Wirebrush4SPAM: a novel framework for improving efficiency on spam filtering services

N. Pérez-Díaz, D. Ruano-Ordas, F. Fdez-Riverola and J. R. Méndez*,†

Department of Computer Science, University of Vigo, ESEI, Campus As Lagoas, 32004 Ourense, Spain

SUMMARY

This paper introduces Wirebrush4SPAM, a plug-in-based C framework specifically designed for the development of fast spam filters by assembling different antispam schemes and techniques. Wirebrush4SPAM can be used to (i) build, execute and deploy simple spam filters and (ii) develop new techniques that can be easily combined and tested to achieve more accurate antispam models. To construct custom filters, programmers should manage three key concepts: filtering functions, parsers and event listeners. The main features of Wirebrush4SPAM include (i) a plug-in-based design, (ii) cache support for developing new plugins, (iii) a smart filter evaluation heuristic for improving filter execution, (iv) configurable rule scheduling and (v) support for domain specific rules. Moreover, Wirebrush4SPAM is 10 times faster than SpamAssassin, which stands for the most popular and highly extensible framework for spam filtering. Wirebrush4SPAM is an open-source project licensed under the terms of GNU lesser general public license and both source code and documentation are publicly available at http://www.wb4spam.org/. Copyright © 2012 John Wiley & Sons, Ltd.

Received 27 June 2011; Revised 9 December 2011; Accepted 23 May 2012

KEY WORDS: C framework; plug-in-based development; rule-based evaluation; efficient spam filtering; learning schemes

1. INTRODUCTION

Given the importance of e-mail communication and their low cost, it is often used to distribute different kinds of advertisements (most of them unauthorized). Malicious usage of electronic data distribution and all other forms of unsolicited communications, also designated as spam, have reached scales never seen before [1]. As an example, Yahoo! Inc. is one of the greatest e-mail provider companies supporting more than 270 million user accounts [2]. If only half of these users receive one message per day, Yahoo! servers should analyse and filter more than 1,500 messages per second. In this context, spam message detection and removal requires great computational effort and carries important economic costs [3].

Nowadays, the most popular e-mail filter middleware is SpamAssassin [4], which provides the basis for other relevant products used in the spam filtering industry like McAfee SpamKiller [5] or Symantec Brighmail [6]. Despite SpamAssassin being a great tool for developing and deploying spam filters, the filters’ execution is very slow partly because SpamAssassin is written in object-oriented Perl. Typical SpamAssassin filter distributions (e.g. the default filter included in Debian or Ubuntu GNU/Linux) spend an average time of 1 s to analyse and filter a message in a 2 × Intel Xeon CPU E5520 2.27 GHz (Supermicro Computer Inc., San José, California, USA) (4 cores by CPU) configuration. Therefore, the hardware required to
achieve the filtering throughput needed by leading companies introduces high costs in their e-mail infrastructure.

During the last years, the number of cores included in computer processors has multiplied augmenting parallel processing capabilities. However, CPU clock frequencies have not increased accordingly. In this situation, achieving execution time reductions by using single-thread software seems very difficult. Therefore, the optimisation of existing applications should be done by an intelligent exploitation of multithread and multiprocessing capabilities included in the newest computers. This is one of the most relevant problems found when using SpamAssassin filter in current multi-core processors, in which one of the cores exhibits high activity (near 100%) while the rest are idle. Actually, SpamAssassin only supports multithreading via the spamd daemon when classifying more than one message at the same time.

However, SpamAssassin brought to the spam filtering domain two great functionalities: (i) the possibility of modelling the filter operation and (ii) the facility for distributing its behaviour by using rules. These two essential characteristics have been widely exploited to develop advanced spam filtering services by leading companies and other small and medium enterprises. In this context, the increment of spam deliveries experimented during the last years together with the existence of SpamAssassin software have guaranteed the profitability in this young business.

Ironically, the massive increment of spam deliveries threatens to fold this successful business model and the whole Internet e-mail service. The message delivery lag introduced by advanced antispam techniques is increasing because of the complexity of software filters. To alleviate the time required for message delivery, filtering should be carried out during simple mail transfer protocol transaction. In such a situation, the time spent to filter the target message have to be lower than Transmission Control Protocol timeouts, otherwise the message could be lost. However, actual SpamAssassin distributions fail to effectively support this filtering scheme. Therefore, a complete solution to decrease the time used for both e-mail classification and delivery is essential for efficiently identifying spam messages during simple mail transfer protocol transaction. In such a situation, both spam filters and MTAs (mail transfer agents) should be improved [7]. In this domain, the loss of a legitimate e-mail caused by poor management of resources during the filtering is not acceptable, so speed, CPU consumption and computational overhead issues have great impact on spam filtering.

To cope with this situation, we have detected four different bottlenecks in SpamAssassin software that should be addressed to improve general message filtering throughput: (i) the programming language, (ii) the absence of caches for intermediate results, (iii) the execution of rules in contexts that makes them irrelevant and (iv) the lack of an appropriate threading/multiprocessing scheme. Moreover, based on our previous experience in the spam filtering domain [8, 9], we have identified some interesting features for effectively deploying filters that include both domain-specific and sufficient condition rules.

In this work we present our novel Wirebrush4SPAM platform. Wirebrush4SPAM design and functionalities are initially inspired in the SpamAssassin framework, but it has been written from scratch in C language including the following characteristics: (i) a plug-in-based design, (ii) cache support for developing new plug-ins, (iii) an SFE (smart filter evaluation) heuristic for improving filter execution, (iv) configurable rule scheduling and (v) support for domain specific rules.

The rest of the paper is structured as follows: Section 2 introduces the current status of the SpamAssassin framework. Section 3 describes the Wirebrush4SPAM project evidencing the main differences and improvements with respect to SpamAssassin. Section 4 shows how to build Wirebrush4SPAM filters while Section 5 presents and discusses the results of an empirical efficiency comparison between Wirebrush4SPAM and SpamAssassin. Finally, Section 6 summarizes the main conclusions extracted from this work and outlines future research lines.

