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

(Technical Report)*

Cyprien Gueyraud Nik Sultana

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

6th March 2023

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.

1 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 [33]. 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 [15].

The availability of programmable switches [6, 12, 19, 20] and smart NICs [26, 27, 38], and the development of the P4 language [4] have spurred much interest into in-network computing applications. These applications go beyond forwarding [23] and target a variety of network functions [2], including in-network consensus [8] and caching [21].

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 report extends our AnNet paper [13] with additional background details, problem analysis, and algorithm analysis.

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. Some of the challenges encountered are intrinsic to modern, secured networks. For example, in-network payload analysis is impossible on traffic that is encrypted end-to-end, such as SSL flows. Although this challenge is insurmountable in some settings, it does not arise in some significant scenarios. 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 [3, 10, 17, 24].

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 standard dataset: the Enron Email Dataset [7]. The full evaluation is given in §8 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 [1]. This paper makes the following contributions:

- §6 adapts the *generative programming* approach [34] 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.
- §7 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.
- §8 presents an evaluation that uses a standard spam dataset [7], and discusses quantitative trade-offs for tuning resource usage against classification accuracy.

Related work is described in §9 and future work is described in §10.

2 Why Semantic Forwarding Matters

Semantic forwarding involves taking into account the packet payload to make a forwarding decision. This is unlike normal packet forwarding, which is restricted to inspecting packet headers to make forwarding decisions based on simple matches, such as on a packet's destination address field. Semantic forwarding overlaps with Intrusion Detection Systems (IDS) [29, 32] but IDS carry out a very specific function, consisting of accepting or dropping packets by inspecting their contents, and escalating notifications. We envisage a more general form of semantic forwarding where packets and flows are forwarded to different hosts or networks depending on their contents.

Example applications of semantic forwarding include: **(1) Spam early-stage filter** to reduce the load on servers and clients—compared with a host-based spam filter running on a server or client, an in-network filter would analyze every cleartext message sent through the network, regardless of its intended destination, and have better visibility of communication patterns in a network. As an early-stage filter, it would redirect the message to another device that can analyze the message more closely. **(2) Emergency response** to detect the signs of an emergency by scanning social media messages in real time, and provide valuable information to law enforcement (such as the location and context of an emergency). **(3) Misinformation detection** by analyzing message contents according to sentiment patterns [39], and tagging, slowing or stopping the spread of misinformation based on similarity analysis. **(4) Protecting users from abuse** including trolling, bullying, and phishing. In-network text analysis could be used to detect, mark, or filter abusive content before it reaches or leaves certain networks, such as schools, datacenters, and campus and corporate networks.

3 Technical Challenges and Limitations

This section analyzes the various challenges that face in-network analysis of natural language text, before the next section starts developing an in-depth use-case.

1. **Programming abstractions**: in-network SDKs, languages and architectures—such as micro-C [37], P4 [4], and PISA [5] targets—usually offer only frugal libraries and datatypes, and often require explicit management of the memory hierachy. P4 does not support loops or fractional datatypes, and this makes it difficult to code non-trivial in-network applications.

2. **End-to-end confidentiality**: end-to-end encryption such as SSL stymies in-network analysis. Thus the in-network analysis of text must be used when the network can inspect network payloads. There are several contexts that allow this. For example, in a datacenter, this is usually possible after a "front end" service terminates end-to-end encryption such as SSL [3, 17].

3. **Vantage point**: if an on-switch P4 program is to analyze message content, then the message must be forwarded through that switch. Thus we must ensure that the analysis switch is suitably placed within the network topology, and that the forwarding tables are configured appropriately.

4. **Computational requirements**: the resources on current-day programmable switches and NICs are very scarce when compared to a datacenter-class servers. This limits the complexity of programs that can be deployed on switches and NICs. Although there has been progress on disaggregating in-network resources [36], these techniques are not yet mainstream and require manual effort.

5. **Traffic Volume and Latency**: higher traffic volumes require load-balancing for more complex DPI, which in turn complicates the deployment. The architecture of the programmable platform also impacts latency and the volume of traffic that can be processed—ranging from $\mathcal{O}(10)$ Gbps [35] to the current state of the art of $\mathcal{O}(10^4)$ Gbps [18].

6. **Message Length**: messages can be segmented because of packet fragmentation (because of a link's MTU) or because of a protocol's segment-size limit. This complicates the setup since it requires flow reconstruction [14].

Items 2-6 are standard challenges faced by intrusion detection systems, for which various mitigations have been developed over the years [25]. This paper focuses on item 1 which comes into particularly sharp focus when using P4 to write in-network applications for *non-trivial payload processing*.

4 Overview of In-Network Text Analysis

Semantic forwarding relies on in-network text analysis. This section describes the top-level structure of the approach presented in this paper, using spam classification as a real-world example for semantic forwarding.

