some special-purpose hardware such as FPGA or a CAM/TCAM. These solutions are usually hard to reprogram, and it is usually complicated to update their signature set. They also tie the engine to a specific type of hardware, which might make it difficult to embed these solutions. Software implementations [16]–[22], while easy to apply, reprogram, and update, have obvious performance disadvantage of being implemented on a general purpose system. All of these works *are orthogonal to our work* in the sense that all of them can be applied on top of our engine to further accelerate the DPI process.

In this paper we leverage the many repetitions in web traffic to accelerate the DPI process. Another approach that also leverages traffic repetitions is deduplication. Network data deduplication is used to reduce the number of bytes that must be transferred between endpoints, thus reducing the required bandwidth [23]–[32]. In these works, the authors find a redundancy of 35%-45% in general traffic and up to 90% redundancy in web traffic, depending on the type of the traffic.

We present an algorithm that accelerates the DPI process by leveraging the repetitions in plain, non-deduplicated traffic. Leveraging repetitions in DPI engines is entirely different from deduplication, which requires extensions and modifications on both the server and client sides, while a DPI engine scans traffic on the route between them and cannot force deduplication or assume it is used. Furthermore, leveraging repetition in DPI requires finding the repetitions on the fly, and repetitions can be short. Note that these requirements do not exist for deduplication solutions.

The work presented in [33] provides a limited solution to accelerate the DPI process using the Aho-Corasick algorithm. In this work, a repetition is defined as a repeated string that also starts at the same state in the DFA. Thus, this approach only works when scanning several copies of the exact same string, or when the same strings are stored over and over along with different starting states. However, not only can this approach can miss a repeated string, it only checks sequential strings of fixed length. The solution is thus limited and can only take advantage of repetition of big chunks such as 256-1280 bytes.

III. Enhanced Aho-Corasick Algorithm

At the heart of our solution is a modification to the widely-deployed Aho-Corasick signature matching algorithm. Our

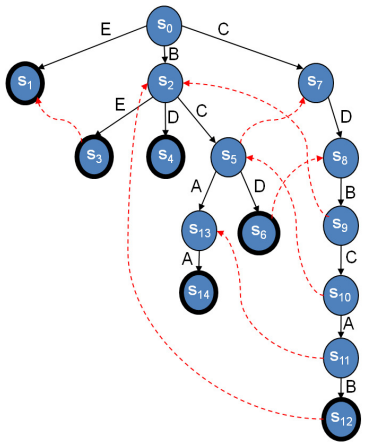


Fig. 1. The Aho-Corasick trie corresponding to the signature set *E,BE,BD,BCD,BCAA,CDBCAB*}. Solid black edges correspond to *forward* transitions, while dashed red edges correspond to *failure* transitions.

modification enhances the algorithm so that it will be able to skip previously scanned bytes, which are saved in a dictionary along with some auxiliary information. In this section we first briefly describe the Aho-Corasick algorithm and its properties, and then describe the required modifications to the algorithm. We prove that although the modified algorithm skips bytes, it detects the same patterns as the original algorithm.

A. Background: The Aho-Corasick Algorithm

The Aho-Corasick (AC) algorithm [7] matches multiple signatures simultaneously, by constructing a trie that represents the signature set. Usually, this trie is then converted into a Deterministic Finite Automaton (DFA) for better performance and then, with this DFA at its disposal the text is processed in a single pass, as we do in our implementation, described in Section VI.

The trie is constructed in two phases. First, all the signatures are added from the root as chains, where each state corresponds to one byte. When signatures share a common prefix, they also share the corresponding set of states in the trie. The edges of the first phase are called *forward transitions*. In the second phase, *failure transitions* are added to the trie. These edges solve situations where, given an input byte b and a state s, there is no forward transition from s using b. In such a case, the trie should follow the failure transition to some state s' and take a forward transition from there. This process is repeated until a forward transition is found or until the root is reached, leading to possible *failure paths*.

Figure 1 shows an example of an AC trie. Let the *label* of a state s, denoted by $L(s)$, be the concatenation of bytes along the path from the root to s. Furthermore, let the depth of a state s be the length of the label $L(s)$. The failure transition from s, $f(s)$, is always to a state s', whose label $L(s')$ is the longest suffix of $L(s)$ among all other trie states.

The trie is traversed starting from root. When the traversal goes through an *accepting state*, it indicates that some signatures are a suffix of the input; one of these signatures always corresponds to the label of the accepting state. Note that the unique structure of the trie guarantees that the converted DFA has exactly the same number of states, but many more transitions, to take care of all possible inputs without failure transitions.