2. THE SPAMASSASSIN FRAMEWORK

Nowadays, SpamAssassin stands for the most popular and highly extensible framework for spam filtering. The whole project has been developed following an object-oriented design and includes a
plug-in architecture to facilitate its extensibility. SpamAssassin can be seen as a spam filter development language together with the corresponding interpreter that can be easily connected with third-party MTA software.

In this section we present a detailed study of SpamAssassin operation, software architecture and limitations. First, Section 2.1 introduces the SpamAssassin filtering process and examines internal software design elements used to implement the whole framework operation. Next, Section 2.2 presents some SpamAssassin framework limitations identified while carrying out this work and addressed in our Wirebrush4SPAM framework.

2.1. Architecture overview

SpamAssassin is a framework able to automatically classify e-mail messages through user-defined spam filters. Therefore, SpamAssassin provides a clear separation between filter definition and the implementation of different filtering techniques. A SpamAssassin filter is defined by a set of scored rules and a global threshold called \textit{required\_score}. Each rule is composed of a Boolean expression (used as trigger) and its associated individual score. Following this simple structure, an e-mail is classified as spam when the sum of individual scores from triggered rules is greater or equal than the value of \textit{required\_score}.

To develop Boolean expressions for rules, SpamAssassin supports the usage of Perl regular expressions [10] to find patterns in e-mail headers and/or the body. With the goal of extending/adapting basic SpamAssassin functionalities for developing specific rules, there are several implementations of filtering techniques provided by SpamAssassin plug-ins as filtering functions. Rules belonging to the user filter are usually included in .cf files located in the /usr/share/spamassassin and /etc/mail/spamassassin directories. Figure 1 shows an example rule extracted from the default SpamAssassin filter included in Debian GNU/Linux distribution (files /usr/share/spamassassin/50_scores.cf and /usr/share/spamassassin/23_bayes.cf). As specified in Figure 1 line 01, the rule BAYES\_00 is applied on the \textit{body} of the target message and is triggered when the call to function \textit{check\_bayes} becomes true. The function call also receives the optional parameters \textit{nice} and \textit{learn} as showed in Figure 1 line 02. In line 04 a specific \textit{score} is provided to increment message result every time the rule is triggered. Rule \textit{score} configuration on the latest SpamAssassin distributions can include four different values to handle the following situations: (i) Bayes and network tests are disabled, (ii) Bayes is disabled but not network tests, (iii) network tests are disabled but not Bayes and (iv) everything is enabled. Additionally, rules can also include an optional description as shown in Figure 1 line 03.

SpamAssassin can be easily integrated with most popular MTA software to filter all incoming e-mails. Communication between MTA and SpamAssassin daemon (spamd) is usually carried out through a Transmission Control Protocol connection following the SpamAssassin network protocol [11]. Figure 2 shows the SpamAssassin operation workflow together with the main classes and methods supporting it. As shown, the SpamAssassin filtering process includes the execution of four sequential steps: (i) e-mail parsing, (ii) rule execution, (iii) learning and (iv) report generation.

As Figure 2 shows, Mail::SpamAssassin is the main class guiding the whole filtering process. During the first filtering step, SpamAssassin instantiates an object of class Mail::SpamAssassin::Message to parse the target e-mail. The evaluation of rules is accomplished by executing regular expressions and functions provided by plug-ins (Mail::SpamAssassin::Plugin) through calling the \textit{callback} method from Mail::SpamAssassin::PluginHandler class. To complete the learning stage, each plug-in should provide an implementation of its training process. Finally, an instance of

![Figure 1. SpamAssassin rule example.](image-url)

Copyright © 2012 John Wiley & Sons, Ltd.


DOI: 10.1002/spe
Mail::SpamAssassin::PerMsgStatus class, compiles all the information generated during the whole classification process and produces a specific filtering report according to the request received by the `spamd` daemon.

To better understand the SpamAssassin plug-in architecture, Figure 3 shows an extract of its class diagram where private methods and attributes have been removed for clarification purposes. Moreover, only three plug-in implementations are shown and some public non-relevant methods have been also hidden. Each SpamAssassin plug-in is implemented as a Mail::SpamAssassin::Plugin subclass that makes at least one call to the inherited `register_eval_rule` method. This method is used to register a filtering function that can be used in any SpamAssassin filter. The `check_bayes` function shown in Figure 1 is provided by the Mail::SpamAssassin::Plugin::Bayes plug-in (usually located in the `/usr/share/perl5/` directory). As expected, its constructor (new subroutine showed in Figure 4) contains the source code where the subroutine `check_bayes` is registered as a function that can be later used for filter definition.

As we can observe from Figure 3, Bayes, SPF (sender policy framework) and Test are SpamAssassin plug-ins providing diverse functionalities to build custom filters. Bayes registers the function `check_bayes`, Test registers `check_test_plugin` (which always gets true) and finally, SPF registers `check_for_spf*` functions.

Additionally, in SpamAssassin some plug-ins implement different learning schemes (e.g. Bayes plug-in). In such a situation, the learning process cannot be accomplished while filtering rules are being executed, because the message classification has not been completed yet. SpamAssassin plug-in learning support is provided through the implementation of `learner_new`, `learn_message` and `learner_close` inherited methods (see Figure 3). Nevertheless, these functions have been introduced since the SpamAssassin 3.3-1 version, and only `bayes_learn` abstract method was available to address this functionality in previous distributions.

Regarding learning issues, Mail::SpamAssassin::Plugin::AutoLearnThreshold is also another important class (see Figure 3). This SpamAssassin plug-in only implements the function `autolearn_discriminator` that computes a tri-state result (1, 0 or undefined) to decide if the message will be learned as spam, ham or not learned. This outcome is made after classifying the target message and depends on both, some SpamAssassin configuration parameters (`auto_learn`, `bayes_auto_learn_threshold_nonsmtp` and `bayes_auto_learn_threshold_spam`) and the result of the
classification stage. If the result of calling this function gets 1 or 0, bayes_learn and learn_message methods (the latest is only available in SpamAssassin 3.3-1) are automatically invoked. Moreover, if required by developers, they can inherit learner_new and learner_close methods that are automatically called by SpamAssassin to initialize and shutdown the learning subsystem.

