

Towards In-Network Semantic Analysis: A Case Study involving Spam Classification

Cyprien Gueyraud
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA

Nik Sultana
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA

Abstract—Analyzing free-form natural language expressions “in the network”—that is, on programmable switches and smart NICs—would enable packet-handling decisions that are based on the textual content of flows. This analysis would support richer, latency-critical data services that depend on language analysis—such as emergency response, misinformation classification, customer support, and query-answering applications.

But packet forwarding and processing decisions usually rely on simple analyses based on table look-ups that are keyed on well-defined (and usually fixed size) header fields. P4 is the state of the art domain-specific language for programming network equipment, but, to the best of our knowledge, analyzing free-form text using P4 has not yet been investigated. Although there is an increasing variety of P4-programmable commodity network hardware available, using P4 presents considerable technical challenges for text analysis since the language lacks loops and fractional datatypes.

This paper presents the first Bayesian spam classifier written in P4 and evaluates it using a standard dataset. The paper contributes techniques for the tokenization, analysis, and classification of free-form text using P4, and investigates trade-offs between classification accuracy and resource usage. It shows how classification accuracy can be tuned between 69.1% and 90.4%, and how resource usage can be reduced to 6% by trading-off accuracy. It uses the spam filtering use-case to motivate the need for more research into in-network text analysis to enable future “semantic analysis” applications in programmable networks.

Index Terms—Programmable Networking; Spam Filtering; P4; Software-Defined Networking

I. INTRODUCTION

Packet-handling decisions are based on simple analyses that can be carried out quickly to deliver good performance and simpler end-to-end engineering [1]. These analyses typically rely on table look-ups that are keyed on well-defined (and usually fixed size) header fields, such as address, protocol, and counter fields such as TTL (Time To Live) in the IP protocol. The simplicity of this processing ensures that packet handling can scale, and any expensive, “deep inspection” of packet is usually done selectively to avoid creating bottlenecks [2].

The availability of programmable switches [3], [4] and smart NICs [5], and the development of the P4 language [6] have spurred much interest into in-network computing applications. These applications go beyond forwarding [7] and target a variety of network functions [8].

In this paper, we use in-network programmability to analyze natural language messages that are contained in packet

payloads. This is an interesting but technically challenging problem, and this paper contributes techniques and open-source code for building such analyzers in P4.

This problem is interesting since the in-network analysis of free-form natural language expressions would enable making packet-handling decisions that are based on the textual content of flows. In turn, this would support richer, latency-critical data services that depend on language analysis—such as emergency response, misinformation classification, customer support, and query answering applications.

The problem is challenging however for a variety of reasons. In-network payload analysis is impossible on traffic that is encrypted end-to-end, such as SSL flows, but there are important situations that provide deployment opportunities for this idea. We envisage P4-based text analyzers being deployed on clear-text traffic in networks that lie behind SSL endpoints, or where the operator can selectively decrypt traffic, such as in corporate and datacenter networks [9]–[12].

This paper focuses on technical challenges that are intrinsic to processing packet payloads and analyzing natural language text using P4. It describes solutions to these problems:

- Analyzing the payloads of arbitrary packets in P4, a language that is primarily intended for header processing.
- Tokenizing payloads into words using P4’s primitive datatypes, and mitigating its lack of loop syntax.
- Approximating a division operator that is needed for arithmetic.
- Implementing probabilistic reasoning (needed for Bayesian filtering) using P4’s language primitives.

To our knowledge, there are no published descriptions that tackle these challenges together. As a demonstration of the applicability of the proposed techniques, the paper presents the first Bayesian spam classifier written in P4.

This Bayesian classifier draws upon all the techniques described in the paper and it is evaluated on a real-world dataset: the Enron Email Dataset [13]. The full evaluation is given in §V but the main take-aways are summarized next. This evaluation yielded the following findings on trade-offs between resource usage and classification accuracy:

- Spam classification accuracy was 69.1% when using word matching and 90.4% using Bayesian filtering. Both techniques were implemented in P4.

- Resource usage: word matching uses no registers and only 6% of the P4 lines of code used by Bayesian filtering. This indicates the trade-off between accuracy and resource cost.
- Analyzing only the first 200 bytes of a message leads to a drop of 20% in accuracy when using Bayesian filtering, but reduces resource usage by 22,800%. This indicates an effective compromise between loss of accuracy and preserving resources.