TABLE I. SAMPLE DICTIONARY: EACH STRING IS ASSOCIATED WITH THE DFA STATE REACHED BY SCANNING IT FROM THE ROOT.

	string	*saved state*
	BYTAFGBC	s_5
	CABXTHGH	s_0

The correctness of the AC algorithm essentially stems from the following simple property (see, e.g., [22, Property 2]):

Property 1 *Let $b_1,\ldots b_n$ be the input, and let $s_0,\ldots,s_n$ be the sequence of states the AC algorithm goes through, after scanning the bytes one by one (s_0 is the root of the DFA). For any $i\in\{1,\ldots,n\}$, $L(s_i)$ is a suffix of $b_1,\ldots,b_i$; furthermore, it is the longest such suffix among all other states of the DFA.*

B. Enabling Skips within the Execution of the Aho-Corasick Algorithm

To enable skipping repeating data, we add to the Aho-Corasick algorithm an auxiliary *dictionary* that contains (popular) strings. We explain in Section IV how the dictionary is created, and how it is accessed from the data-path. In this section, we will show how our modified algorithm uses the dictionary in order to skip bytes during execution without missing signatures.

1) Scanning the dictionary: We assume that the dictionary is a set of strings. For each string, separately, we initiate a DFA scan from the initial state s_0. If a match is found by the end of the string, we delete the string from the dictionary[2]. Otherwise, we save the state reached by the DFA after scanning this string along with the string itself.

2) Scanning the data: During DFA traversal, for each input byte, the algorithm checks whether it can skip subsequent bytes using one of the strings in the dictionary. More formally, let $(b_1,\ldots,b_n)$ denote the data; when scanning byte b_i, the algorithm looks for the gram $gram_k(b_i)=(b_i,b_{i+1},\ldots,b_{i+k-1})$. If x is found, the algorithm proceeds in two steps.

First, it performs a *left-margin resolution* step, in which we start scanning the bytes $(b_i,b_{i+1},b_{i+2},\ldots,b_{i+k-1})$ one by one until, when scanning a byte b_{i+j} we reach a state in the automaton whose depth is less than or equal to j.

Then, if b_{i+k-1} was not reached in the left-margin resolution step, the algorithm transitions to the state which was saved along with $gram_k(b_i)$ and continues scanning from byte b_{i+k}.

3) Correctness proof: The correctness of our algorithm stems from the fact that after skipping $gram_k(b_i)$, the algorithm transitions to the same state as it would if $gram_k(b_i)$ was scanned byte by byte. In addition, we need to ensure that if some signature is detected when $gram_k(b_i)$ was scanned byte by byte, it will also be detected in our algorithm.

Theorem 1 *Let the traffic be $(b_1,\ldots,b_n)$ and let $(s_0,\ldots,s_n)$ be the sequence of states the traditional Aho-Corasick algorithm*

[2]In practice, this rarely happens and does not have any effect on the overall system performance.

TABLE II. EXAMPLE OF SCANNING PROCESS FOR INPUT STRING *CDBCABYTAFGBCD*.

b_i	C	D	B	C	A	B	Y	**T**	**A**	**F**	**G**	**B**	**C**	D
dictionary hit/miss	miss	miss	miss	miss	miss	hit	-	-	-	-	-	-	-	miss
s after scanning b_i	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_0	-	-	-	-	-	s_5	s_6
depth	1	2	3	4	5	6	0	-	-	-	-	-	2	3
j (left-margin res.)	-	-	-	-	-	1	2	-	-	-	-	-	-	-

goes through, after scanning the bytes one by one (starting from the root of the DFA). Assume that our modified algorithm scans the traffic up to byte b_i, it is in state s_i, and it found the string $gram_k(b_i) = (b_i, b_{i+1}, \ldots, b_{i+k-1})$ in the dictionary. Let z_{i+k} be the state of our algorithm after scanning byte b_{i+k}. Then, (i) $s_{i+k} = z_{i+k}$; (ii) if there are one or more accepting states in states $(s_i, \ldots, s_{i+k})$, the left margin resolution step does not end before scanning byte $b_{i+j'}$, for which $s_{i+j'}$ is the last accepting state in $(s_i, \ldots, s_{i+k})$.

Proof: If the left-margin resolution step does not end before reaching byte b_{i+k} then our modified algorithm operates exactly the same as the traditional algorithm and therefore reaches the same state. Otherwise, let j be the index in which the left-margin resolution step ends. By construction this implies that the depth of s_{i+j} is at most j, which implies that the depth of s_{i+k} is at most k (each transition in the automaton increases the depth by at most 1).

By Property 1, $L(s_{i+k})$ is the longest suffix of $(b_1, \ldots, b_{i+k})$ among all states. Since its depth is at most k, it implies that $L(s_{i+k})$ is in fact the longest suffix of $gram_k(b_i) = (b_i, \ldots, b_{i+k})$. On the other hand, by applying Property 1 on the Aho-Corasick scan of $gram_k(b_i)$ (which was performed while scanning the dictionary), we get that $L(z_{i+k})$ is also the longest suffix of $gram_k(b_i)$, which implies that $L(s_{i+k}) = L(z_{i+k})$, and therefore, $s_{i+k} = z_{i+k}$.