As shown, SpamAssassin plug-ins can be easily developed by inheriting from the Mail::SpamAssassin::Plugin class, registering functions in the class constructor by using the register_eval_rule method and implementing these functions as class methods. If auto learning capability is required, it can be obtained by overriding the method learn_message in SpamAssassin 3.3-1 or bayes_learn in previous versions. Moreover, SpamAssassin also includes some utilities for message parsing through Mail::SpamAssassin::Message and Mail::Internet classes. Finally, the main class of the SpamAssassin architecture is Mail::SpamAssassin, which can be used from another Perl application by taking advantage of the source code included in Figure 5.

2.2. Shortcomings of SpamAssassin

During our perceptive analysis of SpamAssassin, we detected some weaknesses causing a downturn in filter efficiency. The usage of an interpreted programming language, the lack of appropriate cache structures, the absence of rule execution schemes to improve efficiency and different deficiencies in multithreading scheme are included in this group. Moreover, some companies reported the need...
of implementing new features to enhance spam filtering like supporting the execution of both suf-
ficient condition and domain specific rules. All of these drawbacks are presented and discussed in
this section.

First, the choice of an interpreted language instead a compiled one seems suitable to reduce devel-
opment time. However, some technical and scientific works suggest that execution of applications
by virtual machines and interpreters is slower than compiled software [12–14]. Following these
studies, the Perl language seems not to be the best programming alternative to develop efficient
spam filtering software.

SpamAssassin does not use caching schemes to store intermediate results such as network test
responses (including SPF, RBL/RWL (realtime black and white lists), Razor, Pyzor, etc.) or partial
computations of Bayes probabilities. Nevertheless, network tests are commonly based on domain
name system queries and time elapsed during their execution is not constant. To delimit time require-
ments of these tests, some companies are offering offline services to download databases offline and
avoid network operations during test execution (e.g. Spamhaus DNSBL Datafeed‡). However, most
Internet service providers do not offer this kind of service and therefore, cache improvements are
required. Finally, tests demanding high computational requirements should be computed once per
e-mail, even when the message is received two or more times. In this sense, while executing some
general experiments to measure the performance of SpamAssassin, we detected a significant filtering
time increment when adding more rules to the built-in Naïve Bayes classifier. After a precise verifi-
cation, we corroborated that older SpamAssassin versions computed Bayes probability one time per
check_bayes call. Fortunately, SpamAssassin 3.3-1 solves this problem, but when the server receives
more than one copy of the same message (a common situation) it executes a complete Bayes analysis
for each e-mail.

From another point of view, every time SpamAssassin classifies a new incoming message, it eval-
uates all the rules defined in the user filter. We believe in the existence of situations where the
execution of the filter can be stopped without changes in the final classification result. These sit-
uations should be identified and mathematically modelled to save computational resources during
filter operation. These improvements are missed in the current SpamAssassin framework.

Another interesting issue is that SpamAssassin is able to take advantage of threading/
multiprocessing capabilities by enabling multiple instances of Mail::SpamAssassin class handled by
a single spamd instance and a fork-based scheme. The number of SpamAssassin server processes can
be easily modified assigning new values to –min-children and –max-children configuration options.
This design is easy to implement but concurrency is only possible when filtering more than one
message at the same time. Moreover, implementing thread safe schemes is not required to develop
a new plug-in, and this fact can wrongly lead to believe in the impossibility of parallel executing
functions. However, two functions can be concurrently executed without problems if they are pro-
vided by different plug-ins, because there is no support for data/variable sharing between them. In
fact, the execution of Input/Output operations (e.g. network operations and tests implemented in

‡ http://www.spamhaus.org/datafeed/index.lasso
SpamAssassin) and highly demanding tasks (i.e. computing naïve Bayes probabilities) are the basis of the argumentation used to introduce parallelism in computer science.

Taking into consideration efficiency when executing rules, a sufficient condition is a concept designed to allow the definition of short cuts to classify an e-mail as spam (or ham) regardless of its global score. This can be easily implemented in SpamAssassin or Wirebrush4SPAM frameworks by assigning a very high score (to a sufficient condition spam rule) or a very low score (to a sufficient condition ham rule). Although the implementation of this feature is simple, sufficient condition rules should be kept in mind when addressing filter-scheduling schemes.

From another perspective, supporting domain-specific rules allows the definition of conditions that cannot be required for every filtered message. A domain-specific rule should be executed only when the message is sent to some of the domains specified by the rule. As a practical example, this feature can be exploited to avoid the execution of rules handling drug terms when target e-mail is delivered to a drug company. We believe this feature should be included in every filtering framework as SpamAssassin or Wirebrush4SPAM.

Most of the shortcomings found in the SpamAssassin middleware have guided the design of the Wirebrush4SPAM framework. The next section introduces the main characteristics of our proposal and explains how we addressed previously commented issues.

3. WIREBRUSH4SPAM: ESSENTIAL CONCEPTS AND PRINCIPLES

As previously commented, when analysing SpamAssassin software the first issue detected was related with the execution speed of the object-oriented Perl source code. As suggested in previous studies, C and C++ are the fastest platforms to develop services and Perl source code can be up to 64 times slower than them. These studies also corroborate that the C language is a little bit more efficient than C++.

Filter middleware should be as fast as possible because, at the same time that new and more complex antispam techniques are developed, spam deliveries are constantly growing. In such a situation, old structured programming with C represents the most suitable way to guarantee efficiency requirements in next-generation filter middleware. In this context, optimization flags (i.e. -O1, -O2 and -O3) should also be kept in mind while building final versions.

The main challenge for developing a C filter framework was the need for designing a highly extensible architecture. To solve this problem, we analysed two different alternatives of previously developed successful plug-in frameworks for C [15, 16]. Although libPlugin [15] last commit took place in 2008 and seems to be discontinued, C-Pluff [16] last commit was in 2010. Attending to this fact, the entire architecture of Wirebrush4SPAM was developed using the C-Pluff framework as its working base.