To address challenge 1 from the previous section, we jointly apply two approaches: **(1)** We adapt ideas from *generative programming* [34] by generating P4 code according to explicit, compile-time parameters to P4-generating algorithms. These parameters specify the maximum word size and number of words to tokenize, the degree of factoring between messages and words that have common prefixes, and the tuning of prior probabilities for Bayesian filtering. By changing these parameters we can instantiate different filters that trade-off accuracy for resource-use. We will first encounter these parameters in tokenization (§6.1). **(2)** To carry out probabilistic calculations in P4, we reduce division and fractional calculations to addition and multiplication over unsigned integers. This will be explained in Figure 6 and Algorithm 2.

Challenges 2-6 from the previous section are circumvented in this work since we target deployments over unencrypted and datagram-bound messages as discussed in §3. This allows us to focus on P4 programmability issues in this paper.

In our implementation, once the P4 code is generated using the algorithms described in this paper, it is deployed in a software switch. For the Bayesian filter we run a *training phase* in order to initialize the filter. Spam messages are labelled in the Enron dataset used in the evaluation, thus providing ground truth. The filters in our prototype indicate their classification of a message by setting a bit in the message upon egress, which an evaluation script then examines to determine the classifier's accuracy.

5 P4 Features and Limitations

P4 [4] is the most successful language for programmable network hardware. It is used to target both switches [18] and NICs [27], and used to implement various in-network programs [2, 23]. 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 [28]. Not all P4-programmable hardware provides native floating point support, so fractional calculations are done in at the level of the P4 program using our custom encoding.

6 Language Analysis Techniques and their mapping into P4

This section presents three techniques for natural language analysis in P4: language tokenization (§6.1) to split messages into words, word matching (§6.2) to use a blocklist, and Bayesian filtering (§6.3) to build a probabilistic, updateable model of spam classification. All these techniques are combined in the P4 prototype that is described in §7.

6.1 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 [34]. 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 §8.1 and Figure 9.
- $MaxWordSupported$ 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 further below.

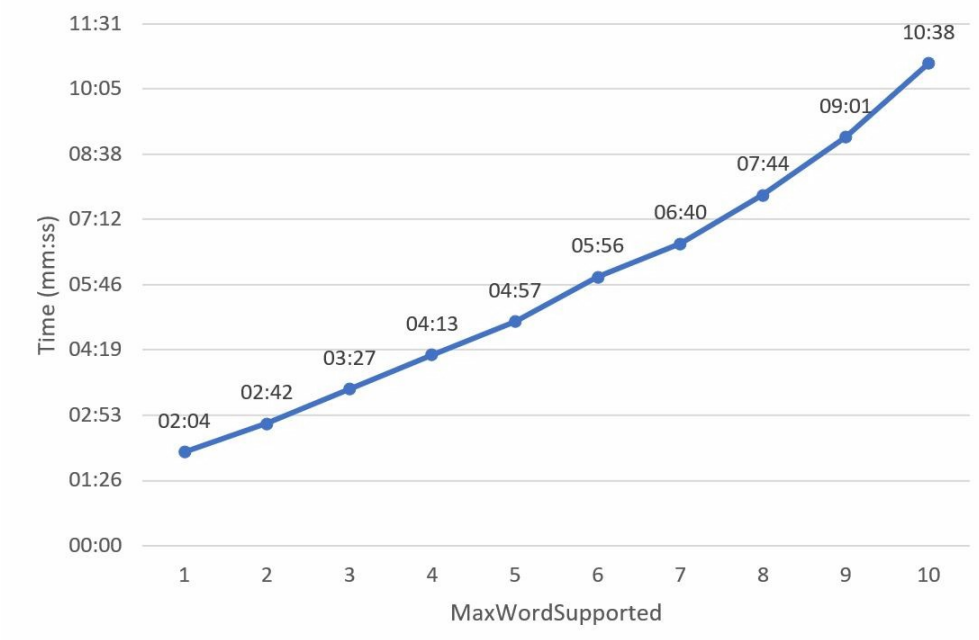
Algorithm 1 Tokenization of the message

Require: Message packet is received by the switch, and it extracts the $Mail$ pseudo-header from that packet.
Ensure: $ID[]$ is initialized with codes for message contents.