The system described in this paper is made available online as open source [14]. An accompanying Technical Report [15] goes into more detail on the algorithms used and their analysis. This paper makes the following contributions:

- §III adapts the *generative programming* approach [16] to develop a lightweight language tokenization approach that can be expressed in P4 for payload processing. Using this we develop two text classification techniques in P4: blocklist matching and Bayesian classification.
- §IV presents the first implementation of a spam filter in P4. It provides an example of a non-trivial P4 program in terms of both features and size. The core part of the implementation consists of around 1000 lines of P4, and the remaining part of the system (that depends on parameter choices that trade-off accuracy and resource-usage) are between 300 and 18630 lines of P4.
- §V presents an evaluation that uses a standard spam dataset [13], and discusses quantitative trade-offs for tuning resource usage against classification accuracy.

Related work is described in §VI and future work is described in §VII.

II. P4 FEATURES AND LIMITATIONS

P4 [6] is the most successful language for programmable network hardware. It is used to target both switches [4] and NICs [5], and used to implement various in-network programs [7], [8]. Read-write dataplane memory is made available as configurable-width *registers* in P4, and they are used for stateful P4 programs. There are two features that strongly limit P4's expressiveness in this paper's subject area: the lack of loops and fractional datatypes. In this paper, loops are needed to iterate over characters and words during tokenization, and to perform iterative calculations. Fractional datatypes are needed to represent probabilities in Bayesian reasoning.

In this paper, we overcome the first limitation by unrolling loops using macros, using a well-known technique from other programmable targets. For fractional datatypes we adapt a technique for fixed-point precision [17]. P4 hardware typically does not provide floating point instructions, so fractional calculations are done natively in P4 using our custom encoding.

III. LANGUAGE ANALYSIS TECHNIQUES AND THEIR MAPPING INTO P4

This section presents three techniques for natural language analysis in P4: language tokenization (§III-A) to split messages into words, word matching (§III-B) to use a blocklist,

and Bayesian filtering (§III-C) to build a probabilistic, up-dateable model of spam classification. All these techniques are combined in the P4 prototype that is described in §IV.

A. Tokenizing

The tokenization approach presented in this paper is parametrized as described below. Parameters are instantiated at compile time to generate P4 code—this is an example of the *generative programming* approach [16]. The parameter values can be changed to trade-off between resource usage and accuracy in the resulting P4 code.

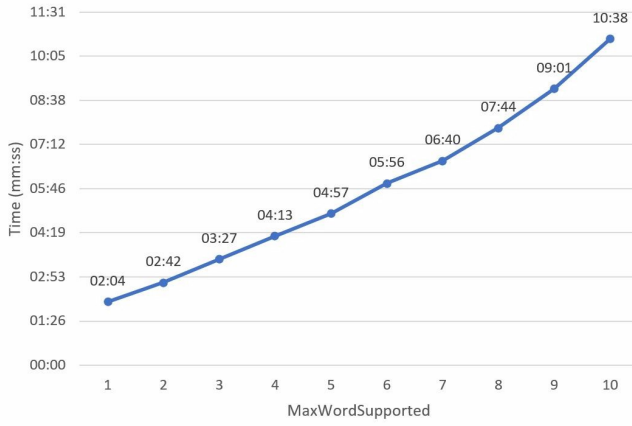
- *MaxN* is the maximum of number bytes of the message that the tokenization algorithm will analyze. The system will not tokenize words past this limit. *MaxN* is the length of analyzed prefix in the message. We set it to 200 and explain the rationale for this choice in §V-A and Figure 2).
- *MaxWordSupported* is the maximum number of words the system will analyze. The program maps the message's string of bytes into a string of words, up to *MaxWordSupported* in length. We observe that the choice of *MaxWordSupported* affects the program's compilation time. We set it to 10, and the choice for this is explained in Figure 1.
- *MaxWordLen* is the maximum size of a word for the creation of the word's Identification Number (*wID*). We set it to 10, and the rationale for this is explained in the Technical Report [15].