Similarly, assume that $j < j'$. Then the depth of $s_{i+j'}$ is smaller than j', which implies that the length of the signature corresponding to $s_{i+j'}$ is smaller than j'. By Property 1, this signature is a suffix of $(b_i, \ldots, b_{i+j'})$, and therefore it is fully contained in $gram_k(b_i)$. This contradicts the construction of the dictionary, in which strings that contain signatures are deleted. ■

C. Motivating Example

We demonstrate the insights behind our algorithm using the following motivating example. Assume that the pattern set is {*E,BE,BD,BCD,BCAA,CDBCAB*}, whose corresponding Aho-Corasick automaton is depicted in Fig. 1. In addition, we assume that the dictionary contains the strings depicted in Table I. For each such string, the resulting state in the independent Aho-Corasick scan is also saved.

There are two kinds of matches that involve strings from the dictionary:

1) Signatures whose prefix is a suffix of a string in the dictionary. For example, the prefix *BC* of the signatures *BCD* and *BCAA* is a suffix of the first string.
2) Signatures whose suffix is a prefix of a string in the dictionary. For example, the suffix *CAB* of the signature *CDBCAB* is a prefix of the second string.

Assume that the input traffic is *CDBCABYTAFGBCD*. The first five bytes did not yield any dictionary match and the Aho-Corasick is at state s_{11}. Next, string *BYTAFGBC* is in the dictionary. Since the depth of s_{11} is 5, which is greater than 0, we continue the scan with *B*. The new current state is s_{12}, whose depth is $6 > 1$, and therefore we continue to the next character, *Y*. After that, the current state is s_0, whose depth is less than 2. Thus, the left-margin resolution step is completed, and we can skip to the saved state s_5. The algorithm skips the rest of the strings' bytes (in this case k-2 bytes) and continue the scan with byte *D*. Then, the algorithm reaches the accepting state s_6 and finds the signature *BCD*. The flow of the example is presented in Table II and the skipped characters are marked in bold.

IV. SYSTEM DESIGN

Our system is divided into two components: the *slow path* and the *data path*.

A. The Slow Path

The slow path is responsible for creating a dictionary of repeated fixed-length strings (namely, k-grams, where k is the length of the strings). As explained in Section III, for each stored k-gram, we initiate an Aho-Corasick scan from the initial state s_0 and save the DFA state in the end of this scan. This information is sufficient for the data path to adjust its state after skipping that gram.

We note that while our dictionaries aim to store the most popular k-grams, they suffer from inherent inaccuracies, which sometimes reduce our mechanism's gain; our experiments show, however, that these inaccuracies are not significant. Naturally, the most important reason for such inaccuracies is that the dictionary is built on *offline, slightly outdated data*. Moreover, in a typical multi-core environment, the slow path runs on a single core and gets only *samples* of the packets. Finally, we use an off-the-shelf approximate heavy-hitters algorithm [2], which finds popular k-grams but sometimes not the optimal dictionary.

Many heavy-hitter algorithms use a sliding window and store all popular k-grams. However, this results in *dictionary pollution*, in which $m-k+1$ substrings of length k of a very popular string of length m are stored in the dictionary, while our mechanism never accesses all the strings but only m/k of them.[3] The algorithm presented in [2] solves this problem by trying to concatenate k-grams to longer strings, resulting in heavy hitters of variable length (that is, the parameter k is then the minimal length of the heavy hitters). Since our data path works on fixed-size grams better, we split each heavy hitter string of length m to $\lfloor m/k \rfloor$ consecutive k-grams.

[3] For example, assume the string `abcdefgh` is very popular in the traffic, and $k = 4$. The dictionary holds the following 4-grams: `abcd`, `bcde`, `cdef`, `defg`, and `efgh`. Most of the time, the data path uses the 4-grams `abcd` and then `efgh` in order to skip over the long popular string.

The resulting dictionary is stored as an open hash table, where colliding keys are chained. Keys are added in order of popularity, such that the most popular key is first in the chain, to improve average lookup time.

B. The Data Path

The data path uses a sliding window of length k to extract k-grams from the data. For each k-gram, the algorithm searches the dictionary and retrieves the corresponded entry if a match is found. If there is no match, one byte is scanned with the Aho-Corasick algorithm, the window slides one byte, and the process repeats itself with the next k bytes of the data. If there is a match, *left margin resolution* is performed (see Section III-B): the matched k-gram is scanned byte by byte until reaching a state whose depth is smaller than or equal to the position of the last scanned byte in the gram. Then, the data path adjusts its state to the stored state in the corresponding dictionary entry and advances to the end of the k-gram. Namely, if the k-gram has started in the i-th byte of the traffic, the next byte to be scanned will be the $(i+k)$-th one.

Since the dictionaries might not reside in fast memory or cache and might therefore require slower access operations, we first query a *Bloom filter* [34] to ensure that the gram is in the dictionary. A Bloom filter is a compact set representation (in our case, the set is all the grams in the dictionary) that enables efficient approximated set membership queries[4]; thus, if the gram is not in the dictionary, the overhead of our mechanism is reduced by one order of magnitude (see Table III for exact numbers). We note that Bloom filters sometimes generate *false positives*, which in our case imply redundant accesses to the dictionary. However, this only results in a performance penalty as the dictionary-miss is detected immediately afterwards. Since the false positive rate is very small, this performance penalty is usually insignificant. Algorithm 2 describes the data path.

a) Hardware Implementation Analysis: The data path can be implemented in hardware to utilize parallelism: In such an implementation, a dictionary lookup can be done in parallel to Aho-Corasick scan, and once a k-gram is found in the dictionary, a skip can be performed.