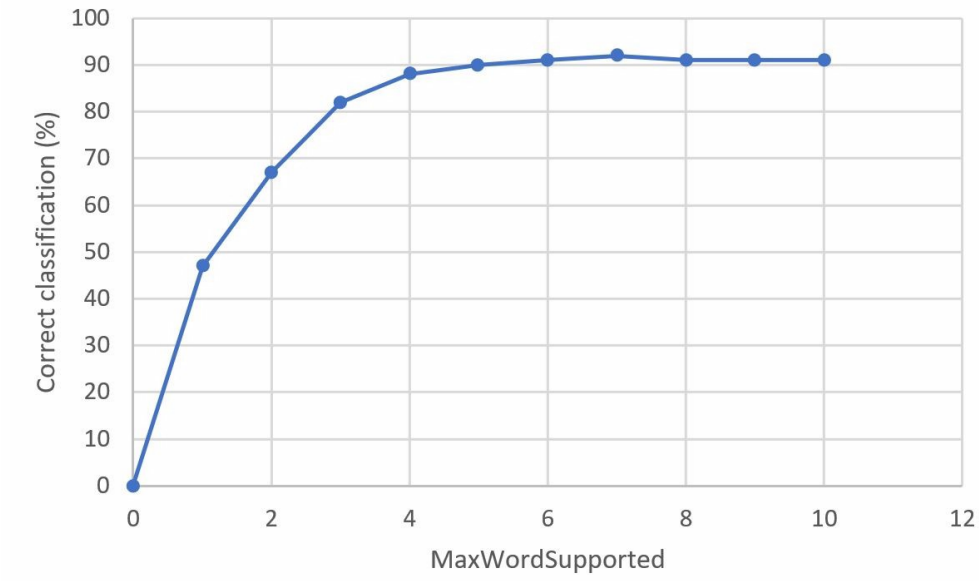
$N \leftarrow$ size($Mail$) ▷ $N \leq MaxN$
for $0 \leq x < N$ **do**
    $Letter[x] \leftarrow Mail.x$
end for
$WordOffset \leftarrow 0$
$WordIdx \leftarrow 0$
while $0 \leq LettIdx < N$ and $WordIdx < MaxWordSupported$ **do**
    ▷ Space is a separator between words.
    if $Letter[LettIdx]$ = Space **then**
        $wID \leftarrow 0$ ▷ ID that we will compute for the word.
        $WordLength \leftarrow LettIdx - WordOffset$
        ▷ Compute wID for the word as a function of each character that forms the word.
        for $0 \leq Z < WordLength$ **do**
            if $Letter[WordOffset + Z] =$ "a" **then**
                $wID = wID + 1 \times 2 \times (MaxWordLen - Z)$
            end if
            if $Letter[WordOffset + Z] =$ "b" **then**
                $wID = wID + 2 \times 2 \times (MaxWordLen - Z)$
            end if
            ⋮
            if $Letter[WordOffset + Z] =$ x **then**
                $wID = wID +$ idMap(x, Z)
                ▷ Where idMap(x, Z) is the linear mapping of character x, as done for "a" and "b" above.
            end if
        end for
        $ID[WordIdx] \leftarrow wID$
        ▷ Current words ends at the separator.
        $WordOffset \leftarrow LettIdx$
        ▷ Prepare for next word.
        $WordIdx \leftarrow WordIdx + 1$
    end if
    $LettIdx \leftarrow LettIdx + 1$
end while

Each word is mapped to a unique Identification Number (wID). This number is determined letter by letter as explained in Figure 5. This codes the word up to a maximum number of letters; this maximum is the value of the constant $MaxWordLen$ in Algorithm 1. Shorter words are padded with trailing zeroes, to reach the length of $MaxWordLen$ characters. Words longer than $MaxWordLen$ letters are truncated to $MaxWordLen$ characters. The Identification Number has to have enough bits to code every word we need. Each letter needs two decimal digits (Figure 5), so if we want to use $MaxWordLen = 10$, then the Identification Number needs to be set to 20 decimal digits (2×10). The number of representable words will be less than 2.7×10^{19}, and 66 bits will be sufficient to represent all possible words in this scheme.

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 [9]) about 95% of words have a length of 10 or less (Fig. 2). Adding more letters



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



(b) Classification correctness as $MaxWordSupported$ increases.

Figure 1: Effects of varying $MaxWordSupported$.

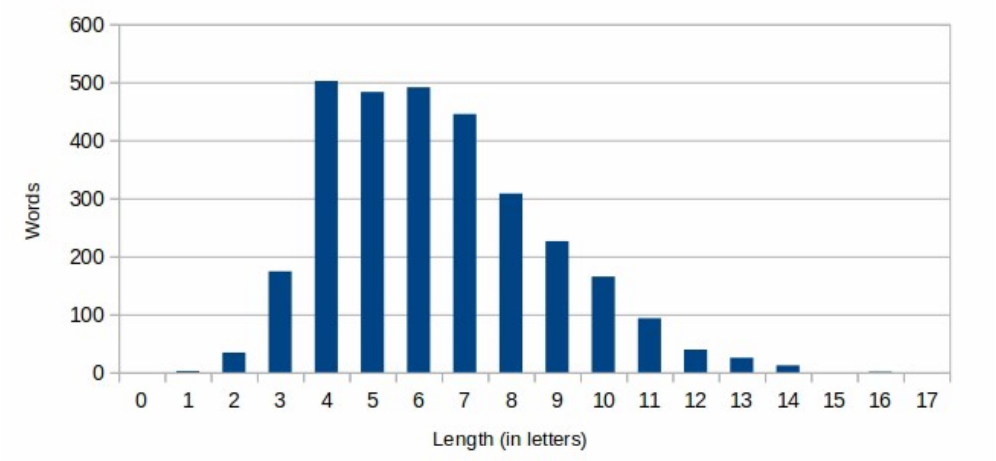


Figure 2: Distribution of the length of English words, based on a sample of 3000 of the most commonly-used words [9].