Each word is mapped to a unique Identification Number (*wID*). This codes the word up to a maximum number of letters; this maximum is the value of the constant *MaxWordLen*. Shorter words are padded with trailing zeroes, to reach the length of *MaxWordLen* characters. Words longer than *MaxWordLen* letters will be truncated to *MaxWordLen* characters. The Identification Number has to have enough bits to code every word we need. Each letter needs two decimal digits, so if we want to use *MaxWordLen* = 10, then the Identification Number needs to be set to 20 decimal digits (2×10).

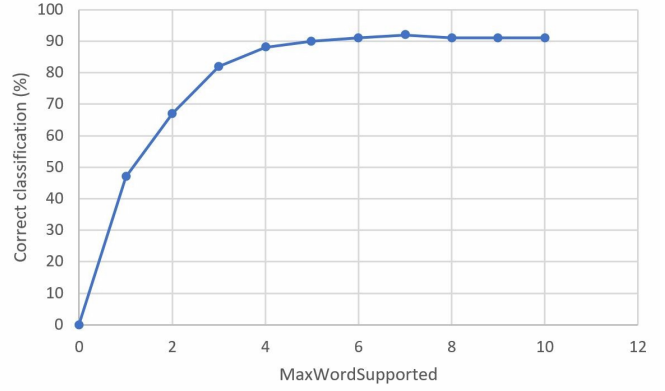
We set *MaxWordLen* = 10 to maximize the number of words analyzed: on a sample of 3000 words (of the most used words in the English language [18]) about 95% of words have a length of 10 or less. Adding more letters will not give more precision to the program and will increase the ID number very quickly: an additional letter means adding two digits to *wID* (multiplying *wID* by 100), and therefore adding seven bits for every Words/ID number.

B. Word matching

Word matching is a simple technique that recognizes specific words such as those in a *blocklist*. In this technique, the filter has in memory a list of words that are chosen based on a particular property. In our case, each of these words have a high likelihood of appearing in spam messages, and their presence increases the probability that a message is spam. When using this technique, a filter will tokenize the message



(a) Time of compilation as *MaxWordSupported* increases.



(b) Classification correctness as *MaxWordSupported* increases.

Fig. 1: Effects of varying *MaxWordSupported*.

as described in the previous section. It then counts the words in the message that also appear in the blocklist. If a threshold is reached in the number of matches, then the filter classifies the message as spam. Otherwise, the message is classified as “ham”—i.e., *not* spam.

To implement a blocklist in P4, after tokenizing a message as described in §III-A we match the tokenized words to a list that is stored in P4 registers. The words in the blocklist are represented by their respective Identification Numbers. The generative approach described earlier is used to unroll a loop to match each message word with the blocklist, and increment a count of matches. The length of the P4 program grows in proportion to the blocklist size and the message prefix that is analyzed. The P4 state (registers) required is linear in the length of the blocklist.

C. Bayesian filtering

A Bayesian filter [19] estimates the probability that a message is spam. It improves on word matching by casting the classification into a conditional probability function that is updated when new messages are encountered. The technique examines each word w_i of the message and calculates the probability as follows:

$$\Pr(S|w_i) = \frac{\Pr(w_i|S) \times \Pr(S)}{\Pr(w_i|S) \times \Pr(S) + \Pr(w_i|H) \times \Pr(H)} \quad (1)$$

where:

- $\Pr(S|w_i)$ is the probability that a message is a spam, with the word w_i in it;
- $\Pr(S)$ is the overall probability that any given message is spam;
- $\Pr(w_i|S)$ is the probability that the word w_i appears in spam messages;
- $\Pr(H)$ is the overall probability that any given message is *not* spam (it is referred to as **ham**);
- $\Pr(w_i|H)$ is the probability that the word w_i appears in ham messages.

In this formula, $\Pr(H) + \Pr(S) = 1$. Furthermore, before any test, there is no reason to assume that the next message is likely to be a spam or a ham. Thus we can assume $\Pr(H) = \Pr(S)$, which leads to the simplified formula:

$$\Pr(S|w_i) = \frac{\Pr(w_i|S)}{\Pr(w_i|S) + \Pr(w_i|H)} \quad (2)$$

To calculate the probability that a message is spam:

$$P = \frac{\sum_{i=1}^n (\Pr(S|w_i))}{n} \quad (3)$$

where:

- P is the probability that a message is a spam based on its content;
- n is the number of words in the message.