An additional benefit of a hardware implementation is that the Bloom filter and dictionary data structures can be put into faster memories. Current SRAM chips, which are limited to at most few megabytes, operate with access latency of about 1-10 nanoseconds, compared to DRAM chips, which provide access latency of more than 60 nanoseconds [36]. The data structures for the Bloom filter and dictionary hash table are very small in comparison to the AC DFA (tens or hundreds of megabytes for the latter, to a few megabytes in the worst-case scenario for the former). Thus, they can be located on a SRAM chip, while most of the AC DFA must reside in DRAM.

[4]We have implemented the Bloom filter as a bit array. We used the Intel on-chip instruction `crc32q` as the hash function. If more than one hash function is used, we used different random seeds for CRC to achieve independent hash functions as described in [35]. Upon a Bloom filter hit, the same hash computation is later used to access the dictionary hash table. Note that for this reason sliding window computation is easy.

```
function ScanGram(B = (b_0, b_2, ..., b_{n-1}), n)
    cur_s ← s_0
    i ← 0
    while i < n do
        gram ← (b_i, b_{i+1}, ..., b_{k-1})
        h ← Hash(gram)
        j ← 0
        if h ∈ BloomFilter then
            entry ← Dictionary[h]
            if gram = entry.gram then
                while cur_s.depth > j do
                    cur_s ← AcScanByte(cur_s, gram[j])
                    i ← i + 1
                    j ← j + 1
                i ← i + (k − j)
                cur_s ← entry.state
            else
                cur_s ← AcScanByte(cur_s, gram[j])
                i ← i + 1
        else
            cur_s ← AcScanByte(cur_s, gram[j])
            i ← i + 1
```

Fig. 2. The data path algorithm

V. Analysis

In this section we analyze the various parameters that influence the performance of our system. Given traffic of length n bytes, let k be the length of grams. Let b_i be the i-th byte of the traffic, and let $gram_k(b_i) = (b_i, \ldots, b_{i+k-1})$. We denote the dictionary as the set D and the Bloom filter that represents it as $BF(D)$. With slight abuse of notations, $x \in BF(D)$ if the Bloom filter indicates that the gram x is in D.

We first classify the bytes of the traffic according to the way our system scans them. Specifically, byte b_i is a *miss byte* if the algorithm queries the dictionary and discovers that $gram_k(b_i) \notin D$. A byte is a *left-margin byte* if the algorithm scans this byte as part of a left-margin resolution of a matched gram. Finally, a byte is a *skipped byte* if it is neither a miss byte nor a left-margin byte. For ease of presentation, we refer to left-margin and skipped bytes collectively as *in-gram bytes*. Finally, we call byte b_i a *hit byte* if the algorithm queries the dictionary and discovers that $gram_k(b_i) \in D$; note that a hit byte can be either a skipped byte or a left-margin byte.

We define p as the probability that a byte is an in-gram byte. This immediately implies that the number of miss bytes is $n(1-p)$ and that the number of hit bytes is $np\frac{1}{k}$, since only the first byte of each matched gram is a hit byte. We then define c as the average number of Aho-Corasick scan iterations until the left margin resolution is completed.

The *false positive rate* of the Bloom filter is defined as follows:

$$\text{FPR}_{BF(D)} = \Pr[x \in BF(D) \text{ and } x \notin D].$$

We note that unlike previous parameters, $\text{FPR}_{BF(D)}$ does not depend on the inspected traffic and only describes the accuracy of the Bloom filter. When it is clear from the context, we will refer to $\text{FPR}_{BF(D)}$ as FPR.

Given a byte b_i which is the beginning of a gram $x = gram_k(b_i)$, we consider three disjoint events:

1) $x \in D$: The gram x is in the dictionary D. This happens for $\frac{np}{k}$ input bytes.
2) $x \notin D$ and $x \in BF(D)$: The gram x is not in the dictionary D but $BF(D)$ falsely indicates that x is in D. This happens for $\text{FPR} \cdot n \cdot (1-p)$ input bytes.
3) $x \notin D$ and $x \notin BF(D)$: The gram x is not in the dictionary D and $BF(D)$ correctly indicates that. This happens for $(1-\text{FPR}) \cdot n \cdot (1-p)$ input bytes.

We next quantify the processing times of each operation:

- AC - The average processing time of a single byte scan using the Aho-Corasick algorithm.
- DICT - The average processing time of accessing the dictionary and retrieving the entry of a specific k-gram.
- BF - The average query time of a Bloom filter query.