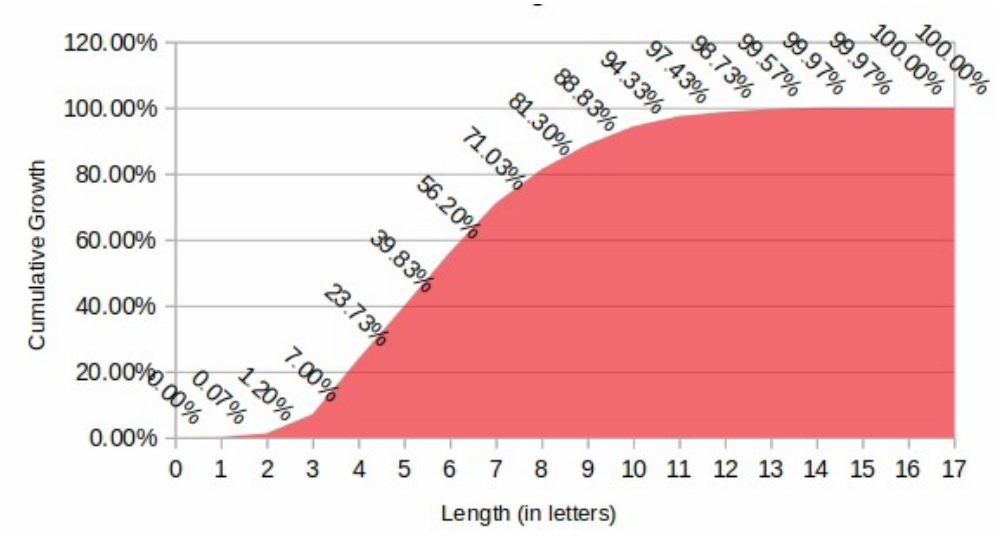


Figure 3: Number of words that are less than a given length.

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. If we want to change it, we have to change $MaxWordLen$ in the tokenization algorithm (Alg. 1), but also the list of every words' group has to be modified to increase or decrease the length of their wID.

6.2 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 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 §6.1 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.

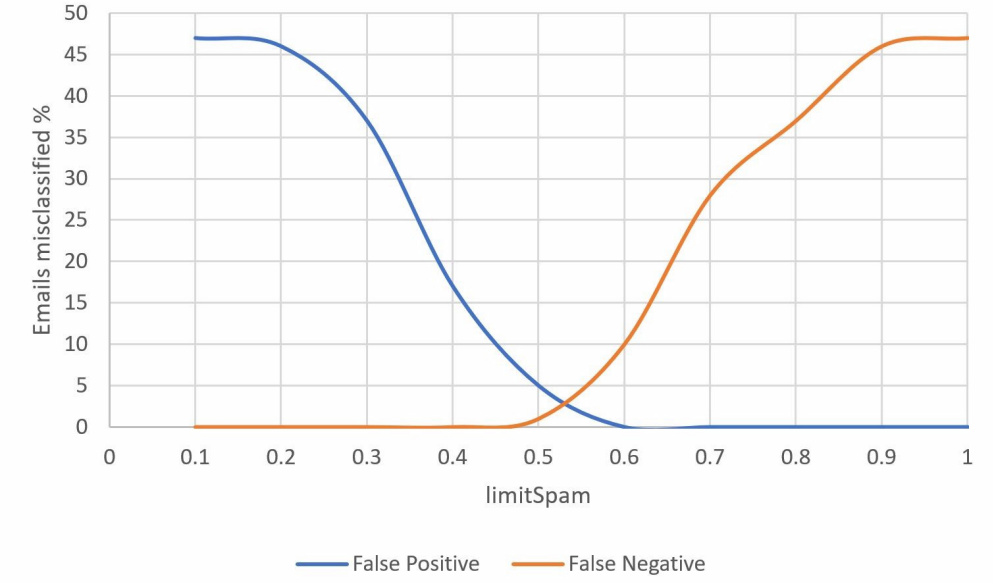


Figure 4: Effects of varying $limitSpam$

a	b	c	d	e	f	...	y	z
01	02	03	04	05	06	...	25	26

(a) Mapping from letters to codes.

e	x	a	m	p	l	e			
05	24	01	13	16	12	05	00	00	00

= 05240113161205000000

(b) Example mapping.

Figure 5: Calculating a word's Identification Number (wID).

If there is a match with the code of a word in the list, the P4 algorithm will increase the message's weighting as spam. 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. Once a threshold is reached, the message is classified as being spam. 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.

6.3 Bayesian filtering

A Bayesian filter [11] 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:

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

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:

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

To calculate the probability that a message is spam:

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

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 are 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 §6.1, Bayesian filtering involves additional parameters:

- $limitSpam$ is the probability above which we consider the message to be spam: if $P > limitSpam$, then the email is considered as spam. We set it to $limitSpam = 0.5$. We arrived at this value by carrying out experiments on how this choice affected false positives and false negatives. Figure 4 shows the rationale for this choice, based on a random sample of 100 emails from the dataset [7].
- $MaxWordGroups$ 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 §8.1.