If we frequently encounter a word in messages that are classified as spam, then when we encounter this word in future messages we are more likely to classify those messages as spam. After analyzing a message, we update the probability of every word in that message: the filter updates its values to improve its accuracy.

In addition to the parameters described in §III-A, Bayesian filtering involves additional parameters:

- *limitSpam* is the probability above which we consider the message to be spam: if $P > \text{limitSpam}$, then the email is considered as spam. We set it to $\text{limitSpam} = 0.5$. We arrived at this value by carrying out experiments on how this choice affected false positives and false negatives. The Technical Report [15] explains the rationale for this choice.
- *MaxWordGroups* is the maximum number of *word groups* that will be supported by the generated P4 code. A *word group* includes words with similar *wID*: in our system, two words are *similar* if they have the same prefix up to a given length—4 letters in our prototype. Word groups factor the differences between words, and this reduces the number of registers that are allocated to store

the state related to the probability of each word. That state is needed to calculate the overall probability that a message is spam. The smaller the value of *MaxWordGroups* is, the fewer groups are supported, and thus the less memory is used by the generated P4 program. We set this parameter to 1840, as described in §V-A.

After tokenizing a message as described in §III-A, Bayesian filtering involves two additional steps. The first step finds the *wID* of the word and determines the corresponding register for that word. The second step involves storing each *wID*. Every word is assigned two counters saved in a register: the counters track the number of times the word was found on spam, and the number of times it was found in non-spam emails. These counters are used to discretize the Bayesian formula into P4 as explained in the Technical Report [15].

After finding the probability number of every word, we compute the final probability that the message is spam. If the probability $P > \text{limitSpam}$, the program classifies the message as spam.

The second step in our Bayesian approach updates registers and probability numbers of words in the message. Using the *wID* from the first step, we can reason about the words that were found in the message. If the message was classified as spam, we increment the number of times that we found this word in spam.

IV. PROTOTYPE

The prototype implements all the techniques described in §III and it is structured into two parts. The core part consists of around 1000 lines of P4 and contains functionality that is independent of the choices that are made for the parameters described in earlier. It contains the code that receives the packet, handles each character, creates an Identification Number (*wID*) for each word, and the skeleton code that performs calculations for Bayesian filtering. At various points the core program hands-over to parameter-generated code—for example, to determine how much of the message to tokenize.

The rest of the code is generated based on the choices for parameters described earlier sections. It includes the declaration and use of registers needed by the matching and Bayesian algorithms, and the unrolled loops that use and updates those registers. In the different instantiations we carried out of the parameters, this part of the prototype ranges in size from 300 lines for a basic word matching technique, to about 18630 lines for an advanced Bayesian filter.

V. EVALUATION

We evaluate three important properties of this approach through our prototype: **first**, we analyze the effect of “compressing” words with common prefixes (through *MaxWordGroups*) and how this effects resource usage in P4 (§V-A); **second**, we analyze the effect of analyzing a message prefix (*MaxN*) on classification accuracy and resource usage (§V-B); and **third**, we compare the accuracy and resource usage of word matching compared with Bayesian filtering (§V-C).

For the evaluation we used the Enron Email Dataset [13], which is a standard dataset for spam research. The full dataset contains more than 500,000 emails but since it contains raw messages with the names of senders and receivers, among other details, we used a subset of 32,625 emails from this dataset that was cleaned and adapted by others for research use. For the dataset we used, the average length of emails is 1482 characters, the median is 701 characters, the maximum length is 228,377 characters and the minimum length is 10 characters.

The experiments described in this section were carried out using P4’s BMv2 soft-switch on a Xeon E5-2678 v3 server clocked at 2.50GHz, having 128GB RAM, and running Ubuntu 22.04 LTS using kernel version 5.15.0-41-generic.

A. Effect of *MaxWordGroups* on program size

One challenge we faced during our work was the size of the generated P4 program; we therefore sought to optimize this. As explained earlier, for the Bayesian technique we use a fixed set of words, and every word was assigned two registers: one for counting the times that the word appears in spam, the second for the counting a word’s occurrences in non-spam messages. The size of these registers is set to accommodate the number of occurrences of the words that appear most frequently in the dataset that we use. For example, if we exclude words that do not need to be analyzed (i.e., *stop words* such as “a”, “is”, “the”... [20, Chapter 1]), the word “enron” appears 60849 times in non-spam messages. So we set registers to 16 bits to host this maximum.