3.1. Framework architecture

The main difference between SpamAssassin and Wirebrush4SPAM framework is the renouncement of both class and hierarchy concepts as the main scheme for implementing new software features in favour of adopting a more traditional approximation including call-backs and handler functions. Nevertheless, Wirebrush4SPAM architecture has been inspired in an object-oriented design where classes have been substituted by C modules implementing abstract data types. Figure 6 presents the general overview of the proposed architecture.

As we can see from Figure 6, key characteristics from the object-oriented paradigm (including encapsulation and information hiding) have been ported to the Wirebrush4SPAM architecture. However, we have not taken advantage of some object-oriented characteristics like object inheritance or subtype polymorphism that was the basis of the SpamAssassin plug-in development. Instead of this, wb4spam is the main program able to load the C-Pluff plug-in architecture, initializes the core plug-in and subsequently forwards messages to it. The core plug-in is able to classify e-mails by evaluating rules included in a ruleset. For the execution of each rule, the core plug-in first obtains the message content by using a parser_t data type and then executes a function_t to check the matching with the target e-mail. To increase filtering speed, each parser_t is launched only when the filter...

Figure 6. General overview of Wirebrush4SPAM architecture.

contains a rule that requires its usage, being executed only one time per message by using a parsed contents caching scheme. Once Wirebrush4SPAM has classified the new incoming message, the core plug-in calls all registered eventhandler_t to notify them of the final decision about the e-mail. In the proposed scheme, event handlers are the Wirebrush4SPAM mechanism to support automatic learning processes.

Parsers, filtering functions and event listeners represent the main Wirebrush4SPAM concepts. They are modelled as extensions that can be connected with the core plug-in through the corresponding extension points. Therefore, a Wirebrush4SPAM plug-in (except the core) is composed of a set of parsers, filtering functions and event listeners sharing some semantic or functional relationships. Figure 7 exemplifies the Wirebrush4SPAM plug-in architecture by representing four available plug-ins. As we can realize from Figure 7, Bayes plug-in registers a filtering function and an event listener used to support the learning requirements of this plug-in. Moreover, Wirebrush4SPAM also includes an EMLParser plug-in that contains header, full and body parsers used to extract information and tokenize the corresponding parts of any e-mail represented in RFC2822 format [17].

Figure 7. Conceptualization of Wirebrush4SPAM plug-in architecture.
Finally, we have also included in Figure 7 the first plug-in developed in the Wirebrush4SPAM project — the false plug-in. This plug-in registers the dummy false function that does not match with any e-mail. This plug-in stands for the SpamAssassin TestPlugin that registers the test function matching with any message.

Figure 8 includes a detailed schema of the Wirebrush4SPAM plug-in architecture focusing on the parser functions and event listeners implemented by the Bayes plug-in. As we can see from this figure, every C-Pluff plug-in contains a descriptor file named plugin.xml that specifies runtime features including (i) the plug-in id, (ii) available extension points, (iii) implemented extensions, (iv) a dynamic library file, (v) existing plug-in dependences and (vi) different user-defined information. These extensions defined by a plug-in are connected with their corresponding extension points by some sentences included in the plug-in descriptor. As showed in Figure 8, the Bayes plug-in implements check_bayes filtering function and bayes_learn event handler by extending the corresponding extension points. The core plug-in defines three extension points matching the main concepts of Wirebrush4SPAM platform: parsers, filtering functions and event listeners. For each extension point, an XML schema file (.xsd) should be created containing the parameters to be included in a plug-in descriptor (plugin.xml).

To get a deeper insight about the required files for defining a given plug-in, Figure 9(a) presents an extract of the source code from Wirebrush4SPAM false plug-in. As we can observe, the plug-in contains the definition of a function and some CP_EXPORT sentences that make available a function_t variable. When the plug-in does not use a certain data structure associated with it, the export sentence is similar to the one included in line 04. Figure 9(b) shows the plug-in descriptor file for the false plug-in the example.
To analyse a more elaborated example, Figure 10(a) shows some fragments from the Wirebrush4SPAM bayes_plugin.c file. As we can observe, plug-ins using internal data structures are more complex to define and require a cp_plugin_runtime export sentence to identify the source to execute, create, initialize, stop and destroy data structures. Moreover, required exports for function_t, parser_t or eventhandler_t variables should be defined inside the start function as shown in lines 26 to 33. Finally, Figure 10(b) contains the source code belonging to the descriptor of Bayes plug-in. As previously mentioned, the descriptor file is required to link the exported extensions to the core extension points.

3.2. Solving SpamAssassin inefficiencies

The design of a new architecture to develop Wirebrush4SPAM using C as programming language and C-Pluff as plug-in development platform alleviates some of the computational overheads inherent to SpamAssassin. Our proposal to cope with the remaining SpamAssassin shortcomings are introduced and discussed in this section.

First, as commented before, the implementation of cache schemes is required to improve SpamAssassin middleware. This approach should be used to avoid unnecessary highly demanding tasks and improve the time required to execute network tests. In such a situation, we found that using a cache for storing previous Bayes results can avoid a lot of unnecessary Bayes analysis. The proposed solution in Wirebrush4SPAM combines a linked list structure with a hashmap to keep the order of elements providing a fast access to the stored information. Moreover, this cache structure should be reusable in conjunction with other antispam techniques. To handle this issue, we have developed a cache data type included in the core plug-in as shown in Figure 11.

As we can observe from Figure 11, the cache facility is a data type with four functions used to efficiently manage the data structure. We have successfully applied this core facility while executing SPF [18], Bayes and RBL/RWL [19, 20] functions.