When $x \in D$, the expected processing time is $\text{BF} + \text{DICT} + c \cdot \text{AC}$. When $x \notin D$, the expected processing time is $\text{BF} + \text{FPR} \cdot \text{DICT} + \text{AC}$. Putting all terms together implies that the average per-byte processing time is:

$$\frac{p}{k}(\text{BF} + \text{DICT} + c \cdot \text{AC}) + (1-p)(\text{BF} + \text{FPR} \cdot \text{DICT} + \text{AC}).$$

This processing time immediately yields that in order to accelerate the regular signature matching process (whose average per-byte processing time is AC), the ratio of in-gram bytes, denoted $p_{\min}$, should be at least:

$$p_{\min} > \frac{k(\text{BF} + \text{FPR} \cdot \text{DICT})}{(k-1)\text{BF} + (k \cdot \text{FPR} - 1) \cdot \text{DICT}) + (k-c)\text{AC}}.$$

When p_{min} is lower than this value, the software implementation is not expected to provide any speedup and thus, on such traffic, should not be used.

A. Hardware Implementation Analysis

When the data path is implemented in hardware, the Bloom filter and dictionary lookups can be done in parallel with Aho-Corasick scans. This implies a significant reduction in scanning of miss bytes: $\max\{\text{AC}, \text{BF}\}$ instead of $\text{AC} + \text{BF}$ in a software implementation. In addition, left-margin resolution of a gram can be done in parallel with the corresponding hit byte processing. When no bytes are scanned in the left boundary of a gram, one Aho-Corasick scan is still performed (since the BF query result is not known). This effectively results in a slightly higher number of in-gram bytes to scan, on average, denoted by c'; note that $c' \leq c+1$. The average per-byte processing time in such an implementation becomes:

$$(1-p) \cdot \max\{\text{AC}, \text{BF} + \text{FPR} \cdot \text{DICT}\} + \frac{p}{k}\max\{c' \cdot \text{AC}, \text{BF} + \text{DICT}\})$$

For all reasonable parameter values, the hardware implementation outperforms the naïve Aho-Corasick implementation that does not leverage traffic repetitions. Note that our approach can be used in conjunction with other hardware approaches for string matching and improve their performance.

VI. Experimental Results

Our experimental code for the data path is based on the multi-pattern matching code of the Snort [3] intrusion detection and prevention system, which implements the Aho-Corasick DFA. We added to the basic DFA code the ability to receive a dictionary, to build a Bloom filter and a dictionary hash table, and to perform skips according to our mechanism. In our experiments we limited the number of k-grams in the dictionary to about $45K$. We found that this is usually enough to achieve a high skip ratio while maintaining a relatively fast dictionary lookup process. For this number of elements, we used a Bloom filter with one hash function[5] and 0.5M-1M bits. For HTTP traffic, we used Snort's pattern-set ($\sim 4K$ patterns).

Performance was evaluated on a system with the Intel Sandy Bridge Core i7 2600 CPU with a 32 KB L1 data cache (per core), 256 KB L2 cache (per core), and 8 MB L3 cache (shared among cores). The operating system was Linux Ubuntu 11.10.

A. Traffic Sources

We used the following traffic traces for the experiment:

- **Popular Websites:** we crawled several worldwide and local popular websites and downloaded pages up to depth 2. We repeated this process every 1.5 hours to track changes in HTTP responses. For our experiments we only considered HTML content.
- **General HTML Traffic:** HTML responses from a set of HTTP traffic traces, as described in Section VI-B.
- **Cache-Miss Attack Traffic:** In a cache-miss attack, attacker sends a large number of similar patterns, multiple times, in order to force the AC DFA out of its locality area in the cache, and thus experience much more cache misses [37]. These traces mix general HTTP traffic with cache-miss attack packets, in increasing attack intensity (bandwidth), as described in [18].

B. HTTP Content Characteristics

In order to examine the potential gain from our mechanism on general HTTP and HTML traffic, we analyzed the characteristics of HTTP content. We used a 9 GB trace collected from a campus wireless network. The trace contained 348,094 HTTP flows, where an HTTP flow is defined by a request and its corresponding response.

When all the HTTP traffic was analyzed as a whole, there were fewer repetitions. However, different content types behave very differently. For example, partial repetitions are very common in *text/html* or *text/plain* types, and entire file repetitions are mainly found in images. Some of the types do not contain repetitions at all (except for some random strings),

[5] In our case we found that a single hash function provided sufficient accuracy while adding more functions greatly reduced performance.

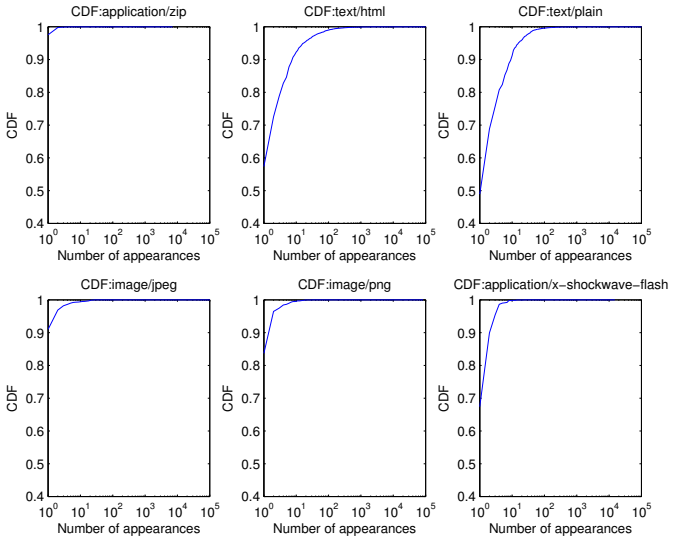


Fig. 3. CDF of number of occurrences of 16-byte sequences by content type.

e.g., *application/zip*. It is clear that we have to treat each type differently.