Recall that $\Pr(S|w_i)$ represents the probability that a message containing the word w_i is spam. Using Formula 2 and Algorithm 3, we can calculate this as follows:

$$\Pr(S|w_i) = \frac{\Pr(w_i|S)}{\Pr(w_i|S) + \Pr(w_i|H)} \text{(using Formula 2)} \quad = \frac{\frac{SpamCount_i}{totalSpam}}{\frac{SpamCount_i}{totalSpam} + \frac{HamCount_i}{totalHam}} \text{(using Algorithm 3)}$$

This probability needs to be updated each time we receive a new message and classify it as spam or ham. Carrying out this update involves changing the contents of registers according to the Bayesian formula, **but this is difficult in P4 because of the lack of suitable datatypes**. We therefore derived a reduction of this approach that is both mathematically adequate *and* expressible in P4.

To help explain our approach, we contrast it with the traditional Bayesian calculation. If we obtain another spam message and it contains this word w_i, then we update the probability $\Pr(S|w_i)$ to a new probability $\Pr(S|w_i)^{+1}$ for the next calculation. We identify two ways of updating $\Pr(S|w_i)$:

1 (Traditional approach) : If we have the value of $\Pr(S|w_i)$ saved in registers, then we need to find $A, B \in \mathbb{R}$ such that $\Pr(S|w_i)^{+1}$ is a function of $\Pr(S|w_i)$:

$$\Pr(S|w_i)^{+1} = \quad A \times \Pr(S|w_i) + B$$

2 (Our approach) : If we saved counters in registers, then we just have to rely on incrementation and division to calculate $\Pr(S|w_i)^{+1}$:

$$\Pr(S|w_i)^{+1} = \quad \frac{\frac{SpamCount_i + 1}{totalSpam + 1}}{\frac{SpamCount_i + 1}{totalSpam + 1} + \frac{HamCount_i}{totalHam}} \quad \text{; we increment specific counters}$$

Since P4 only provides finite-precision integers, approach **1** cannot be applied directly—we would first need to implement fractional numerals and reason about adequate precision to avoid underflow. For example, if the update involves multiplying by 0.001 but we can only support precision of up to 0.01, the update would not take place as planned. That is why we chose to implement solution **2** in the prototype—based on incrementation and division (Fig. 2)—and saved counters in registers.

Figure 6: Why we saved counters in registers instead of probabilities.

After tokenizing a message as described in §6.1, 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 Figure 6.

When the word is found, with these two counters, a calculation (as an ingress action) can give the Bayesian probability: the probability that the email is a spam with this word in it. The reason why we saved in registers counters instead of directly probabilities is explained in Figure 6. After finding the probability number of every word (Alg. 2 and Alg. 3), we compute the final probability that the message is spam. If the probability $P > limitSpam$, the program classifies the message as spam.

We determined $limitSpam$ using an experiment with 100 random emails of the dataset. Figure 4 show results of this test: the first curve represent shows the number of False Positives the program had in the test, and the second represent the number of False Negatives. When $limitSpam$ is too low, the program finds a high number of False Positives, and when $limitSpam$ is too high, the program finds a high number of False Negatives. The ideal is to set $limitSpam$ to a value where both categories are low. It is why we chose $limitSpam = 0.5$.

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.

Algorithm 2 Division with P4

Require: $A \leq B$ and $B > 0$ and $A \geq 0$.
Ensure: The value of $\frac{A}{B}$ is expressed as a percentage.

```
if A = 0 then return 0
else
    if A = B then return 1
    else
        for 0 < x < 100 do
            if (B × (x − 1)) ≤ (A × 100) ≤ (B × (x + 1)) then return x
                ▷ Demonstration Formula 7
            end if
        end for
    end if
end if
```

$$
\begin{aligned}
&\text{Let } X \in \mathbb{R} \text{ and } A, B, \lfloor X \rfloor \in \mathbb{N}, B > 0 \\
&\frac{A}{B} = X\% \quad \Leftrightarrow \frac{A}{B} = \frac{X}{100} \quad \Leftrightarrow \frac{A}{B} \approx \frac{\lfloor X \rfloor}{100} \\
\Leftrightarrow &(A \times 100) \approx (B \times \lfloor X \rfloor) \\
\Leftrightarrow &(B \times (\lfloor X \rfloor - 1)) \leq (A \times 100) \text{ and} \\
&(A \times 100) \leq (B \times (\lfloor X \rfloor + 1))
\end{aligned}
$$

Figure 7: Division is approximated using simpler operators.

Algorithm 3 Probability of spam conditional on word w_i

Ensure: Calculate $\Pr(S|w_i)$: the probability that the message is a spam with the word w_i in it.