From another perspective, while developing SpamAssassin-based filters, we also found high scores after the execution of a few rules. In such a situation, achieving a final score lower than the globally defined required_score threshold is improbable. Additionally, we can experiment with a similar situation when the message achieves very low scores. In this context, some time could be saved by using our SFE heuristic. The idea behind SFE emerged from the lazy Boolean expression evaluation scheme and lets us avoid the execution of some filter rules when the partial score of an e-mail is too high or too low for exceeding the required_score threshold.
Figure 10. Source code and descriptor file for the Bayes plug-in available in Wirebrush4SPAM platform.
Initially, we defined \textit{too high} and \textit{too low} parameters as static configuration values that should be specified in advance by Wirebrush4SPAM administrators. By using this approach during filter evaluation, if an e-mail score gets out of the bounds, the filter execution can be stopped and a final classification is conducted. However, this straightforward approach possesses several drawbacks including: (i) how to define the appropriate interval values to get the same result obtained without SFE and achieve a filter speed improvement and (ii) what happens when these rules with positive/negative scores are executed at the beginning of the filter, being more probable to reach a value out of the bounds. Because of these issues, filter execution might abort prematurely with a wrong e-mail classification.

All these drawbacks forced us to adopt a more flexible and reliable solution. Therefore, we developed an SFE scheme where boundaries are automatically adjusted. In this approach, we maintain two variables called \textit{pending\_add} and \textit{pending\_subtract} that stand for the sum of positive scores and the sum of negative scores from unexecuted rules, respectively. The formal definition of these variables is showed in Equation (1).

\[
pending\_add_t = \sum_{r_i \in UnexecutedRules_t} score(r_i) > 0 \\
pending\_subtract_t = \sum_{r_i \in UnexecutedRules_t} score(r_i) < 0
\]  \hspace{1cm} (1)

where \textit{UnexecutedRules}_t is the set of rules that has not been executed at instant \(t\) and \(score(r_i)\) stands for the final score associated to rule \(r_i\).

Intuitively, the score of a given e-mail cannot get lower than the \textit{required\_score} if at any instant \(t\), the sum of all the negative scores belonging to unexecuted rules would not get the message score lower than \textit{required\_score} \((score + pending\_subtract_t \geq required\_score)\). The same situation occurs when the score of a message cannot get higher than \textit{required\_score} \((score + pending\_add_t < required\_score)\). These situations are used to define a variable score interval as shown in Equation (2). Therefore, the execution of the filter can be stopped if the score for the target e-mail gets out of the bounds defined by the interval specified by Equation (2).

\[
[required\_score - pending\_add_t, required\_score - pending\_subtract_t]. \hspace{1cm} (2)
\]

Using our SFE technique we can obtain a considerable time saving while executing spam filters. Moreover, SFE can be disabled for filter development, test and/or debugging tasks by modifying a configuration parameter of Wirebrush4SPAM.
Another important topic previously commented is that filter frameworks should take advantage of the newest processor generations by exploiting their multiprocessing capabilities. To accomplish this objective, a suitable combination should be achieved between fork processes and threads. Despite the fact that the classification of several messages at the same time can be useful in some situations, we are aware of the need of optimising rule evaluation for each message by allowing the concurrent execution of different rules. This approach implies an increment in the number of parallelizable subtasks and, according to Amdahl law [21], the possibility of achieving an interesting increment in execution speed by using parallelism. Therefore, we developed Wirebush4SPAM under the principle of concurrent rule execution instead of concurrent message classification previously used by SpamAssassin.

In our Wirebush4SPAM framework, the fork-based approach used by SpamAssassin has been replaced by a threading (lithread) scheme to reduce the computational overhead and simplify the development tasks. With the goal of increasing filtering efficiency, we also implemented an easy concurrent message filtering feature (like SpamAssassin). Nevertheless, performance degradation can occur in some situations when the number of threads gets high and use several critical sections [22, 23]. Therefore, we should not allow indiscriminate thread creation.

Our approach to restrict the number of threads being concurrently executed is based on avoiding the parallel execution of two extensions (filtering functions, event handlers or parsers) when they are implemented in the same plug-in. This scheme introduces the following advantages: (i) mutual exclusion is not a problem for plug-in developers and (ii) there is no thread locking while executing source code from plug-ins. Under this approach, the maximum number of messages that can be concurrently classified is equal to the number of plug-ins used by the filter. Using a lower value for message filtering concurrency can be useful to classify e-mails when the number of cores is too low.

To achieve a greater degree of parallelism and more efficiency while using SFE scheme, we have also developed a prescheduling technique for managing rules. Currently prescheduling ensures that rules executing functions that belong to the same plug-in must be launched as far as possible, preventing the problem of threads waiting until the finalization of other rules. The prescheduling implemented in Wirebush4SPAM is carried out only once, after rule loading. Furthermore, it admits the deployment of new prescheduling techniques to allow the use of custom heuristics.

Despite having achieved relevant advances from current efforts to improve the multithreading/multiprocessing schemes, we believe that this issue has not been completely studied and should be included as future work.

Although filtering efficiency is the most important goal covered in the present work, we have also addressed some other interesting spam filtering characteristics. Multidomain filtering is a feature in great demand for enterprises offering spam filtering services, because some customers are small companies or final users demanding low-priced complete e-mail solutions. Therefore, a spam filter can be shared to classify the e-mail delivered to several (and usually different) domains.

The idea of multidomain filtering introduces a new challenge on spam classification. Filters needed for classifying e-mails received by a drug store could be very different to those required to classify messages in a clock and watch shop. The first one should not recognize as spam messages containing terms like ‘viagra’ or ‘ciallis’, while the second one presents the same issue while classifying messages containing terms like ‘rolex’ or ‘watch’. As we show in this example, there are some rules that cannot be applied on e-mails from different domains.

Moreover, we also know the interest of some final users to write their own filtering rules. To give support to this feature, filtering enterprises should provide web control panels allowing the customization of some filtering behaviours. Although this is probably a bad idea from a practical point of view, these customization features are greatly evaluated for a lot of customers (even being naïve). Therefore, these facilities should include a reset-to-default action to undo the changes made by inexperienced users.

To support rules included by final users or rules that are only valid in some domains, we have included in Wirebush4SPAM the optional domain configuration directive for each rule. This directive is used to specify a list of target domains in which the rule is applicable. This feature is included in the filter example introduced in the next section and contributes to the definition of complex filters with a reasonable computational demand.
4. FILTER DEVELOPMENT IN WIREBRUSH4SPAM

As discussed in previous sections, the main goal of the Wirebrush4SPAM platform is focused on maximizing the spam filtering speed while minimizing the associated computational cost. However, we have also considered the need of developing new extensions to provide alternative filtering strategies that can be easily combined to build accurate spam filters.