If we uniquely represent the probability of each word, this would require a lot of state if the set of words is large. In the dataset we used, there are more than 75,000 different words, which means the creation of more than 150,000 registers. And this then entails an increase in the number of lines of the program—because P4 has no loops, we unwind the loops in our algorithm when mapping it to P4 code. Since we have to add at least 6 lines of P4 for each word, then to handle the 75,000 words in our dataset, the program would grow over 450,000 lines of P4 code.

To mitigate this growth, we create groups of similar words. We call these *word groups*. Words with the same prefix are put in the same group, and we allow them to share registers. This reduces accuracy but it also reduces the number of registers, P4 code, and therefore memory needed. We created groups with *wID* that share the first 8 digits. With *MaxWordLen* = 10, intervals can host up to about 300 million different *wID* values. This optimization leaves us with *MaxWordGroups* = 1840, and takes us from about 75,000 groups (i.e., one for each word) to only 1840 groups. This is 40× lower. This linear improvement results in an quadratic reduction in resources needed. This is explained further in the Technical Report [15].

B. Effect of *MaxN* on program size

Most text analysis require loops to iterate through a message, and the lack of loops in P4 makes analysis challenging to implement. To mitigate this, we only analyze the beginning

of the message, up to $MaxN$ letters. We found that the filter is more effective when it only scrutinizes the message prefix, as illustrated in Figure 2. That graph shows the number of misclassifications (number of false positives + number of false negatives) when the program analyzes only a part of the message, compared to the number of misclassifications if the program takes the entire email every time, defined as $\max(MaxN)$. $\max(MaxN)$ is the length of the longest message in the entire dataset. The graph shows 4 experiments, each relying a disjoint, random sample of 1000 emails from the dataset.

The y-axis represents the number of misclassifications with a value of $MaxN$ divided by the number of misclassifications with the maximum of $MaxN$ possible. In the dataset we use, $\max(MaxN) = 228,377$ —i.e., the largest email of the dataset has 228,377 characters. If the curve goes below $y = 1$ (the yellow line), it means that there are fewer misclassifications when using that value of $MaxN$ compared to if we use maximum value of $MaxN$. With $MaxN = 200$ letters, the program starts to be more effective than if it analyzes the entire message. That is, the algorithm has about 20% less wrong classifications than if it analyzes whole messages. So the program does not need to go through to entire message to have results we need. $MaxN$ goes hand in hand with $MaxWordSupported$: the more we increase $MaxN$ in order to analyse a bigger part of the message (like we can increase $MaxWordSupported$) the longer the P4 program that is generated. One line is added for each letter, thus if we want to analyze the entire message and set $MaxN$ to its maximum, the program will have 228,377 lines added instead of the 200 lines we would get if we set $MaxN = 200$. Knowing the core part of the program has about 1000 lines of code (§IV), analyzing whole messages would increase the size of the code by more than 22,800% (instead of adding only 20% of the size of the program with $MaxN = 200$). As a result of its increased size, the time needed to compile the P4 code will increase too. In Figure 1a we measured the increase in compilation time with the increase in message prefix analyzed.

C. Comparing filtering techniques

Figure 3 presents the results of experiments that were done using the prototype. We compare results from two versions of the program. The first version implements only word matching, and the second implements a Bayesian filter. The experiments used a random sample of 1000 emails from the Enron dataset that were separated in ten partitions of 100 random emails.

To evaluate the word matching technique we define a blocklist by first analyzing the words occurring in spam messages in the Enron dataset and ordered them into a list based on how frequently they appear in spam messages. We do the same for ham messages and form a second list. We remove words from the spam list if they appear in the top 100 words of the ham list—to avoid confusing frequently-occurring ham words with spam words. We then picked the 40 top words in the spam list, and that formed the blocklist. Using this list, we