For each content type, we counted the repetitions of each k-gram in the data. Figure 3 presents the cumulative distribution functions (CDFs) of the number of repetitions per type for $k = 16$. As presented in the figure, for type *application/zip*, almost 100% of the strings appear only once and a few strings appear up to 10 times. However, for the type *text/html*, 40% of the strings appear more than once, 10% appear more than 10 times, and some of them appear more than 10,000 times. Similar numbers can be found for *text/plain*. Figure 4 presents the total repetitions of the different types. The types in the figure were sorted by their frequency in our traces (also represented by the gray line). The potential skip ratio was calculated as the number of bytes in all grams that appeared more than once, divided by the total number of bytes of the specific type. This is represented in the figure by the black bars in the figure. The white bars represents the potential skip ratio for all the data, i.e. the black bar times the gray line at each point. As we can see from the figure, 90% of the HTML data and 85% of the plain data can be potentially skipped (assuming infinite dictionary space), which means that our mechanism has high potential to improve performance, either in software or in hardware.

There are other content types for which much of the traffic need not to be scanned, e.g. `xml`, `asm` and `c`, but their frequency in our traces is low, and therefore their impact on the global skip ratio is almost negligible. By skipping only HTML and plain data, we achieve more than a 35% skip ratio for all the data.

C. Potential Performance Analysis

To assess the potential performance gain of our mechanism, we first isolated each component of the model described in Section V. We measured times for each operation separately (e.g. Bloom filter lookup, Aho-Corasick DFA lookup, dictionary hash table search) in units of nanoseconds per input byte. Each operation was isolated and timed using a different

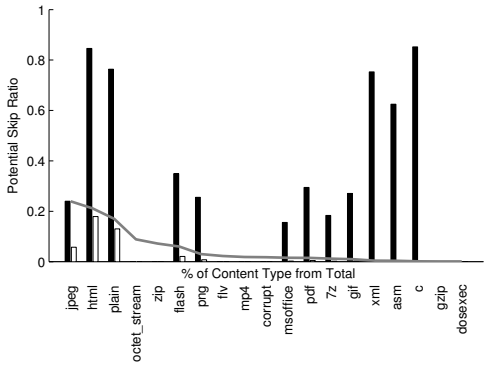


Fig. 4. Skip ratio per content type when using grams with 32-byte width.

TABLE III. SAMPLE MEASUREMENTS FOR MODEL COMPONENTS.

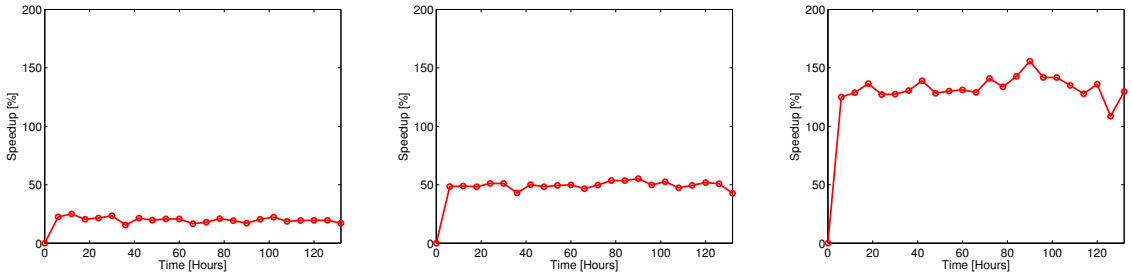
Traffic	Component	Rate
Popular Website (youtube.com)	BF	2.8 ns/byte
	AC	4.2 – 4.3 ns/byte
	DICT	27 – 28 ns/byte
	p (p_{min})	79% (65%)
	FPR	2.46%
	c	1.87 bytes
Attack Traffic (100% intensity)	BF	2.1 – 2.5 ns/byte
	AC	38 – 50 ns/byte
	DICT	28 – 40 ns/byte
	p (p_{min})	84% (27%)
	FPR	3.2 – 3.7%
	c	10.1 bytes
General HTML Traffic	BF	2.02 – 2.3 ns/byte
	AC	4.35 – 4.5 ns/byte
	DICT	25 – 28 ns/byte
	p (p_{min})	47% (61%)
	FPR	3.39%
	c	1.8 bytes

timer, in separate runs. Note that the different components have different values with each traffic source as different traffic induces different AC behavior, different dictionaries, and different Bloom filters.