$\Pr(w_i|S) \leftarrow \dfrac{SpamCount_i}{totalSpam}$
▷ $SpamCount_i$: counts the occurrences of w_i in spam.
▷ $totalSpam$ is the total number of spam messages we found in all messages analysed.
$\Pr(w_i|H) \leftarrow \dfrac{HamCount_i}{totalHam}$
▷ $HamCount_i$: counts the occurrences of w_i in ham.
▷ $totalHam$ is the total number of ham messages we found in all messages analysed.
$\Pr(S|w_i) \leftarrow \dfrac{\Pr(w_i|S)}{\Pr(w_i|S) + \Pr(w_i|H)}$

Algorithm 4 Finding the word analysed

Require: $idNumber$ of the word analysed (A)
Ensure: Find registers of the word analysed

```
for Word_B in All_ID_Word[] do
    counterSpam ← register.word_B.1
    counterHam ← register.word_B.2
end for
```

Algorithm 5 Update the statistics for each word.

Ensure: The registers and counters that encode the spam and ham probabilities for each word are updated.

for $0 < i < MaxWordSupported$ **do**
  for $0 < j < MaxWordGroups$ **do**
    if $wID_i = wID_j$ **then**
      $allProba[i] \leftarrow \Pr(S|w_j)$
      ▷ The explanation of how $\Pr(S|w_j)$ is
      ▷ calculated is explained in Algorithm 3.
    end if
  end for
end for
$P \leftarrow avg(allProba)$
if $P > limitSpam$ **then**
  ▷ $limitSpam$ is the probability above which we
  ▷ consider the message to be spam.
  for $0 < x < MaxWordSupported$ **do**
    for $0 < y < MaxWordGroups$ **do**
      if $wID_x = wID_y$ **then**
        $counterSpam_y \leftarrow counterSpam_y + 1$
      end if
    end for
  end for
  $totalSpam \leftarrow totalSpam + 1$
else
  for $0 < x < MaxWordSupported$ **do**
    for $0 < y < MaxWordGroups$ **do**
      if $wID_x = wID_y$ **then**
        $counterHam_y \leftarrow counterHam_y + 1$
      end if
    end for
  end for
  $totalHam \leftarrow totalHam + 1$
end if

7 Prototype

The prototype implements all the techniques described in §6 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 most of the header of the program, describing the structure of the packet. It also includes 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.

8 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 (§8.1); **second**, we analyze the effect of analying a message prefix ($MaxN$) on classification accuracy and resource usage (§8.2); and **third**, we compare the accuracy and resource usage of word matching compared with Bayesian filtering (§8.3).

For the evaluation we used the Enron Email Dataset [7], 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.

8.1 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"... [31, 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. (See Figure 8 for an explanation of the resources needed.) 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. To have different wID numbers share registers, we form an interval of ID numbers and check against that interval. To do this, we update Algorithm 5 on lines 3, 16, and 25 as follows: change "**if** $wID_x = wID_y$" into "**if** $wID_x \in [\min(wID_y); \max(wID_y)]$". In the updated code, $\min(wID_y)$ is the first wID of the group's interval and $\max(wID_y)$ is the last wID of the interval. 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

The size of the Bayesian filter's P4 code grows quadratically with the number of word groups it can support because (1) it compares every word group with every other word group, and (2) the loops are unrolled, leading to a quadratic number of lines of P4. Since P4 lacks loops, the running time of the filter is linear in the number of P4 lines, thus making it quadratic in the number of word groups supported by the filter.
We now unpack more details about the coefficients for the quadratic growth in P4 lines of code. This is structured around the two core loops in Algorithm 5. Recall that $MaxWordGroups$ is the number of supported work groups by this instance. The first loop generates $MaxWordGroups \times 3$ lines per group, broken down as follows:

$$\begin{pmatrix} 1 & \text{check if there is a match between the two words} \\ +1 & \text{open the register with the counter of spam and take it} \\ +1 & \text{open the register with the counter of ham and take it} \end{pmatrix} \text{; Algorithm 5 lines 1-9}$$

Using these counters, the program can now calculate the Bayesian probabilities (Algorithm 3). Then the message is classified as spam if $P > limitSpam$—the probability above which we consider the message to be a spam—or as ham in the other case (Algorithm 5 lines 10-13). The secondlLoop generates $MaxWordGroups \times 3$ lines =

$$\begin{pmatrix} 1 & \text{check if there is a match between the two words} \\ +1 & \text{open the register with the counter of spam, increment if spam} \\ +1 & \text{open the register with the counter of ham, increment if ham} \end{pmatrix} \text{; Algorithm 5 lines 14-20}$$

Figure 8: Number of P4 lines needed for each word for the Bayesian filter.

for each word) to only 1840 groups. This is $40\times$ lower. This linear improvement results in an quadratic reduction in resources needed (Fig. 8).

8.2 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. Our experiments showed that the filter is more effective when it only scrutinizes the message prefix, as illustrated in Figure 9. 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. Each line in the graph represents an experiment: there are four experiments in the graph, 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 (§7), 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.