A filter is composed of a combination of different techniques that can be viewed as an expert committee. Each expert (rule) issues a decision previously weighted. Administrators develop rules and weigh each one according to their own experience. In this context, the framework should afford the implementation of different techniques (functions) and parsers allowing the definition of rules by administrators. Table I introduces a brief description about the filtering plug-ins and functions available in the Wirebrush4SPAM platform.

As showed in Table I, each function usually requires the execution of a specific parser to obtain the information needed to score the message. Table II summarizes the parsers included in the Wirebrush4SPAM platform.

<table>
<thead>
<tr>
<th>Plug-in</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>check_bayes(&lt; min &gt;, &lt; max &gt;)</td>
<td>Check if the probability of the message being spam is included in the specified interval. This function should be only used with body parser.</td>
</tr>
<tr>
<td>rxl</td>
<td>rxl_check(&lt;list_suffix&gt;[, &lt;number_received_header&gt;])</td>
<td>Check if the server that executed the delivery (first one) is included in RBL/RWL lists. This function should be only used with header parser.</td>
</tr>
<tr>
<td></td>
<td>rxl_check(&lt;list_suffix&gt;,&lt;octect_number&gt;, &lt;octect_value&gt;[,&lt;number_received_header&gt;])</td>
<td>Very similar to the previous one. Allows the user to perform the comparison of the RBL/RWL result with a given value.</td>
</tr>
<tr>
<td>spf</td>
<td>spf_fail([received_header_number])</td>
<td>Gets true when SPF records are in the referenced state. The argument is optional. SPF plug-in checks the first received header in the e-mail. If SPF records should be tested over a different received header, a parameter should be specified (order from the beginning). These functions should only be used in conjunction with header parser.</td>
</tr>
<tr>
<td></td>
<td>spf_softfail([received_header_number])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>spf_none([received_header_number])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>spf_neutral([received_header_number])</td>
<td></td>
</tr>
<tr>
<td>regex</td>
<td>eval(&lt;regex&gt;)</td>
<td>Verifies if a POSIX regular expression matches in the e-mail body. It should only be used with body parser.</td>
</tr>
<tr>
<td></td>
<td>eval_header(&lt;header&gt;,&lt;regex&gt;)</td>
<td>Verifies if a POSIX regular expression matches in the specified e-mail header. It should only be used with header parser.</td>
</tr>
<tr>
<td>pcre_regex</td>
<td>pcre_eval(&lt;regex&gt;)</td>
<td>Verifies if a PCRE regular expression matches in the e-mail body. It should only be used with body parser.</td>
</tr>
<tr>
<td></td>
<td>pcre_eval_header(&lt;header&gt;,&lt;regex&gt;)</td>
<td>Verifies if a PCRE regular expression matches in the specified e-mail header. It should only be used with header parser.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plug-in</th>
<th>Parser</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eml_structure_parser</td>
<td>body</td>
<td>Dump the body of an rfc2822 message.</td>
</tr>
<tr>
<td></td>
<td>header</td>
<td>Dump the headers of an rfc2822 message.</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>Dump headers and body of an rfc2822 message.</td>
</tr>
<tr>
<td>url_parser</td>
<td>url</td>
<td>Finds all the URLs contained in the entire message.</td>
</tr>
</tbody>
</table>
Because parsers header and full execute a deep header parsing, filters using these parsers involve a duplicate processing of the same data. The same problem is observed while using body and full parsers. Therefore, the eml_structure_parser plug-in only include a parser called rfc2822 able to perform a full processing of the message. This feature can aid in reducing the computational cost of message processing while minimizing filter development complexity.

Table III. Wirebrush4SPAM event listener.

<table>
<thead>
<tr>
<th>Plug-in</th>
<th>Event listener</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes_plugin</td>
<td>bayes_learn</td>
<td>Execute the Bayes learning of the specified message. Wirebrush4SPAM must specify whether the message has to be learned as ham or spam.</td>
</tr>
</tbody>
</table>

Figure 12. Wirebrush4SPAM filter definition example.
Wirebrush4SPAM plug-ins can include some event listeners that are called after the message classification. As shown before, event listeners are call-backs used to notify some plug-ins about the filtering of a message. This feature is useful for developing some filtering approaches like auto white list or Bayes continuous updating schemes. Table III shows the event listener included in Wirebrush4SPAM used to execute the learning process for Bayes plug-in.

Finally, Wirebrush4SPAM is able to execute META rules (also supported by SpamAssassin), which can be used to combine the results of different rules using Boolean expressions and operators.

Wirebrush4SPAM filters are defined in *.*.cf files located in the filter directory. To build a filter, these files should contain all rules and the required score threshold. Figure 12 shows an example of a Wirebrush4SPAM filter.

As showed in Figure 12, the filter involves the execution of a Bayes scheme to compute the probability of a message being spam. The proposed filter uses some intervals for the Bayes probability and assigns a score for each interval (lines 02, 06, 10,..., to 34). Moreover, it also adds some scores to the target message when it contains the word ‘viagra’ with different variations (line 38) or when the subject of the e-mail contains the word ‘Levitra’ (line 42). These two rules use a different regular expression API to test the specified conditions. The example filter also checks SPF records and a RBL/RWL searching scheme. The RBL/RWL entry in line 50 represents a domain-specific rule that will only be tested when the e-mail receiver is somebody from one of the Galician Universities.

Although all the commented improvements have been successfully introduced, developed and tested in Wirebrush4SPAM, a lot of SpamAssassin plug-ins need to be ported to Wirebrush4SPAM to provide the same functionalities as SpamAssassin. However, most of the proposals included in this work can be easily coded for improving SpamAssassin middleware while SpamAssassin plug-ins are under development.

5. FILTER BENCHMARKING AND RESULT DISCUSSION

To test the suitability of the proposed methods, we designed and executed an experimental benchmarking test using the filter introduced in Figure 12. Results obtained from the experiments carried out are presented and analyzed in Section 5.1. Moreover, Section 5.2 provides a discussion about the potential performance contribution of each specific technique over the performance of SpamAssassin. Finally, Section 5.3 presents some details about the experience achieved while carrying out this work.