Table III shows sample measurements for the different traffic sources we used. Also, for each traffic source, we show the value of p_{min} required for performance gain in the software implementation. By plugging these numbers into the corresponding equations in Section V, we can retrieve the model's prediction of speedup for each traffic source, for both the software and hardware implementations, as shown in Table IV. Note that due to compiler and CPU optimizations, the values in Table III are only rough estimations. This explains the differences between actual results and those predicted by the model. In addition, note that, for example, for general HTML traffic, while the potential skip ratio computed in Section VI-B was 90%, the limited dictionary size and fixed width bound the actual skip ratio to 47%. Note also that when implementing the solution in hardware, the Bloom filter and dictionary data structures may be put into a faster memory, not being subject to cache replacement, and thus provide better rates. However, for the comparison we assumed that rates are equal; if faster memory is used then the potential speedups on hardware will be even faster than those displayed here.

D. Speedup with Software Implementation

Figure 5 shows the actual speedup that our software implementation achieved on traffic from three worldwide and



(a) amazon.com (b) youtube.com (c) ynet.co.il (Local news website)

Fig. 5. Actual speedup achieved by our software implementation on traffic from three popular websites. Baseline throughput values for the three websites in this figure were (a) 1683 Mbps, (b) 1802 Mbps, and (c) 1710 Mbps, when using a single core for data path.

TABLE IV. MODEL PREDICTED SPEEDUPS FOR SOFTWARE AND HARDWARE IMPLEMENTATIONS, AND ACTUAL SPEEDUP ACHIEVED BY OUR SOFTWARE IMPLEMENTATION, FOR VARIOUS TRAFFIC SOURCES.

Traffic	Potential Software Speedup	Actual Software Speedup	Potential Hardware Speedup
Popular Website (youtube.com)	66%	53%	256%
Attack Traffic (100%)	112%	117%	235%
General HTML Traffic	−14%	−15%	145%

local popular websites. All websites we tested gained a positive speedup in all the experiments we performed. Achieved speedup was very close to the predicted speedup.

Our mechanism also improves DPI performance when the system is under a cache-miss attack [18], [37]. Such attacks can decrease the throughput of the AC DFA by a factor of 7 [37]. As displayed in Table III, the potential number of bytes to skip (p) is very high in such cases. During an attack, the depth in the AC DFA is deeper, as a result of the attack itself. Thus, in most cases, the left boundary scan (c) will be longer. However, as can be deduced from the analytical model, there is still a very high potential for throughput gain using our technique. Figure 6 shows the actual speedup achieved using our software implementation on traffic with various attack intensities.

E. Determining the Dictionary Width

The width of grams in the dictionary, denoted k, is an important parameter of our technique. For a fixed width dictionary, the larger k is, the longer the skips we can perform when a gram is in the dictionary. If k is too large, however, the number of grams that can be put into the dictionary is reduced. Our experiments further show that a variable width dictionary does not always perform better, due to the longer dictionary lookup process.

Figure 7 shows how throughput of the software implementation changes with k on a fixed width dictionary ($k = 0$ means no dictionary is used). In this example, the traffic is of a cache-miss attack of 33% intensity, and $k = 32$ gives the highest speedup, 33%.

F. Dictionary Creation and Update

The dictionary is first computed in the slow path when a first chunk of data is available, as described in Section IV-A. It can be computed again at each predefined interval, on the new incoming data. In our experiments, we computed a new dictionary every 10MB-20MB.

When polling the same site repeatedly every predefined interval, we create a dictionary based on several samples together (in our experiments, we polled websites every 90 minutes and created a new dictionary every six hours, on four different samples).

In most cases, a dictionary that was computed once provides a steady speedup for a long time, and it is not necessary to compute a new dictionary frequently. For example, in the popular websites we studied, we found that even when not updating the dictionary for *days*, the potential skip and actual speedup remained almost as they were when we computed a new dictionary over and over again. Figure 8 shows the speedup of the software implementation when scanning youtube.com traffic, similarly to Figure 5(b), but this time, the dictionary is only updated every 6 hours (after computing the first dictionary) and 72 hours (after computing another dictionary, three days later).

VII. CONCLUSIONS

In this work we show how repetitions in network traffic can be used to enhance DPI performance. We analyze the potential improvement using a simple, yet accurate, model, and demonstrate the effectiveness of our mechanism in a set of experiments.

Our mechanism changes the legacy Aho-Corasick algorithm, adding a dictionary of repeating data. The slow path of our mechanism uses an off-the-shelf algorithm to recognize repeating strings and create dictionaries from them. Then, in the data path, instead of simply traversing the Aho-Corasick DFA at each step, we first try to learn whether a skip is possible, and if so, avoid scanning the string again.

We show that on certain common traffic types, for various use cases, our mechanism achieves very high performance gain, when implemented in software or in hardware. We believe that our approach can improve the throughput of DPI in network middleboxes, cloud services, and SDN.

ACKNOWLEDGMENT

This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n° 259085.

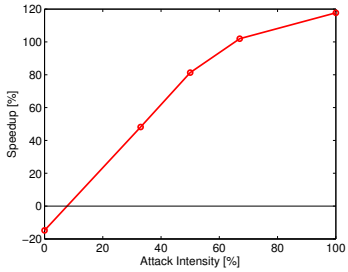


Fig. 6. Speedup achieved by the software implementation on cache-miss attack traffic with different attack intensities.