8.3 Comparing filtering techniques

Figure 10 presents the results of experiments done using the prototype. We compare results from two versions of the program, the first implementing only word matching, and the second implementing a Bayesian filter. The experiments

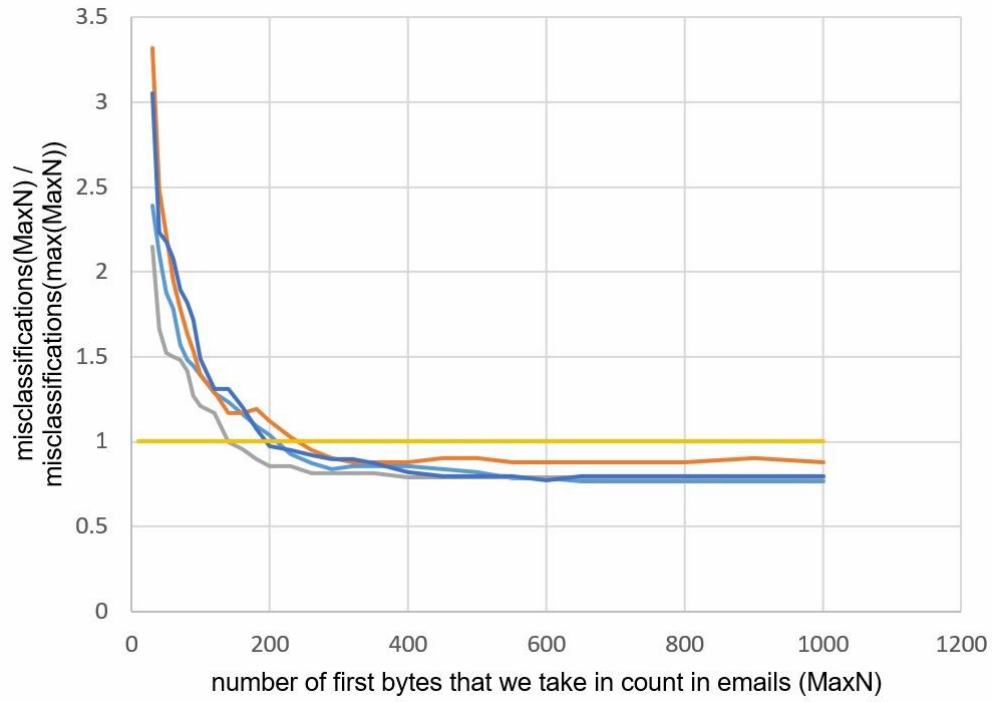


Figure 9: Effectiveness of the Bayesian filter as $MaxN$ increases. $MaxN$ is the length of analyzed prefix of a message (described in §6.1). This graph is explained further in §8.2.

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 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 §8.1 and generated code that performs the calculation described in §6.3. The Bayesian filter follows the process we described in section 7. We first implement registers in order to find every word and calculate Bayesian probabilities. As explained in section 8.1, to reduce the size of the program and the number of registers, we group together words that start the same. It means that similar words will share same registers. We end with $MaxWordGroups = 1840$ as explained section 8.1. 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 1 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.

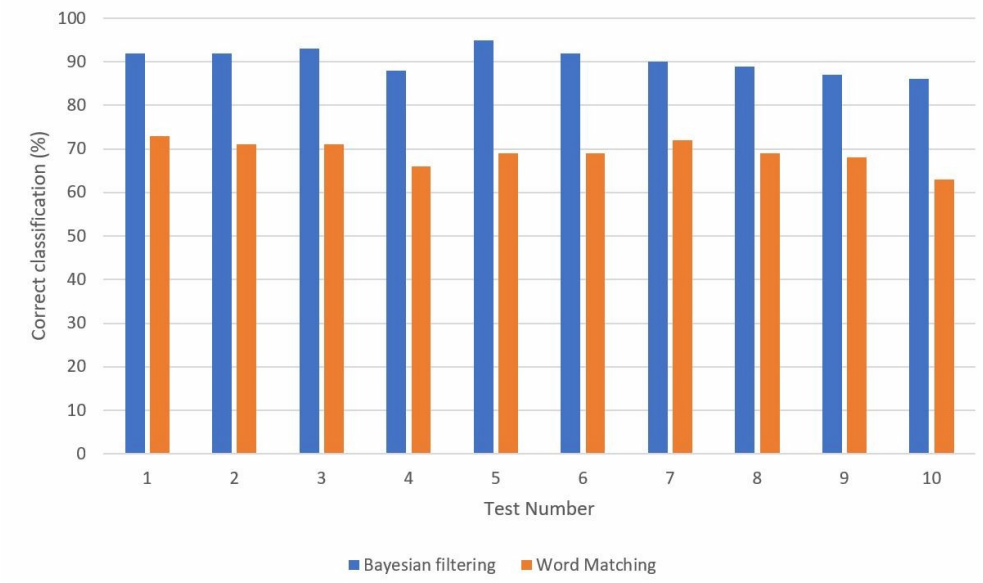


Figure 10: Effectiveness of the two filters on 10 disjoint sets of 100 emails (i.e., 1000 messages total) from the Enron Dataset.

	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 1: Comparison of resources needed by each technique for $MaxWordGroups = 1840$. The composition of these groups is explained in §8.3.

9 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 approach in this paper developed online probabilistic reasoning on a P4-programmable target, and it complements earlier work by Hypolite et al. [16] 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. [30] 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 [22] to improve accuracy and resource-usage: that is, we would afford lower in-network accuracy as a first filtering pass before reaching client-side classifiers which themselves have scarce resources—such as if they are executing on a mobile phone with scarce battery life.