5.1. Comparative analysis

To demonstrate quantitative improvements achieved by our Wirebrush4SPAM platform, we used the filter shown in Figure 12 but disabling domain-specific functionalities to ensure the execution of the same amount of rules in both platforms (Wirebrush4SPAM and SpamAssassin). For comparison purposes, we have compiled a simple corpus comprising four groups of e-mails having the same ham/spam proportion: (i) a set of 500 training messages, (ii) a set of 300 training messages, (iii) a set of 200 training messages and (iv) a set of 150 test messages. We tested SpamAssassin VS Wirebrush4SPAM platforms with three different training scenarios represented by the available training groups.

To compare the performance of the analysed platforms, three processing time measures are used: (i) real time, (ii) user time and (iii) system time. Real time measures the milliseconds from the simulation start to its finalization. User time represents the amount of CPU time spent in executing user-mode code within the process (time used for carrying out the e-mail classification). Finally, system time measures the CPU time spent by the system kernel while executing the target process. Obviously, during the experiments message classification results were the same in both filtering approaches. Table IV summarizes the benchmarking results using an Intel 2.2 MHz. Core 2 Duo CPU with 2 GB of RAM executing an Ubuntu 10.04 GNU/Linux OS.

As we can see from Table IV the time required for training the filter increased proportionally to the size of the corpus. Moreover, Table IV shows that using a training corpus with 500 e-mails Wirebrush4SPAM is able to classify 150 messages in less than 7 s. Moreover, using the same training corpus SpamAssassin needed more than 2 min to classify exactly the same messages.
Table IV. Wirebrush4SPAM VS SpamAssassin benchmarking results.

<table>
<thead>
<tr>
<th>Training corpus size</th>
<th>Time measures</th>
<th>Wirebrush4SPAM</th>
<th>SpamAssassin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Classification (150 e-mails)</td>
<td>Classification (150 e-mails)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Training time</td>
<td>Training time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real 0 m 6.274 s</td>
<td>0 m 5.664 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user 0 m 1.356 s</td>
<td>0 m 1.204 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys 0 m 4.544 s</td>
<td>0 m 1.276 s</td>
</tr>
<tr>
<td>200</td>
<td>real</td>
<td>0 m 13.309 s</td>
<td>0 m 5.771 s</td>
</tr>
<tr>
<td></td>
<td>user</td>
<td>0 m 2.708 s</td>
<td>0 m 1.212 s</td>
</tr>
<tr>
<td></td>
<td>sys</td>
<td>0 m 8.749 s</td>
<td>0 m 1.268 s</td>
</tr>
<tr>
<td>300</td>
<td>real</td>
<td>0 m 21.590 s</td>
<td>0 m 6.682 s</td>
</tr>
<tr>
<td></td>
<td>user</td>
<td>0 m 3.932 s</td>
<td>0 m 1.188 s</td>
</tr>
<tr>
<td></td>
<td>sys</td>
<td>0 m 14.849 s</td>
<td>0 m 1.292 s</td>
</tr>
<tr>
<td>500</td>
<td>real</td>
<td>0 m 16.935 s</td>
<td>2 m 24.887 s</td>
</tr>
<tr>
<td></td>
<td>user</td>
<td>0 m 14.689 s</td>
<td>1 m 15.501 s</td>
</tr>
<tr>
<td></td>
<td>sys</td>
<td>0 m 1.768 s</td>
<td>0 m 7.296 s</td>
</tr>
</tbody>
</table>

The message classification throughput achieved by Wirebrush4SPAM is 1885 messages/minute while SpamAssassin is only 185 messages/min. This improvement has an important effect when developing and deploying complex filters using a wide variety of available techniques.

5.2. Discussing the effect of the proposed improvements

As previously shown, Wirebrush4SPAM is about 10 times faster than SpamAssassin middleware. In this section, we discuss the potential impact of each improvement implemented in Wirebrush4SPAM to assess their relative importance for evaluating cost VS benefit.

The most important improvement included in Wirebrush4SPAM was the complete reimplementation from scratch of the whole middleware and parsers. Although it is a hard task, it contributed to achieve a great filtering speed increment. In fact, Wirebrush4SPAM e-mail parser (built using a finite state machine) is about five times faster than SpamAssassin. Moreover, we also took advantage of using a compiled language to develop Wirebrush4SPAM instead of using an interpreted one. However, the time required to execute a Perl regular expression is quite less than the one needed by other C implementations (e.g. POSIX regex API or Perl Compatible Regular Expression Library) but any other algorithm will run slower when executed by Perl interpreter software.

The Wirebrush4SPAM threading approach has also been a good decision. Using SpamAssassin, filter rules cannot be concurrently executed and therefore, the time spent to classify a given e-mail can be easily estimated as the amount of time required to individually execute all the rules. However, Wirebrush4SPAM is able to concurrently execute several rules if the called functions are implemented by different plug-ins. As an example, our framework could concurrently evaluate five rules from any given experimental filter (e.g. BAYES_00, HAS_VIAGRA_ON_BODY, HAS_SUBJECT_PCRE, SPF_PASS and RWL_DNSWL) using only the time required for the slowest rule.

The optimization of cache schemes is another useful improvement that complements threading issues. Wirebrush4SPAM provides data types and functions to support different caching schemes and all plug-ins included in the current version take advantage of this functionality. In fact, during the execution of BAYES_00 rule, the probability of a target message being spam (computed using naïve Bayes algorithm) is stored in a cache. As supposed, this value is not computed again for the same e-mail while evaluating BAYES_05, BAYES_20, BAYES_40, BAYES_50, BAYES_60, BAYES_80, BAYES_95 and BAYES_99 rules. Therefore, these rules can be quickly evaluated by only checking if Bayes probability is included in a numerical interval. Similarly, during the execution of RWL_DNSWL rules, the results of querying the list.dnswl.org white list are stored in a cache. Afterwards, during the execution of RWL_DNSWL_OCTECT rule, the white list query is not needed again and therefore, its processing is faster. Caches are also used in different parts of Wirebrush4SPAM for optimization purposes. In our experiments, we also corroborated that the classification of a message using caches is up to six times faster than without them.
Despite that our SFE heuristic was activated during the experimental stage, it does not prevent the execution of any rule. Although this feature can reduce the time required to filter a message, the probability of avoiding the execution of some rules is very low in simple filters (like the one used in our experiments). Finally, the execution of domain specific rules did not lead to any performance improvement because this feature is only useful when applying different rules to filter messages received by several domains in a single instance of Wirebrush4SPAM.