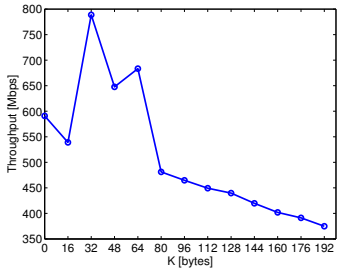


Fig. 7. Throughput change under a cache-miss attack of 33% intensity, with different widths of grams in the dictionary, k.

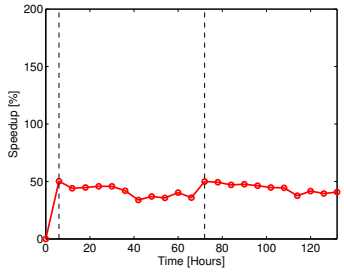


Fig. 8. Speedup of software implementation when scanning traffic from youtube.com, with dictionary update at the beginning and then only after 72 hours. Dashed lines indicate dictionary updates. Baseline throughput is 1802 Mbps.

REFERENCES

[1] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-awareness in SDN," in *SIGCOMM*, 2013, pp. 487–488.

[2] Y. Afek, A. Bremler-Barr, and S. L. Feibish, "Automated signature extraction for high volume attacks," in *ANCS*, 2013, pp. 147–156.

[3] "Snort," http://www.snort.org.

[4] "PCRE - Perl Compatible Regular Expressions," http://www.pcre.org/.

[5] B. W. Watson and G. Zwaan, "A taxonomy of keyword pattern matching algorithms," Eindhoven University of Technology, Tech. Rep. 27, 1992.

[6] R. Boyer and J. Moore, "A fast string searching algorithm," *Commun. of the ACM*, pp. 762 – 772, Oct 1977.

[7] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. of the ACM*, pp. 333–340, 1975.

[8] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," University of Arizona, Tech. Rep. TR-94-17, May 1993.

[9] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on fpgas," in *FPGA*, 2004, pp. 223–232.

[10] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kon, and A. Thomas, "A hardware platform for network intrusion detection and prevention," in *NP*, 2004.

[11] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim, "A high performance NIDS using FPGA-based regular expression matching," in *SAC*, 2007, pp. 1187–1191.

[12] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *USENIX Security*, 2010, p. 8.

[13] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *Selected Areas in Communications, IEEE Journal on*, vol. 24, no. 10, pp. 1781–1792, Oct 2006.

[14] D. Pao, W. Lin, and B. Liu, "A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 2, pp. 10:1–10:27, Oct. 2010.

[15] C.-C. Chen and S.-D. Wang, "An efficient multicharacter transition string-matching engine based on the aho-corasick algorithm," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 25:1–25:22, Dec. 2013.

[16] D. P. Scarpazza, O. Villa, and F. Petrini, "Exact multi-pattern string matching on the cell/b.e. processor," in *CF*, 2008, pp. 33–42.

[17] D. L. Schuff, Y. R. Choe, and V. S. Pai, "Conservative vs. optimistic parallelization of stateful network intrusion detection," in *PPoPP*, 2007, pp. 138–139.

[18] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "MCA^2: multi-core architecture for mitigating complexity attacks," in *ANCS*, 2012, pp. 235–246.

[19] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *SIGCOMM*, 2006, pp. 339–350.

[20] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, Oct 2008.

[21] A. Bremler-Barr and Y. Koral, "Accelerating multi-patterns matching on compressed HTTP traffic," in *INFOCOM*, 2009, pp. 397–405.

[22] A. Bremler-Barr, S. T. David, D. Hay, and Y. Koral, "Decompression-free inspection: DPI for shared dictionary compression over HTTP." in *INFOCOM*, 2012, pp. 1987–1995.

[23] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *SIGCOMM*, 2000, pp. 87–95.

[24] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: findings and implications," in *SIGMETRICS*, 2009, pp. 37–48.

[25] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "Endre: An end-system redundancy elimination service for enterprises." in *NSDI*, 2010, pp. 419–432.

[26] E. Zohar, I. Cidon, and O. O. Mokryn, "The power of prediction: cloud bandwidth and cost reduction," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 86–97, Aug 2011.

[27] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., 1994.

[28] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun 2000.

[29] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy, "On the scale and performance of cooperative web proxy caching," in *SOSP*, 1999, pp. 16–31.

[30] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy, "Organization-based analysis of web-object sharing and caching," in *USITS*, 1999, pp. 3–3.

[31] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[32] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 181–194, Oct 1997.

[33] G. S. Shenoy, J. Tubella, and A. González, "Improving the performance efficiency of an ids by exploiting temporal locality in network traffic," in *MASCOTS*, 2012, pp. 439–448.

[34] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul 1970.

[35] F. Hao, M. Kodialam, and T. V. Lakshman, "Building high accuracy bloom filters using partitioned hashing," in *SIGMETRICS*, 2007, pp. 277–288.

[36] D. Levinthal, "Performance analysis guide for intel core i7 processor and Intel Xeon 5500 processors," http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[37] A. Bremler-Barr, Y. Harchol, and D. Hay, "Space-time tradeoffs in software-based deep packet inspection," in *HPSR*, 2011, pp. 1–8.