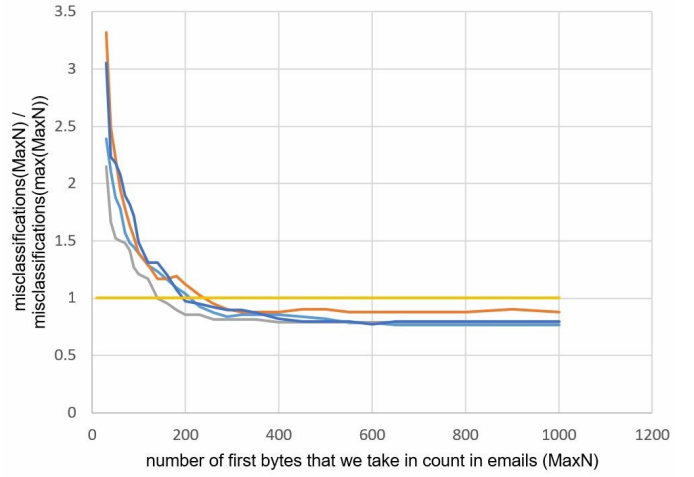


Fig. 2: Effectiveness of the Bayesian filter as $MaxN$ increases. $MaxN$ is the length of analyzed prefix of a message (described in §III-A). This graph is explained further in §V-B.

	Word Matching	Bayesian Filter
Lines of code	1223	19629
Size of P4 file (KB)	55	952
Number of Registers	0	3682
Total Register Bits	0	58912

TABLE I: Comparison of resources needed by each technique for $MaxWordGroups = 1840$. The composition of these groups is explained in §V-C.

obtained an average of 69.1% of correct classification. Correct classification means that a spam is classified as a spam, and a ham is classified as a ham. In other words, it discounts false positives and false negatives.

To evaluate the Bayesian filter we used $MaxWordGroups = 1840$ as explained in §V-A and generated code that performs the calculation described in §III-C. The experiments resulted in an average of 90.4% of correct classification. Tests are realized in exactly same samples than tests realized for the word matching—i.e., a sample of 1000 random emails of the Enron dataset, partitioned into ten sets of 100 emails.

Table I quantifies the difference in resource usage. As reported above, the Bayesian Filter gives better results than Word Matching. However, it requires a lot more resources: for the Bayesian filter, the program needs to store the state from previous analyses to calculate probabilities. This requires the use of registers and it also increases the size of P4 code, the size of files and the memory allocated to the program. In contrast, the matching technique does not need to store any state: it analyzes emails one by one, without needing to remember past classification.

VI. RELATED WORK

This paper focused on the challenges faced when using P4, but it shares similarities with other work on Deep Packet Inspection (DPI) and Intrusion Detection Systems (IDS). The

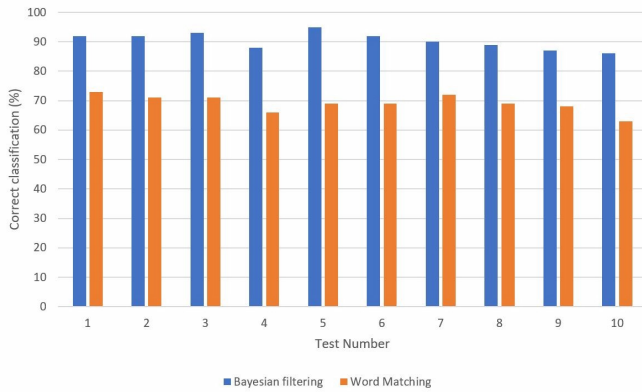


Fig. 3: Effectiveness of the two filters on 10 disjoint sets of 100 emails (i.e., 1000 messages total) from the Enron Dataset.

approach in this paper developed online probabilistic reasoning on a P4-programmable target, and it complements earlier work by Hypolite et al. [21] who used extensions beyond P4 to explore a NIC-based approach for IDS that is based on rule-matching against packet payloads. Phothilimthana et al. [22] explore offloading logic to a NIC for application acceleration. In comparison, this paper offloads a spam classifier to the network to benefit both clients and servers. In future work this can be improved further through client-side classifier integration to improve accuracy and resource-usage.

VII. CONCLUSIONS AND FUTURE WORK

This paper described the first implementation of in-network text classification using P4. It presented new techniques to implement a working prototype. A standard dataset was used to quantify the effectiveness of this approach, and measure trade-offs between accuracy and resource usage.

The prototype described in this paper was not designed to be performant. A next step involves creating a high-performance prototype on CPUs or FPGAs to measure the effects of parameter choices on latency or packet rate.

In addition to the techniques described in this paper, further optimization techniques can be developed to minimize the use of resources while maximizing accuracy. One idea for this involves using n-grams to compare against sequences of words, rather than individual words as done in this paper.