5.3. Learned lessons

In our daily use of Wirebrush4SPAM framework, every new experiment helps us in finding bugs and raises new ideas for implementing novel features. Moreover, our experience designing and using antispam filters led us to several interesting conclusions and lessons. This know-how is presented in this section.

One of the most difficult tasks while developing the Wirebrush4SPAM platform was the detection and correction of several memory leak errors. To find, understand and fix these bugs, VALGRIND (Seward J, et al., Cambridge, United Kingdom) software [24] was very useful and saved us a lot of time. Despite this, we spent more than 15% of the developing time to ensure the quality of the software.

We also want to highlight the difficulties experienced in developing efficient rfc2822 parsers. In the beginning, we used FLEX (fast lexical analyser) [25] for generating fast parsers starting from specific configuration files (.lex). However, after several months using FLEX parsers, we found that specific functions developed from scratch could be more efficient than our initial approach. Taking into consideration this circumstance, all Wirebrush4SPAM parsers were finally coded from scratch and they were optimized to recursively handle multipart messages by using a stack data structure.

Related with the flexibility of Wirebrush4SPAM for building and modulating complex conditions, we also developed source code from scratch to parse and evaluate META rules. This kind of rules is also provided in SpamAssassin core to allow the definition of new rules as logical combinations of previously defined conditions.

As introduced in previous sections, to prevent the execution of unnecessary rules, we defined both too_high and too_low parameters as static configuration values. Therefore, if an e-mail score gets out of the bounds defined by this interval, filter execution can be stopped. However, this approach might cause an unsafe stop status and prematurely abort the filter with a wrong e-mail classification. To correctly address this issue, we found that lazy evaluation techniques borrowed from programming languages could be applied to filter the evaluation domain with few modifications. In this context, our SFE heuristic can save a lot of computing time while executing complex filters.

An important issue affecting the whole project was related with the thread scheme used for taking advantage of the latest processors. In this context, the concurrency available through the utilization of multitasking and multiprocessing operating systems is not enough to make the most of these processors. In this situation, the law of Amdahl should be kept in mind for maximizing the number or parallelizable subtasks (and minimizing the time required for their execution) to make the most of parallelism [21].

All of these efforts combining theory and practice have led us to achieve an important increment on spam filtering throughput. However, we believe that improving performance is still possible by accomplishing part of the ideas presented in the next section.

6. CONCLUSIONS AND FURTHER WORK

This paper has presented Wirebrush4SPAM, a novel C framework specifically designed for efficient spam filtering. As previously shown, our proposal provides several advantages when compared with other existing spam filtering frameworks. These improvements include:

(i) Wirebrush4SPAM was entirely coded using ANSI/C language, which allows a significant reduction of the execution time. However, we had to modify the filter architecture design to adapt it to structured programming.
(ii) Wirebrush4SPAM is highly extensible through plug-ins in ANSI/C language. There is a core plug-in used to guide the filter execution. The core also provides some facilities to help the programmer in the development of new plug-ins.

(iii) Use of caches in SPF, Naïve Bayes (NB), RBL, RWL, and regex (both PCRE and POSIX versions) to store the intermediate results avoiding the unnecessary execution of functions more than once. Moreover, the cache size of each plug-in can be manually configured depending on the requirements of each user and the storage capacity of the computer. Caching facility is provided by the core of Wirebrush4SPAM and can be easily used to develop future plug-ins.

(iv) Wirebrush4SPAM is able to parallelize the evaluation of filter rules for each new incoming message. This characteristic leads to an important increment on speed when executed in the latest multicore processors.

(v) Smart filter evaluation strategy prevents the execution of rules that are irrelevant to the e-mail classification process. When the rules pending to execute do not affect final e-mail classification, the filter is aborted and the message is classified according to the punctuation achieved until that moment.

(vi) Wirebrush4SPAM is 10 times faster than SpamAssassin. All of the above features have contributed to improve Wirebrush4SPAM performance.

We should note that some of the proposals included in this work have been successfully used in different domains to improve the overall execution speed of programmed applications. The utilization of compiled languages and caching techniques has become essential to develop software products with real-time or fast execution requirements. As an example, most popular MTAs (including Sendmail or Postfix), database managers and operating systems are written in C language and make use of caching alternatives to improve their global performance. Moreover, the idea behind our SFE technique is based on lazy evaluation schemes implemented by compilers to optimize the execution of conditional sentences. This approach can be also successfully applied to improve third-party filtering software based on the SpamAssassin framework.

Current and future work on Wirebrush4SPAM includes (i) new facilities like more customization capabilities and a Web environment giving support to both the definition of rules and configuration options, (ii) new prescheduling techniques, (iii) improvement of naïve Bayes classifier to increase the accuracy of spam filtering [26] and (iv) the development of new filtering techniques as auto white list or Vipul’s Razor [27] currently available in SpamAssassin.

Wirebrush4SPAM is a free software distributed under the terms of GNU lesser general public license and both the source code and documentation can be publicly accessed from our website [28].

ACKNOWLEDGEMENTS

This work was partially funded by the projects Optimización de sistemas antispam (08TIC041E), Deseño e validación de filtro antispam intelixente baseado en análise contextual ponderado do contido das mensaxes (09TIC028E) and Sistema aberto de filtrado antispam para xestores de contidos dixitais (10TIC017E) from Xunta de Galicia, Spain.

REFERENCES

16. Lehtinen J. C-Pluff, a plugin framework for C. Available at: http://libplugin.sourceforge.net/intro.html [last accessed 16 May 2011].