10 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] `https://github.com/CyprienGueyraud/Towards-In-Network-\Semantic-Analysis-A-Case-Study-involving-Spam-Classification`.

[2] *NetCompute '18: Proceedings of the 2018 Morning Workshop on In-Network Computing*, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Cooper Bethea, Gráinne Sheerin, Jennifer Mace, , Ruth King, Gary Luo, and Gary O'Connor. SRE Workbook Chapter 11: Managing Load. `https://sre.google/workbook/managing-load/`, 2018. Accessed: 2022-07-25.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[6] Broadcom. Trident4 / BCM56880 Series. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series`. Accessed: 2022-06-10.

[7] William W. Cohen. Enron Email Dataset. https://www.cs.cmu.edu/~enron/, May 2015.

[8] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, may 2016.

[9] Education First. 3000 most common words in English. https://www.ef.com/wwen/english-resources/english-vocabulary/top-3000-words/, 2022.

[10] Cesar Ghali, Adam Stubblefield, Ed Knapp, Jiangtao Li, Benedikt Schmidt, and Julien Boeuf. Application Layer Transport Security. https://cloud.google.com/docs/security/encryption-in-transit/application-layer-transport-security, 2022. Accessed: 2022-07-25.

[11] Paul Graham. A Plan for Spam. http://www.paulgraham.com/spam.html, August 2002. Accessed: 2022-06-06.

[12] Enterprise Strategy Grou. Broadcom Trident 3 Platform Performance Analysis. https://docs.broadcom.com/doc/12395356, May 2019. Accessed: 2022-06-10.

[13] Cyprien Gueyraud and Nik Sultana. Towards In-Network Semantic Analysis: A Case Study involving Spam Classification. In *8th IEEE/IFIP International Workshop on Analytics for Network and Service Management (AnNet 2023)*. IEEE, 2023.

[14] P. Gupta and N. McKeown. Algorithms for Packet Classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2):24–32, mar 2001.

[15] N. Hua, H. Song, and T. V. Lakshman. Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection. In *IEEE INFOCOM 2009*, pages 415–423, 2009.

[16] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. *DeepMatch: Practical Deep Packet Inspection in the Data Plane Using Network Processors*, page 336–350. Association for Computing Machinery, New York, NY, USA, 2020.

[17] Google Inc. Encryption in Transit in Google Cloud. https://cloud.google.com/docs/security/encryption-in-transit#user_to_google_front_end_encryption, 2022. Accessed: 2022-07-25.

[18] 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.

[19] Intel. Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: 2022-06-10.

[20] Intel. Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. Accessed: 2022-06-10.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Bin Liang, Miaoqiang Su, Wei You, Wenchang Shi, and Gang Yang. Cracking Classifiers for Evasion: A Case Study on the Google's Phishing Pages Filter. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 345–356, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[23] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on Load Distribution and the Role of Programmable Switches. *SIGCOMM Comput. Commun. Rev.*, 49(1):18–23, feb 2019.

[24] Alfred Menezes and Douglas Stebila. End-to-End Security: When Do We Have It? *IEEE Security & Privacy*, 19(4):60–64, 2021.

[25] Aleksandar Milenkoski, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Bryan D. Payne. Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices. *ACM Comput. Surv.*, 48(1), sep 2015.

[26] Netcope Technologies. Netcope P4. `https://www.netcope.com/en/products/netcopep4`, June 2017.

[27] Netronome Inc. Agilio CX SmartNICs. `https://www.netronome.com/products/agilio-cx/`, 2016.

[28] Shivam Patel, Rigden Atsatsang, Kenneth M. Tichauer, Michael H. L. S. Wang, James B. Kowalkowski, and Nik Sultana. In-network fractional calculations using P4 for scientific computing workloads. In Marco Chiesa and Shir Landau Feibish, editors, *Proceedings of the 5th International Workshop on P4 in Europe, EuroP4 2022, Rome, Italy, 9 December 2022*, pages 33–38. ACM, 2022.

[29] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23):2435–2463, 1999.

[30] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[31] Anand Rajaraman and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[32] Martin Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[33] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, nov 1984.

[34] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. A Practical Unification of Multi-Stage Programming and Macros. *SIGPLAN Not.*, 53(9):14–27, nov 2018.

[35] Jinshu Su, Shuhui Chen, Biao Han, Chengcheng Xu, and Xin Wang. A 60Gbps DPI Prototype Based on Memory-Centric FPGA. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 627–628, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592. USENIX Association, April 2021.

[37] Stuart Wray. The Joy of Micro-C, 2014.

[38] Xilinx Inc. Xilinx U25. `https://www.xilinx.com/products/boards-and-kits/alveo/u25.html`.

[39] Xinyi Zhou and Reza Zafarani. A Survey of Fake News: Fundamental Theories, Detection Methods, and Opportunities. *ACM Comput. Surv.*, 53(5), sep 2020.