ACKNOWLEDGMENT

We thank Shivam Patel for help with implementing the division operation and setting up the evaluation infrastructure. We thank the anonymous reviewers for their helpful feedback. This work was supported by a Google Research Award and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0106. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of funders.

REFERENCES

- [1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, p. 277–288, nov 1984. [Online]. Available: <https://doi.org/10.1145/357401.357402>
- [2] N. Hua, H. Song, and T. V. Lakshman, "Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection," in *IEEE INFOCOM 2009*, 2009, pp. 415–423.
- [3] Broadcom, "Trident4 / BCM56880 Series," <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series>, accessed: 2022-06-10.
- [4] Intel, "Intel Tofino 3 Intelligent Fabric Processor Brief," <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html>, accessed: 2022-06-10.
- [5] Netronome Inc, "Agilio CX SmartNICs," <https://www.netronome.com/products/agilio-cx/>, 2016.
- [6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014. [Online]. Available: <https://doi.org/10.1145/2656877.2656890>
- [7] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker, "Thoughts on Load Distribution and the Role of Programmable Switches," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, p. 18–23, feb 2019. [Online]. Available: <https://doi.org/10.1145/3314212.3314216>
- [8] *NetCompute '18: Proceedings of the 2018 Morning Workshop on In-Network Computing*. New York, NY, USA: Association for Computing Machinery, 2018.
- [9] C. Bethea, G. Sheerin, J. Mace, R. King, G. Luo, and G. O'Connor, "SRE Workbook Chapter 11: Managing Load," <https://sre.google/workbook/managing-load/>, 2018, accessed: 2022-07-25.
- [10] G. Inc, "Encryption in Transit in Google Cloud," https://cloud.google.com/docs/security/encryption-in-transit#user_to_google_frontend_encryption, 2022, accessed: 2022-07-25.
- [11] C. Ghali, A. Stubblefield, E. Knapp, J. Li, B. Schmidt, and J. Boeuf, "Application Layer Transport Security," <https://cloud.google.com/docs/security/encryption-in-transit/application-layer-transport-security>, 2022, accessed: 2022-07-25.
- [12] A. Menezes and D. Stebila, "End-to-End Security: When Do We Have It?" *IEEE Security & Privacy*, vol. 19, no. 4, pp. 60–64, 2021.
- [13] W. W. Cohen, "Enron Email Dataset," <https://www.cs.cmu.edu/~enron/>, May 2015.
- [14] <https://github.com/CyprienGueyraud/Towards-In-Network-Semantic-Analysis-A-Case-Study-involving-Spam-Classification>.
- [15] C. Gueyraud and N. Sultana, "Towards In-Network Semantic Analysis," Technical Report: repository.iit.edu, Mar. 2023.
- [16] N. Stucki, A. Biboudis, and M. Odersky, "A Practical Unification of Multi-Stage Programming and Macros," *SIGPLAN Not.*, vol. 53, no. 9, p. 14–27, nov 2018. [Online]. Available: <https://doi.org/10.1145/3393934.3278139>
- [17] S. Patel, R. Atsatsang, K. M. Tichauer, M. H. L. S. Wang, J. B. Kowalkowski, and N. Sultana, "In-network fractional calculations using P4 for scientific computing workloads," in *Proceedings of the 5th International Workshop on P4 in Europe, EuroP4 2022, Rome, Italy, 9 December 2022*, M. Chiesa and S. L. Feibish, Eds. ACM, 2022, pp. 33–38. [Online]. Available: <https://doi.org/10.1145/3565475.3569083>
- [18] E. First, "3000 most common words in English," <https://www.ef.com/wwen/english-resources/english-vocabulary/top-3000-words/>, 2022.
- [19] P. Graham, "A Plan for Spam," <http://www.paulgraham.com/spam.html>, Aug. 2002, accessed: 2022-06-06.
- [20] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 2011.
- [21] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, and J. M. Smith, *DeepMatch: Practical Deep Packet Inspection in the Data Plane Using Network Processors*. New York, NY, USA: Association for Computing Machinery, 2020, p. 336–350. [Online]. Available: <https://doi.org/10.1145/3386367.3431290>
- [22] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 663–679. [Online]. Available: <https://www.usenix.org/conference/osdi18/presentation/photilimthana>